Proceedings of

MCHPC'19: Workshop on Memory Centric High Performance Computing

Held in conjunction with

SC19: The International Conference for High Performance Computing, Networking, Storage and Analysis

Denver, Colorado, November 17-22, 2019

MCHPC'19: Workshop on Memory Centric High Performance Computing

Table of Contents

Unified and Heterogeneous Memory

Converging Storage and Memory

Panel

Software and Hardware Support for Programming Heterogeneous Memory

Moderator: Maya B Gokhale (Lawrence Livermore National Laboratory)
Panelist: Mike Lang (LANL), Jeffrey Vetter (ORNL), Vivek Sarkar (Georgia Tech), David Beckinsale (LLNL), Paolo Faraboschi (HPE)

Organization

Organizers
Yonghong Yan, University of North Carolina at Charlotte, USA
Ron Brightwell, Sandia National Laboratory, USA
Xian-He Sun, Illinois Institute of Technology, USA
Maya B Gokhale, Lawrence Livermore National Laboratory, USA

Program Committee
Ron Brightwell, Co-Chair, Sandia National Laboratory, USA
Yonghong Yan, Co-Chair, University of North Carolina at Charlotte, USA
Xian-He Sun, Illinois Institute of Technology, USA
Maya B Gokhale, Lawrence Livermore National Laboratory, USA
Mingyu Chen, Chinese Academy of Sciences, China
Bronis R. de Supinski, Lawrence Livermore National Laboratory, USA
Tom Deakin, University of Bristol, UK
Hal Finkel, Argonne National Laboratory and LLVM Foundation, USA
Kyle Hale, Illinois Institute of Technology, USA
Jeff R. Hammond, Intel Corporation, USA
Dong Li, University of California, USA
Scott Lloyd, Lawrence Livermore National Laboratory, USA
Ivy B. Peng, Oak Ridge National Laboratory, USA
Christian Terboven, RWTH Aachen University, German
Alice Koniges, University of Hawaii, Maui High Performance Computing Center, USA
Arun Rodrigues, Sandia National Laboratory, USA
Chunhua Liao, Lawrence Livermore National Laboratory, USA

Keynote Talks

Prospects for Memory, J. Thomas Pawlowski

Abstract: The outlook for memory components and systems is at the all-time height of excitement. This talk examines the path we have traversed and the likely future directions we must pursue. We will examine scaling across many parameters, "walls" of all sorts concluding with the one that has emerged as the true wall, the dynamics of heterogeneity in systems and their comprising memories, the need for abstraction (still unsatisfied), options available for processing in or nearer to memory, and the imperative technology redirection that is required to make significant strides forward. Time permitting, a lively question and answer period is anticipated.

Brief Bio: J. Thomas Pawlowski is a systems architect and multi-discipline design engineer currently self-employed as a consultant and entrepreneur. He has recently retired from Micron Technology, Inc. where he was employed for 27 years, holding titles including Senior Director, Fellow, Chief Architect and Chief Technologist. He was at the center of numerous new memory architectures, technologies and concepts including synchronous pipelined burst memory, zero bus turnaround memory, double data rate memory, quad data rate memory, reduced latency memory, SerDes memory, multi-channel memory, 3D memory, abstracted memory, smart memory, non-deterministic finite automata (processing in memory technology), processing in interbank memory regions, processing in memory controllers, processing in NAND memory, processing in 3D memory stacks, memory controllers, 3DXpoint memory management, and cryogenic memory among others. Prior to Micron he served for almost 10 years at Allied-Signal Aerspace/Garrett Canada (now Honeywell). In his colorful career Thomas has many design firsts including first FPGAs, microcontrollers and custom microprocessors in aerospace applications, one of the early laptop computer designs (with perhaps the world's first SSD comprising NOR Flash), a novel electronic musical instrument, a line of high-end loudspeakers, and a from-scratch 2-seat electric vehicle design achieving nearly 400mpge efficiency. Thomas holds a BASc degree in electrical engineering from the University of Waterloo and is an IEEE Fellow. Thomas is available for consulting opportunities and would consider employment offers too good to refuse.

Performance Evaluation of the Intel Optane DC Memory With Scientific Benchmarks

Vladimir Mironov
Department of Chemistry
Lomonosov Moscow State University
Moscow, Russian Federation
vmironov@lcc.chem.msu.ru

Igor Chernykh
Supercomputing lab
ICMMG SB RAS
Novosibirsk, Russian Federation
chernykh@sscc.ru

Igor Kulikov
Supercomputing lab
ICMMG SB RAS
Novosibirsk, Russian Federation
kulikov@ssd.sscc.ru

Alexander Moskovsky
RSC Technologies
Moscow, Russia
moskov@rsc-tech.ru

Evgeny Epifanovsky
Q-Chem, Inc.
Pleasanton, CA, USA
epif@q-chem.com

Andrey Kudryavtsev
Intel Corporation
Folsom CA, USA
andery.o.kudryavtsev@intel.com

***Abstract*—Intel Optane technology is a cost-effective solution to create large non-volatile memory pools, which is attractive to many scientific applications. Last year Intel® Optane™ DC Persistent Memory in DIMM (PMM) form-factor was introduced, which is capable of being used as main memory device. This technology promises faster memory access comparing to Intel® Optane™ DC P4800X SSD with Intel® Memory Drive Technology devices available previously.**

In this paper, we present new benchmark data for the Intel Optane DC Persistent Memory. We studied the performance of scientific application from domains of quantum chemistry and computational astrophysics. To put performance of the Intel Optane DC PMM in comparison, we used two memory configurations: DDR4 only system and Optane DC SSD with Intel Memory Drive Technology (IMDT) that is another option for memory extension. We see that PMM is superior to IMDT in almost all our benchmarks, to no surprise. However, PMM was 20-30% more performant in quantum chemistry and only marginally better for astrophysics simulations. We also found, that for practical calculations hybrid setup demonstrates on the same order of magnitude performance as DDR4 system.

Keywords—Non-volatile memory, Intel Optane DC, benchmarks

I. Introduction

Scientific applications are one of the most important consumers of high-performance computing (HPC) resources. They need not only the central processing unit (CPU) power: complex scientific problems usually operate large data volumes. The memory capacity of a node is a common practical limit for scientific calculations. In the past decade, the number of CPU cores per node increased dramatically, from few to more than a hundred. In the same time, the memory capacity per core and thus the amount of memory per process almost did not changed.

An important limitation for extending the memory capacity of a node is high stock price for dynamic random-access memory (DRAM) chips, which is a usual choice for the system memory. As a result, DRAM expenses can take up to 80-90% of the total price for a computer system with large DRAM pool. It is not a surprise that a lot of efforts have been made to find a less expensive and possibly non-volatile alternative for DRAM [1]–[5].

Traditionally, disk drives or persistent memory devices were employed to overcome DRAM limits. The modern attractive option is to use solid-state drives (SSDs). However, even fastest SSDs have orders-of-magnitude less bandwidth and higher latency than DRAM; also, these devices usually operate data is large blocks instead of bytes. Recently Intel introduced a new Intel Optane DC SSDs based on the low latency Optane media technology [6]. In contrast to traditional SSDs they have much lower latency, higher IOPS performance under low queue depth and higher endurance, even if NMVe interface is employed by SSD. In addition to that Optane DC SSD can be used as memory extension of traditional DDR4 with Intel Memory Drive technology [7]. It has been shown [8] that using Optane drives as a software-defined memory (Intel Memory Drive Technology, IMDT) brings acceptable system performance but with a much less costs. However, this setup inherits the block-based access to memory from traditional SSDs and it suffers from operating system overheads when using data on disks [9].

In the current study, we investigate performance of next-generation Intel Optane DC Persistent Memory Modules (PMM). It's the DIMM form factor product and can operate in two modes. In App Direct mode it operates in addition to DDR4 and allows to realize a persistency through PMEM library [10]. In Memory Mode it uses DDR4 memory as cache and transparently allows bigger volatile memory with up to 6TB capacity even for dual-socket systems [11]. The latter configuration should not have the above mentioned limitations of block devices resulting in improved latency and bandwidth.

The performance studies of Intel Optane DC PMM is an active research topic [12]–[14]. Our interest is to investigate the performance of this novel hardware with real-life scientific problems. Two numerical simulation packages were tested: one is a quantum chemistry package Q-Chem [15] and another one is an astrophysical hydrodynamics framework HydroBox3D [16]. The Q-Chem was chosen because its implementation of coupled-cluster methods is considered as the best-of-breed, while those methods usually produce huge amounts of intermediate data. HydroBox3D is a recently developed software with fine optimizations to novel Intel AVX instruction set, limited mostly by main computer memory volume in producing more precise astrophysical simulations. In both cases,

we used PMM in memory mode, which does not require any software modification.

The paper is organized as follows. First, we give a short introduction to the novel Intel Optane PMM technology (Section 2). A brief overview of related papers on PMM benchmarking is given in Section 3. The methodology and benchmark description are presented in Section 4 of this paper. In Sections 5 and 6, we present and discuss the performance results. The concluding remarks are given in Section 7.

II. Overview of Intel Optane DC PMM

Non-volatile Intel Optane DC PMM memory features much higher capacities and lower power consumption than DRAM memory, but at cost of increased latency and lower bandwidth. Intel Optane PMM is currently supported only by the 2nd generation Intel Xeon Scalable processors. This is because PMM communicate with CPU memory controller by a proprietary DDR-T protocol, which is implemented only in this type of processors. DDR-T has a lot in common with a standard DDR4 protocol, but accounts for non-volatile memory specifics. On the Optane side, memory accesses are processed by the Optane memory controller. It is responsible for scheduling I/O operations to the Optane media, handling address mapping, data encryption, power/thermal control, etc. It also includes a small DRAM buffer for the address indirection table. It is worth noting, that while DDR-T protocol uses 64B data granularity (same to DDR4), data access to Optane media is 256B wide.

It was mentioned before, that Intel Optane PMM can operate in two modes: App Direct mode and memory mode. In App Direct mode, PMM is directly exposed to the operating system as non-volatile memory device. It is separated from the main memory and in order to use it one needs to rely on specialized libraries, like PMDK [10] or install a file system on it. In this mode the programmer control every aspect of using data on PMM.

In memory mode, PMM are used to expand main memory of a node transparently to the operating system and the user. DRAM is used as a sophisticated combination of inclusive and non-inclusive caches [17] for Intel Optane DC memory. Since part of the data always resides in DRAM, memory mode does not guarantee persistency.

III. Related work

Since its inception, several performance evaluations of Intel Optane DC PMM technology have been reported. A detailed study of PMM performance in App Direct and Memory modes with synthetic benchmarks is presented in works [12]–[14]. The paper [14] also focuses on power efficiency of PMM-enabled nodes and reported advanced memory allocation techniques, which can improve memory bandwidth or latency. The authors of paper [14] also studied graph-processing frameworks for performance. In the work [12] authors additionally used SPEC and PARSEC benchmark suits to compare the performance of PMM with DRAM. Papers [12], [18] report the performance of database software using PMM. The work [19] is focused on the performance of PMM in App Direct mode in virtualized environment.

Most of these papers note the performance difference between read and write operations on PMM: writing data on Optane can be up to three times slower than reading it. Another important observation is that PMM performance degrades when large number of threads are used or data is accessed on remote NUMA node.

However, the performance of numerical simulation software has not been studied in details. With the current paper we focus on the real-life examples of scientific problems that need large amount of memory.

IV. Methodology

A. Hardware configuration

In current work we used dual-socket 2nd Generation Intel Xeon Scalable Gold 6254 node. Each CPU has six memory channels and three UPI links. Three hardware setups were used for benchmarks:

1) DRAM only. In this setup the node was equipped with a 24×64 GB DRAM memory (1,536 GB total).

2) Optane SSD with Intel Memory Drive Technology (IMDT). In this configuration we used four Optane SSD P4800X drives to use as a part of IMDT software-defined memory. IMDT was configured to use only 192 GB of all DRAM memory as a cache and 1,328 GB of Optane memory (1.5 TB total memory).

3) Intel Optane DC PMM. In the third setup a mixture of 12×16GB DRAM modules and 12×128GB of the novel Optane DC DIMM modules in memory mode were used. The resulting system memory size was 1.5TB.

The detailed node configurations for all the above cases are presented in Table 1.

TABLE I. Hardware configuration

	Optane SSD and Intel Memory Drive Technology	**Intel Optane DC Persistent Memory**	**DRAM only**
DRAM memory	12x16GB reg. ECC DDR4	12x16GB reg. ECC DDR4	24x64 GB reg. ECC DDR4
Optane memory	4x 375GB Intel Optane DC P4800X (320GB per drive in IMDT memory mode)	12x128GB Intel Optane DC PMM (memory mode)	-
CPU	2x Intel Xeon Gold 6254 (2x18 cores, 3.1 GHz base)		
Board	Intel Server Board S2600WFT, BIOS FW 1.93.870CF4F0		
Storage	OS: 375 GB Intel SSD DC S3700 Scratch: 1.5 TB Intel Optane DC P4800X		
OS	CentOS Linux 7.6, kernel ver. 3.10.0		
IMDT F/W	9.0.3365.41	N/A	
Software	Intel Parallel Studio XE 2019.4; Q-Chem v 5.2		

B. Benchmark description

In current study, we used two families of benchmarks. In the first one, we perform high-level quantum chemistry calculations using Q-Chem program package. It is a popular quantum chemistry software with a proprietary license. Q-Chem contains highly efficient implementations of various quantum chemistry methods. In this study, we targets coupled cluster method with

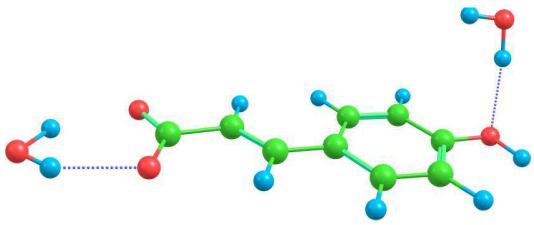


Fig. 1. Chemical system used in EOM-CCSD benchmarks. Carbon, oxygen and hydrogen atoms are colored in green, red and blue respectively.

single and double excitations (CCSD), which is routinely used for precise calculation of properties of small-to-medium size molecules. The computational complexity for CCSD benchmark is $O(N^6)$. Here N denotes the size of the basis set used and depends almost linearly on the number of atoms. This kind of calculation consumes large amount of memory to store intermediate data (two-electron integrals and excitation amplitudes) and thus it is a good candidate for Optane DC PMM benchmarking.

The other one is astrophysical hydrodynamics framework HydroBox3D. It is a novel astrophysical package which combines astrophysical codes AstroPhi [20], GPUPEGAS [21], and PADME [22]. As a result, this framework can be used for complex simulations, which accounts for both classical and relativistic hydrodynamics, chemical reactions and magnetic interactions. An example is modeling of supernovae type Ia explosions (see [23]), which occurs when a white dwarf mass reaches Chandrasekhar limit and used by astronomers as a "standard candles" when measuring distance to remote galaxies. Unfortunately, modeling of some physical and chemical processes cannot be performed efficiently in multi-node environment. In this case, a single node running multiple threads is used. The node must contain enough memory to fit all grid data, which is typically of terabyte scale. Thus, it is also a relevant benchmark for novel Optane memory.

Both these benchmarks were run on all three hardware configurations described above. The details of each benchmark are presented below in corresponding sections.

1) Q-Chem

In this benchmark, we computed CCSD energy and analytical gradient on the electronic ground and first excited states of the PYP chromophore system with two water molecules (see Fig. 1), using aug-cc-pVDZ basis set. The total number of basis set functions is 421. For the excited state calculation, we used equation-of-motion (EOM) CCSD method. PYP chromophore is a model molecule for photophysical properties of fluorescent proteins, which are common tool in biotechnology. We used Q-Chem v5.2 to perform all quantum chemistry benchmarks.

The time-consuming part of CCSD energy algorithm is the iterative, self-consistent step of computing amplitudes for singles and doubles excitation operators. It involves a set of tensor contractions to compute molecular orbital integrals and values of singles and doubles projections. Many of these operations depends on the current values of amplitudes and need to be repeated on every CCSD iteration. The CCSD gradient calculation involves several additional steps, which also based on tensor contraction operations.

CCSD tensor contraction routines involve multiple matrix multiplications. In theory, they are compute-bound, but they still may suffer from bandwidth problems, due to relatively small size of matrices and vectors. In practice, Q-Chem coupled cluster implementation performance may depend on the bandwidth of the memory and storage. Q-Chem uses statically linked Intel MKL library for performing dense linear algebra calculations. The details of tensor contraction implementation in Q-Chem can be found elsewhere [24].

Another time-consuming operation of CCSD method is DIIS (Direct Inversion in the Iterative Subspace) optimization of CCSD amplitudes. DIIS method consumes a lot of memory to store amplitudes and errors from several CCSD iterations. It is usually bandwidth bound.

We have used five CCSD benchmarks. The first two benchmarks are CCSD and EOM-CCSD calculations that use Cholesky decomposition (CD) of electron-repulsion integrals tensor. The next two benchmarks are CCSD and EOM-CCSD calculations, which use resolution of identity (RI) approach to transform four-center electron repulsion integral to three-center integrals. RI-CCSD and CD-CCSD have somewhat reduced memory requirements comparing to the regular CCSD method, which can affect the performance of hybrid memory setups. The last benchmark is regular ground-state CCSD energy and analytical gradient calculation taken for reference. In all EOM-CCSD calculations, the gradient was computed for the excited state.

In all Q-Chem benchmarks, we used 1.3 TB of memory, of which 1 TB was used for intermediate data of coupled cluster method. The final memory footprint exceeds 196 GB of DRAM in hybrid Optane+DRAM configurations and approaches 1 TB limit of CCSD memory. We used 36 threads to perform calculations.

2) HydroBox3D

In this benchmark, we simulated white dwarf evolution using AstroPhi module for numerical solution of hydrodynamics equations. This solver uses novel numerical methods based on a combination of Godunov's method for conservation laws by calculating fluxes through the boundaries [25], operator splitting method to construct a scheme to approximate the advection terms invariant with respect to rotation [25]–[27], and Rusanov's method to solve Riemann problems [28] for determining the fluxes with vectorization of the calculations [29]. There are two main approaches can be used for numeral hydrodynamics simulations of astrophysical problems: the Lagrangian smooth particle hydrodynamics methods and Eulerian mesh-based methods. The most important problem of Lagrangian smooth particle hydrodynamics methods is the inaccurate computation of large gradients and discontinuities for astrophysical problems. This is why mesh-based piecewise parabolic method on the local stencil (PPML) was used in the solver. To solve the Riemann problems, a compact scheme for a piecewise-parabolic representation of the solution in each of the directions is used [30]–[32]. The advantages of this numerical scheme are: 1) high scalability; 2) accuracy on sufficiently smooth solutions and low dissipation on discontinuous solutions; 3) guaranteed non-

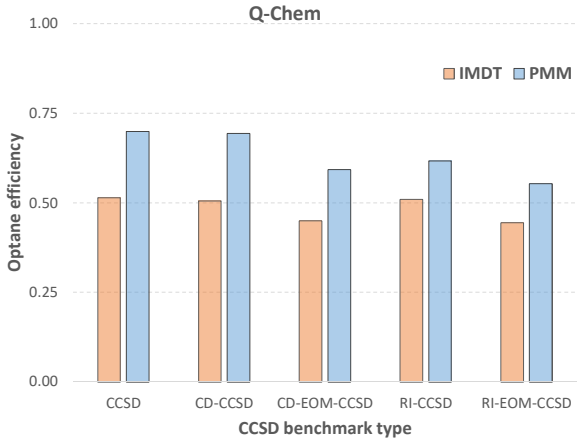


Fig. 2. IMDT and PMM efficiency plots for Q-Chem CCSD energy+gradient benchmark. By efficiency we mean the relative performance of the benchrmark on hybrid memory and on DRAM (T_{Optane}/T_{DRAM}). Higher efficiency is better. Value 1.0 corresponds to DRAM performance.

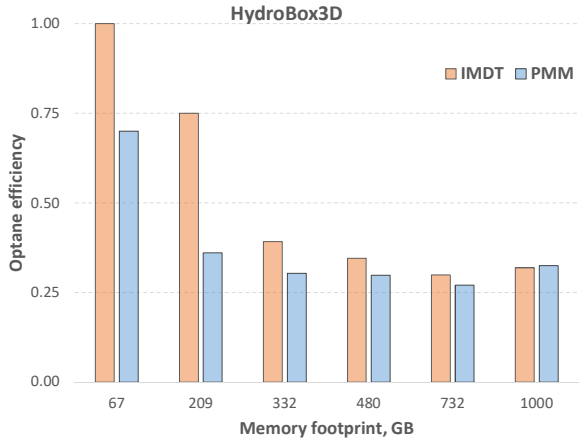


Fig. 3. IMDT and PMM efficiency plots for AstroPhi solver which was run as a part of HydroBox3D framework. By efficiency we mean the relative performance of the benchrmark on hybrid memory and on DRAM (T_{Optane}/T_{DRAM}). Higher efficiency is better. Value 1.0 corresponds to DRAM performance.

decrease of the entropy; 4) extensibility by hyperbolic models; 5) limiter-free and artificial-viscosity-free implementation; 6) Galilean invariance. The numerical scheme is considered in detail in paper [29]. The bottleneck of this calculation is the Lagrangian stage that is parallelized using OpenMP. We used 44 OpenMP threads to perform calculations. This amount of threads is bigger than quantity of hardware threads, but for this benchmark the best performance is observed when the node is slightly oversubscribed [29].

We tested different memory configurations on six tests with increasing grid sizes (see Table 2 for details). The largest workload size was 1TB. Each test was ran for at least three times on each node configuration from Table 1. In this benchmark, we collected an average time of the Lagrangian stage of the AstroPhi solver, which is a dominant contribution to the total time.

TABLE II. HYDROBOX3D MEMORY FOOTPRINT IN BENCHMARKS.

Grid box dimension	1024	1512	1768	2000	2304	2560
Memory size, GB	67	209	332	480	732	1000

V. RESULTS

1) Q-Chem

As expected, the highest performance values are observed for DRAM-configured systems. The efficiency plot for hybrid Optane+DRAM setups versus pure DRAM system is presented in Fig. 2. For CCSD energy and gradient calculation, the hybrid PMM memory is in average 35% slower than DRAM-only node configuration. The IMDT configuration is roughly two times slower than DRAM and 25% slower than Optane DC PMM. Slightly reduced efficiency of hybrid Optane node configurations vs DRAM is observed for RI family of methods. For regular CCSD and CD-CCSD calculation the efficiency of novel Optane DC PMM setup approaches 70%. Also, EOM-CCSD method appears to be 5-10% less efficient on both Optane-based memory setups than corresponding CCSD calculations.

2) HydroBox3D

The results of the HyperBox3D benchmark are presented in Fig. 3. According to these data, Both Optane-based node configurations are approximately three times slower than DRAM-based configuration, when the workload size exceeds the DRAM capacity of the node (196 GB in both cases). On small workload sizes IMDT approach is superior to PMM. At higher values of memory footprints the performance of IMDT and PMM become close to each other, but IMDT is somewhat more efficient, than PMM. Only when the workload size reaches 1 TB, PMM slightly overtakes IMDT. We attribute it to the NUMA effect. In case of IMDT and PMM we have two NUMA memory regions of 750 GB size, which correspond to CPU sockets. By default NUMA policy, memory is allocated first on the one NUMA node until no space left, then on another one. Hence, if only one NUMA node is used, then only half of the Intel Optane DC PMMs and thus only the half of their aggregated bandwidth. IMDT performance is slightly more efficient in this case due to its ability to prefetch the data to the DRAM slot that is local to CPU. The same effect is not pronounced in quantum chemistry benchmark due to much higher arithmetic intensity of the latter (BLAS lvl. 3 in Q-Chem versus stencil numerical methods in HydroBox3D).

VI. DISCUSSION

As it was shown in our previous paper [8], IMDT hybrid memory pools composed from DRAM memory and previous generation Intel Optane drives demonstrate comparable performance as DRAM-only memory in certain scientific benchmarks. To run efficiently on this hybrid setup an application kernel must have arithmetic intensity high enough to compensate for a long time needed to deliver data from memory to CPU. The novel Optane DC PMM have somewhat improved bandwidth and latency characteristics, but the performance gap between PMM and DRAM is still tangible.

We observe different behavior of our benchmark applications on IMDT and PMM configurations. We attribute

this to dissimilar memory access patterns and arithmetic intensity of computational kernels. PMM differs from IMDT in a number of ways. First, the latter does not have an ability to analyze data access patterns and to prefetch data onto DRAM before CPU needs it. It makes it less preferred for certain workloads with a complex access patterns, when it is possible to overlap computation and data access. Instead, PMM relies on hardware supported caching mechanisms for the same purpose. This is good if a workload spends a lot of time working with the same piece of dataset (temporal locality). Second, PMM offers higher bandwidth and lower latencies when accessing Optane memory, which is beneficial on random or stream access patterns. However, in practice, not all workloads can utilize new Intel Optane PMM efficiently.

Generic recommendation would include utilizing application and system profiling tools, such as Intel ® VTune ™ Amplifier or Platform Profiler package to identify system bottlenecks and monitor memory bandwidth utilization separately for DRAM cache and Optane DC PMM. Also standard practice for NUMA locality optimizations would apply here as well cross NUMA traffic monitoring is important as well [33].

VII. Conclusion

Intel Optane DC PMM is a first massive commercially available non-volatile DIMM product. The bandwidth and latency characteristics puts it between DRAM and SSD tiers of memory hierarchy. It can be used as a main computer memory extension by hardware only. It was also possible with previous Intel Optane generation, which is an NVMe device. However, it requires a proprietary software-defined memory solution like Intel Memory Drive Technology.

A lot of scientific problems, which are only limited by the system memory capacity can benefit from Intel Optane PMM. However, PMM advantages over IMDT is not always sound, as we see in our benchmark tests. In any case, Intel Optane technology becomes an affordable solution for large memory configurations, employed for numerical simulations.

Integrating hybrid memory technologies is critical for modern HPC infrastructure. Such memory is not faster than traditional DDR4 memory however for certain applications heavy on the cross-rank communications this is a way to reduce the number of compute nodes and fabric overhead, move it under bigger memory domain and potentially reduce the runtime. Also with adopting different sort of the accelerator technologies in HPC, host memory capacity plays significant role for the data path to reduce storage bottleneck. This is another opportunity for hybrid memory solutions. This is a scope of the following study to explore great possibilities of the technology.

Acknowledgment

I. C. and I. K. thank Russian Foundation for Basic Research (project no. 18-07-00757, 18-01-00166) and budget project No. 0315-2019-0009. V. M. and A. M. thank the Russian Science Foundation (project 19-73-20032). We would like to thank Siberian Supercomputer Center for providing access to HPC facilities.

References

[1] H. Akinaga and H. Shima, “Resistive random access memory (ReRAM) based on metal oxides,” *Proc. IEEE*, vol. 98, no. 12, pp. 2237–2251, 2010.

[2] D. Apalkov *et al.*, “Spin-transfer torque magnetic random access memory (STT-MRAM),” *ACM J. Emerg. Technol. Comput. Syst.*, vol. 9, no. 2, 2013.

[3] J. Handy, “Understanding the Intel / Micron 3D XPoint Memory,” 2015.

[4] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, “Architecting phase change memory as a scalable DRAM alternative,” *Proc. - Int. Symp. Comput. Archit.*, pp. 2–13, 2009.

[5] G. W. Burr *et al.*, “Phase change memory technology,” *J. Vac. Sci. Technol. B, Nanotechnol. Microelectron. Mater. Process. Meas. Phenom.*, vol. 28, no. 2, pp. 223–262, 2010.

[6] Intel Corporation, “Intel Optane Technology.” [Online]. Available: http://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html. [Accessed: 29-Jan-2017].

[7] Intel Corporation, “Intel Memory Drive Technology.” [Online]. Available: https://www.intel.com/content/www/us/en/software/intel-memory-drive-technology.html. [Accessed: 30-Aug-2019].

[8] V. Mironov, A. Moskovsky, A. Kudryavtsev, I. Kulikov, Y. Alexeev, and I. Chernykh, “Evaluation of intel memory drive technology performance for scientific applications,” *ACM Int. Conf. Proceeding Ser.*, pp. 14–21, 2018.

[9] A. Eisenman *et al.*, “Reducing DRAM footprint with NVM in facebook,” in *Proceedings of the Thirteenth EuroSys Conference on - EuroSys ’18*, 2018, vol. 2018-Janua, pp. 1–13.

[10] The PMDK team, “Persistent Memory Programming.” [Online]. Available: https://pmem.io.

[11] Intel Corporation, “Intel Optane DC Persistent Memory.” [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html. [Accessed: 30-Aug-2019].

[12] J. Izraelevitz *et al.*, “Basic Performance Measurements of the Intel Optane DC Persistent Memory Module,” 2019.

[13] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, “An Empirical Guide to the Behavior and Use of Scalable Persistent Memory,” Aug. 2019.

[14] I. B. Peng, M. B. Gokhale, and E. W. Green, “System Evaluation of the Intel Optane Byte-addressable NVM,” Aug. 2019.

[15] Y. Shao *et al.*, “Advances in molecular quantum chemistry contained in the Q-Chem 4 program package,” *Mol. Phys.*, vol. 113, no. 2, pp. 184–215, 2015.

[16] I. Kulikov, I. Chernykh, and A. Tutukov, “A New Hydrodynamic Code with Explicit Vectorization Instructions Optimizations that Is Dedicated to the Numerical Simulation of Astrophysical Gas Flow. I. Numerical Method, Tests, and Model Problems,” *Astrophys. J. Suppl. Ser.*, vol. 243, no. 1, p. 4, 2019.

[17] M. Arafa *et al.*, “Cascade Lake: Next Generation Intel Xeon Scalable Processor,” *IEEE Micro*, vol. 39, no. 2, pp. 29–36, 2019.

[18] K. Wu, A. Arpaci-Dusseau, R. Arpaci-Dusseau, R. Sen, and K. Park, “Exploiting Intel Optane SSD for Microsoft SQL Server,” *Proc. 15th Int. Work. Data Manag. New Hardw. - DaMoN’19*, pp. 1–3, 2019.

[19] Q. Ali and P. Yedlapal, “Persistent Memory Performance in vSphere 6.7 with Intel Optane DC Persistent Memory,” 2019. [Online]. Available: https://www.vmware.com/techpapers/2018/optane-dc-pmem-vsphere67-perf.html. [Accessed: 30-Aug-2019].

[20] I. M. Kulikov, I. G. Chernykh, A. V. Snytnikov, B. M. Glinskiy, and A. V. Tutukov, “AstroPhi: A code for complex simulation of the dynamics of astrophysical objects using hybrid supercomputers,” *Comput. Phys. Commun.*, vol. 186, pp. 71–80, 2015.

[21] I. Kulikov, “GPUPEGAS: A new GPU-accelerated hydrodynamic code for numerical simulations of interacting galaxies,” *Astrophys. Journal, Suppl. Ser.*, vol. 214, no. 1, 2014.

[22] V. Protasov, I. Kulikov, I. Chernykh, and I. Gubaydullin, “PADME - New code for modeling of planet georesources formation on heterogeneous computing systems,” *MATEC Web Conf.*, vol. 158, 2018.

[23] I. Iben, Jr. and A. V. Tutukov, "On the Evolution of Close Triple Stars That Produce Type Ia Supernovae," *Astrophys. J.*, vol. 511, no. 1, pp. 324–334, 1999.

[24] I. A. Kaliman and A. I. Krylov, "New algorithm for tensor contractions on multi-core CPUs, GPUs, and accelerators enables CCSD and EOM-CCSD calculations with over 1000 basis functions on a single compute node," *J. Comput. Chem.*, vol. 38, no. 11, pp. 842–853, Apr. 2017.

[25] S. K. Godunov and I. M. Kulikov, "Computation of discontinuous solutions of fluid dynamics equations with entropy nondecrease guarantee," *Comput. Math. Math. Phys.*, vol. 54, no. 6, pp. 1012–1024, 2014.

[26] V. A. Vshivkov, G. G. Lazareva, A. V. Snytnikov, I. M. Kulikov, and A. V. Tutukov, "Computational methods for ill-posed problems of gravitational gasodynamics," *J. Inverse Ill-Posed Probl.*, vol. 19, no. 1, pp. 151–166, 2011.

[27] I. Kulikov, G. Lazareva, A. Snytnikov, and V. Vshivkov, "Supercomputer simulation of an astrophysical object collapse by the fluids-in-cell method," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 5698 LNCS, pp. 414–422, 2009.

[28] V. V. Rusanov, "The Calculation of the Interaction of Non-Stationary Shock Waves with Barriers," *Zh. Vychisl. Mat. Mat. Fiz.*, vol. 1, no. 2, pp. 267–279, 1961.

[29] I. M. Kulikov, I. G. Chernykh, and A. V. Tutukov, "A New Parallel Intel Xeon Phi Hydrodynamics Code for Massively Parallel Supercomputers," *Lobachevskii J. Math.*, vol. 39, no. 9, pp. 1207–1216, 2018.

[30] M. V. Popov and S. D. Ustyugov, "Piecewise parabolic method on local stencil for gasdynamic simulations," *Comput. Math. Math. Phys.*, vol. 47, no. 12, pp. 1970–1989, 2007.

[31] M. V. Popov and S. D. Ustyugov, "Piecewise parabolic method on a local stencil for ideal magnetohydrodynamics," *Comput. Math. Math. Phys.*, vol. 48, no. 3, pp. 477–499, 2008.

[32] I. Kulikov and E. Vorobyov, "Using the PPML approach for constructing a low-dissipation, operator-splitting scheme for numerical simulations of hydrodynamic flows," *J. Comput. Phys.*, vol. 317, pp. 318–346, 2016.

[33] Intel Corporation, "Using the Latest Performance Analysis Tools to Prepare for Intel Optane DC Persistent Memory." [Online]. Available: https://techdecoded.intel.io/resources/using-the-latest-performance-analysis-tools-to-prepare-for-intel-optane-dc-persistent-memory/. [Accessed: 30-Aug-2019].

Optimizing Data Layouts for Irregular Applications on a Migratory Thread Architecture

Thomas B. Rolinger*†, Christopher D. Krieger† and Alan Sussman*

* University of Maryland, College Park, MD USA
† Laboratory for Physical Sciences, College Park, MD USA

tbrolin@cs.umd.edu, krieger@lps.umd.edu, als@cs.umd.edu

***Abstract*—Applications that operate on sparse data induce irregular data access patterns and cannot take full advantage of caches and prefetching. Novel hardware architectures have been proposed to address the disparity between processor and memory speeds by moving computation closer to memory. One such architecture is the Emu system, which employs light-weight threads that migrate to the location of the data being accessed. While smart heuristics and profile-guided techniques have been developed to derive good data layouts for traditional machines, these methods are largely ineffective when applied to a migratory thread architecture. In this work, we present an application-independent framework for data layout optimizations that targets the Emu architecture. We discuss the necessary tools and concepts to facilitate such optimizations, including a data-centric profiler, data distribution library, and cost model. To demonstrate the framework, we have designed a block placement optimization that distributes blocks of data across the system such that access latency is reduced. The optimization was applied towards sparse matrix-vector multiplication on an Emu FPGA implementation, achieving a geometric mean speed up of 12.5% across 57 matrices. Only one matrix experienced a loss of performance of 6%, while the maximum runtime speedup was 50%.**

***Keywords*-migratory threads, data layout, optimizations, irregular applications**

I. Introduction

While processor speeds have steadily increased, memory speed has only achieved a fraction of that scaling. As a result, modern processors can become data-starved, as the memory system cannot keep up with the rate at which data is needed by the processors. Architectural advances such as cache hierarchies have reduced memory latency, allowing many applications to take advantage of the increase in processor speeds. However, applications that operate on sparse data, such as graphs and tensors, induce irregular data access patterns that pose significant challenges due to the lack of locality in their memory accesses.

Due to the importance of irregular applications for high performance data analytics, optimizing both the algorithms as well as the way in which sparse data is stored has been an active area of research [1], [2]. On the other hand, rather than adapting irregular applications to traditional computing systems, another approach is to develop novel architectures that address the challenges directly in hardware [3], [4]. While there have been several such approaches, a common theme among them is to provide fine-grained memory accesses and to move computation closer to memory. One recent example is the Emu migratory thread architecture [5]. Instead of bringing data through a cache-memory hierarchy to the processor, the Emu system uses light-weight migratory threads that physically move to the location of the data being accessed. By only transferring light-weight threads on remote memory accesses, the Emu system aims to reduce the total load on the memory system.

For many memory-bound applications, the way in which data is distributed across system resources is important for performance. For the Emu system, this is critically important because computation migrates to statically placed data, which means that data layout explicitly controls load balancing and utilization of compute resources. A poor data layout can lead to a significant surge of threads migrating to the same hardware resource, creating load imbalance.

An efficient algorithm to determine an optimal data layout that minimizes a metric such as cache misses is not only difficult to approximate, but it is in fact NP-hard [6]. In response, smart heuristics and profile-guided techniques have been developed to find "good" data layouts. However, traditional optimization methods are largely ineffective on a system like Emu due to fundamental architectural differences, such as a lack of hierarchical memory. Irregular applications also defeat many static data optimizations because their data access patterns are not known until runtime. To make matters worse, even when all memory accesses are known for a particular execution via profiling, it is still not obvious how the data should be laid out. Therefore, while data layouts are crucial to performance, there are significant challenges to determining a high performing layout for an irregular application on Emu.

In this work, we present a framework that facilitates data layout optimizations for the Emu architecture. This framework is application-independent, as it only relies on memory access behavior rather than knowledge about application intent or data structures. The primary goal of these data layout optimizations is to reduce memory access latency by avoiding thread migrations, while still maintaining a good load balance across the system resources.

Our contributions are as follows:

- We design and implement required tools to study and

optimize data layouts on the Emu system. These include a data block distribution library and a data-centric memory profiler.
- We develop a cost model for capturing the underlying performance characteristics of data block distributions on the Emu architecture. This cost model is used to guide optimizations.
- We design and implement a data block placement optimization that aims to redistribute blocks of data across the system such that their memory access latency is minimized.
- To demonstrate the cost model and block placement optimization, we perform a case study on sparse matrix vector multiply (SpMV) across 57 matrices. We show that data layouts can be generated for the input vector of SpMV that provide as much as a 50% reduction in runtime and a geometric mean of 12.5% when compared to a default round-robin layout. Furthermore, only one matrix had a performance loss after optimization of 6%.

The rest of this paper is organized as follows: Section II presents an overview of the current Emu architecture and relevant aspects of its programming model. Section III describes the foundational tools that were developed to facilitate data layout optimizations. The cost model and block placement optimization are presented in Section IV and V, respectively. Section VI presents the case study of SpMV and our performance results. Related work is discussed in Section VII. Finally, future work and conclusions are discussed in Section VIII.

II. Emu Architecture

The basic building block in an Emu system is a *nodelet*. A nodelet contains a number of cache-less, multi-threaded *Gossamer Cores*, banks of narrow channel DRAM, and *memory-side processors*. A group of nodelets, connected by a *migration engine* to provide a means for the movement of threads between nodelets, is called a *node* in the system. Groups of nodes are further connected over a communication network to form a complete Emu system.

A Gossamer Core (GC) is a general purpose processing unit developed specifically for the Emu architecture. The design complexity of a GC is significantly reduced relative to a traditional CPU due to the lack of caches, and thus, the lack of cache coherency logic. A single GC can support up to 64 concurrent light-weight threads, where each thread is limited to one active instruction at any given time.

One way that Emu addresses the challenge of irregular access patterns is by utilizing narrow channel DRAM (NCDRAM) on each nodelet. The NCDRAM modules consist of eight 8-bit channels rather than a single 64-bit interface. This provides fine-grained memory accesses that are better suited for applications that typically only need 8 bytes of data out of an entire cache line.

Attached to each bank of NCDRAM is a memory-side processor (MSP). A MSP can perform a *remote operation* on behalf of an executing thread, atomically. Remote operations

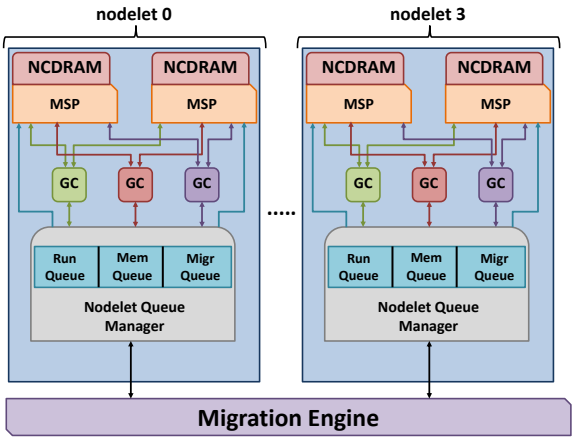


Fig. 1. A single node in the Emu Chick system. Each of the 4 nodelets contains 3 Gossamer Cores (GCs), two banks of narrow channel DRAM (NCDRAM) and two memory-side processors (MSPs).

do not return a result and do not migrate the entire thread, but instead generate a packet that encapsulates the operation to perform, the data on which to perform it, and the address at which to store the result (which also serves as a second data operand). Once the remote operation has completed, an acknowledgement is sent back to the issuing thread. The thread issuing the remote operation may continue execution after the packet has been sent off to the destination nodelet. However, a thread cannot migrate until all outstanding acknowledgements have been received. All writes to remote memory are automatically transformed into remote writes by the Emu compiler.

The current available Emu system is called the Emu Chick. It consists of 8 nodes, where each node is comprised of four nodelets. Figure 1 depicts a single node in the Emu Chick system. On each nodelet, there are three GCs and two banks of NCDRAM, each bank with its own MSP. A given nodelet can support up to 192 concurrently executing threads. The nodes are connected via a Serial RapidIO (SRIO) interconnect in a mesh-like network where each node has direct links to 6 other nodes; the remaining nodes require two hops for communication. Each node is implemented on an Arria 10 FPGA. More details regarding the Emu system and architecture can be found in the work by Dysart et al. [5].

Programs executed on an Emu system are written in C, where specialized software routines are provided to control data allocation (see Section II-B). Parallelism is controlled using the Cilk parallel extensions to C [7], where the `cilk_spawn` keyword will spawn a thread to execute a specified function. Users can give hints to `cilk_spawn` to indicate the nodelet where the thread should be created, providing a mechanism to initially distribute threads across the system.

A. Thread Migrations

When a thread executing on a GC issues a load to a remote memory location, the process of migrating that thread begins. The GC directs the Nodelet Queue Manager (NQM) to migrate the thread to the nodelet where the requested data is located.

The NQM is the entity that interfaces with the migration engine, MSPs and GCs to service the various queues shown in Figure 1. The thread is then packaged up into a context containing live registers, a program counter, stack counter, and status information [8]. The context is roughly 200 bytes long. This thread context is placed into the migration queue on the source nodelet to wait until it can be sent by the migration engine to its destination nodelet. When serviced by the migration engine, the thread context is sent to the remote nodelet's NQM, which places it into the run queue. Application developers are virtually unaware of migrations because the entire migration process is implemented in hardware.

B. Data Allocation and Placement

Emu implements a Partitioned Global Address Space (PGAS) composed of the memory from each nodelet. Emu provides a software library with routines to control how data is allocated, distributed, and accessed within this global address space. For allocation on a single-nodelet, the standard `malloc` routine is provided to allocate memory on the nodelet where the thread is executing. A modified allocation routine is also provided that allows memory to be allocated on a specified remote nodelet.

For distributed allocations, a simple block distribution routine is provided, where blocks of data are allocated and distributed across the nodelets. Users specify the number of blocks and the block size in bytes, but all blocks are required to have the same size. Furthermore, the location of the blocks is restricted to a cyclic, round-robin placement on the nodelets. The routine returns a 2-dimensional array, where it is the programmer's responsibility to handle the mapping from the original, conceptual index space to the blocked 2-dimensional indices, typically using integer division and modulo arithmetic. The Emu toolchain provides a utility library to assist with these index calculations.

C. Simulating and Profiling Applications

Emu also provides an architectural simulator that can execute the same programs that run on the hardware and output detailed metrics regarding thread migrations and memory accesses. As the current Emu hardware does not provide any performance counters, the simulator is crucial to understanding an application's behavior. The Emu simulator provides code-centric profiling support that includes thread and migration activity at the scope of functions and/or the entire program. However, to provide this data, the simulator must run in timed architectural mode, which increases the wall clock time required to simulate a program by more than 20 times over execution in untimed mode.

The profiler lacks direct support for data-centric views of the program, such as showing how many migrations were generated with respect to a specified data structure in the code. This type of information is key to determining an efficient data layout. Fortunately, the simulator is able to produce a basic memory trace of an executing program without running in timed architectural mode. From the memory trace, users can observe every memory instruction that was issued and details such as the source and destination nodelet, the address that was referenced in the instruction, and a cycle time-stamp of when the instruction was issued. With the memory trace, customized tools can be developed to perform more advanced profiling.

III. Prerequisite Tools

The goal of this work is to develop a framework for reasoning about data layouts on the Emu architecture, and to apply this framework to optimize data layouts by reducing the number of thread migrations incurred. Before we present our approach for optimization, we discuss the prerequisite tools and concepts that are necessary to define, manipulate and profile data layouts on Emu.

A. Block Distribution Library

As mentioned in Section II-B, Emu provides a simple data distribution routine that uses a cyclic round-robin block placement strategy across nodelets with fixed sized blocks. If a programmer chooses not to use the block distribution routine provided by Emu, he or she can create a customized layout, but will be responsible for directly coding data allocations and index mappings. Consequently, any changes to a custom layout, such as block sizes or locations, would require the application source code to be modified to account for the new index mapping.

To address these issues, we developed a block distribution library that supports variable block sizes as well as the flexibility to define block placement strategies beyond cyclic round-robin. Additionally, the library abstracts index mapping for the user. If the layout is subsequently modified, including by an optimizer, the user's code that accesses the data is unchanged. For distributions where all the blocks are the same size, the library will use optimized routines for indexing. These routines incur the same amount of overhead as those provided by the Emu toolchain. For distributions with variable block sizes, we adopt the approach used by Global Arrays [9], which does add overhead due to the complexity of finding the correct block for a given access.

B. Data-centric Memory Profiler

Another fundamental prerequisite for data layout optimizations is the ability to gather performance metrics related directly to a data layout, rather than the entire program. To address the limitations of Emu's code-centric profiling capabilities, we developed a data-centric profiler that can generate memory access profiles for specific allocations in a program, as specified by the user.

Our profiler is split into three distinct stages. First, we provide a light-weight API that allows users to directly specify within their code which data allocation they wish to profile. This involves "registering" the allocation and then making explicit calls to start and stop "tracking" the allocation, giving full control of when the profiling should take place. The second stage is executing the program within the Emu simulator. During this stage, two output files are generated: (1) tracking

TABLE I
LIST OF SYMBOLS AND THEIR DESCRIPTIONS WITHIN THE COST MODEL

Symbol	Description
N	number of nodelets, numbered from 1 to N
B	number of blocks, numbered from 1 to B
W	number of windows of activity, numbered from 1 to W
C	NxN map of memory access latency, in cycles, between nodelets
P	$1xB$ map of blocks to their current nodelet location
A	NxB map of the number of memory accesses between nodelets and blocks
L	NxB map of memory access latency, in cycles, between nodelets and blocks
T_{max}	maximum number of concurrent threads supported on a nodelet
L_b	total memory access latency, in cycles, for block b across all nodelets
L_μ	average total memory access latency, in cycles, across all blocks
$T_{blk}(b, w)$	number of active threads for block b during window w
$T_{ndlt}(n, w)$	number of active threads on nodelet n during window w
$Util_{blk}(b, w)$	resource utilization of block b during window w
$Util_{ndlt}(n, w)$	resource utilization of nodelet n during window w
$U_{blk}(b)$	median resource utilization of block b across all windows
$U_{ndlt}(n)$	median resource utilization of nodelet n across all windows
$\mathrm{PerformanceImpact}_b$	expected performance impact of block b
$\mathrm{PlacementCost}_{b,n}$	expected cost of placing block b on nodelet n

metadata and (2) a memory trace. The tracking metadata is generated by our API and includes the address ranges of the allocated blocks and the timestamps for when the profiling should occur. The memory trace is generated directly by the simulator, as discussed in Section II-C. The trace information is not stored in memory nor made available to the running application.

The metadata from our API and the memory trace file are ingested during the final step, which performs the actual profiling. A memory access event from the trace is attributed to a data allocation if it occurred during the relevant cycle range of profiling and if it accessed an address that was within the distributed blocks of that allocation. The output from the profiler consists of various files that describe metrics related to the block distribution such as incoming and outgoing memory accesses to/from each block. The profiler also breaks down these metrics across discrete windows of time. With a data-centric view of the program, we can gain insight into temporal behavior, such as how the read and write activity from/to a block changes over time.

IV. COST MODEL

We now have the ability to use a data distribution in a way that allows for easy modification and the ability to gather performance metrics for a given distribution. However, a cost model is needed to decide which blocks the optimizations should be applied to. The model needs to consider the penalty of migrations while also factoring in the load over time on the nodelets. In this section, we describe how we derived our cost model from both the profile data as well as hardware related metrics. In this work, we use the model only to evaluate the benefit of relocating blocks to different nodelets, but the model is general enough to guide other optimizations, which will be considered in future work.

In the following subsections, we describe the components of the model and how they are derived. We refer to the number of nodelets in the system as N and the number of blocks in the distribution as B. Table I presents the various symbols used in the cost model along with a brief description of their meaning.

A. Communication Cost Map

While the profiler produces metrics for how many memory accesses were made to each block of the distribution, it does not provide information about how expensive a given memory access is. The Emu system does not have a traditional memory hierarchy, but does exhibit non-uniform memory access latency depending on the source and destination nodelets involved in the access. Inter-node migrations have a higher latency than intra-node migrations since the former use the SRIO interconnect while the latter use the faster, crossbar-based migration engine. Furthermore, since the Emu Chick does not have a direct all-to-all communication topology between nodes, there are pairs of nodes where migrations between them have an even higher multi-hop latency.

In order to capture these costs, we developed a benchmark to measure memory access latency between all pairs of nodelets in the system. We store these costs within an NxN map **C**, where element $c_{i,j}$ is the cost of a migration, in cycles, from nodelet i to nodelet j. The diagonal entries in **C** represent local memory accesses and are weighted negatively. This is done to encourage local accesses by reducing the cost metrics derived from **C**.

We observe that with respect to local accesses, intra-node migrations require 2x as many cycles. For migrations between nodelets on separate nodes, which are connected with a single SRIO link, the cost is 3x as much as a local access. Finally, migrations between nodelets on separate nodes that require two hops over the SRIO network require 4x as many cycles as a local access.

We note that the cost model currently does not consider remote operations. The reason is that thread migrations are significantly more expensive, as they require moving the entire thread context and block the thread from executing any further instructions. Future work will incorporate remote operations

into the model, as they are relevant for other optimizations not considered in this work.

B. Placement Map and Block Memory Access Map

One of the defining features of a block distribution is the nodelet location of each block. This information is stored in a one-dimensional map **P**, where p_i represents the nodelet ID where block i is currently placed. This information is provided directly by the data distribution definition.

In addition to the placement of the blocks, the model needs to know how many accesses were made to each block. The profiler produces such information, which includes the source nodelet for each access. We store the memory accesses in a NxB map **A**, where $a_{i,j}$ represents the number of memory accesses from nodelet i to block j. We can derive a complete nodelet-to-nodelet trace of the memory accesses by looking up block j in **P**.

C. Block Latency Map

Given the **C** and **A** maps, we can compute the total memory access latency cost for each block with respect to a given nodelet by multiplying the number of accesses made to a block by the latency cost for those accesses. We store this information in a NxB map **L**. The total memory access latency incurred by block i across all nodelets can be computed by simply summing up the i^{th} column of **L**. We represent this total latency value for block b as L_b. As local accesses are negatively weighted, a block that is dominated by local accesses will have a negative total memory access latency.

D. Resource Utilization

The cost model described thus far only has a static view of the data layout performance. What it lacks is how the various blocks are accessed over the execution of the program, as well as the amount of activity on each nodelet over time. By only considering an end-of-program cost of memory accesses, optimizations may make poor decisions, such as relocating all the blocks to a single nodelet. The end result would be severe resource over-subscription, leading to increased execution time.

To address these issues, the cost model computes a resource utilization statistic over time. Our data-centric profiler provides various memory access statistics over discrete windows of time that are computed for every block and every nodelet. For a given block, the number of unique threads that made memory accesses to the block is logged for each window. Similarly, the number of unique threads that made memory accesses to any data on a nodelet is logged for each window. We consider such threads as *active threads* during a given window. This information is crucial when considering optimizations such as block placement because regardless of where a block is located, the threads will still issue accesses to that block. The cost model needs to be aware of how much "load" the block will bring to a nodelet when it is moved, as well as how much load is already on that nodelet.

We define the load of a block as the ratio of the number of threads that accessed the block to the maximum number of threads supported on a nodelet. Similarly, the load on a nodelet considers how many threads are active on the nodelet as compared to the maximum number of threads supported. The cost model computes the amount of over- or under-utilization, where 0 indicates perfect utilization, negative values indicate under-utilization and positive values indicate over-utilization. For a given block b and nodelet n, their respective utilization during window w is computed as:

$$Util_{blk}(b,w) = \frac{T_{blk}(b,w)}{T_{max}} - 1 \tag{1}$$

$$Util_{ndlt}(n,w) = \frac{T_{ndlt}(n,w)}{T_{max}} - 1 \tag{2}$$

where $T_{blk}(b,w)$ and $T_{ndlt}(n,w)$ represent the number of active threads for block b and nodelet n during window w, respectively. T_{max} is the number of concurrent threads supported per nodelet, which is 192 for the current Emu Chick system. Note that $T_{ndlt}(n,w)$ includes not only the threads that are accessing data layout blocks on nodelet n during window w, but threads that are accessing any memory on the nodelet. This allows the cost model to account for activity on the nodelet that is not directly related to the data layout in question, but is still relevant for performance.

The cost model takes the median of the utilization values across all windows to arrive at an estimate of the load over the program execution. We will refer to these median utilization values as simply $U_{blk}(b)$ and $U_{ndlt}(n)$ for block b and nodelet n, respectively.

E. Block Performance Impact

Because the migratory thread architecture moves computation to the nodelet that contains needed data, changing a block's placement also creates a rebalancing of work. This loading effect places an additional constraint on selecting a block for relocation. For example, if a block is heavily utilizing resources, it will have greater impact on performance to relocate it since the load that it induces may lead to over-subscription on its new nodelet. Likewise, if a block is lightly utilizing resources, it will be much easier to move to a different nodelet without negative performance consequences. For these reasons, the cost model computes the performance impact of a block by considering both the latency cost as well as the utilization induced by the block.

This performance impact cost is used to guide the selection of blocks for relocation and is computed as follows for a given block b:

$$\text{PerformanceImpact}_b = L_b + (L_\mu \times U_{blk}(b)) \tag{3}$$

where L_μ is the average memory access latency across all blocks. The purpose of using L_μ is to convert the utilization term into the same cycle units as L_b, as well as provide for a means to scale the performance impact by the degree of over- or under-utilization. The more over-utilized a block is, the higher its performance impact will be. Likewise, under-utilization will lower the performance impact. In future work,

we plan to better understand the effect of thread activity on the memory system and incorporate that into the model.

F. Block Placement Cost

When deciding on whether to relocate a block to a particular nodelet, it is necessary for the model to consider the impact on the block's latency cost as well as the impact on the utilization of the nodelet, which can be hosting multiple blocks. This is important to consider, as over-utilizing the nodelet can have negative performance effects for all blocks hosted there, not just the block being moved. To this end, the cost model computes a placement cost for block b on nodelet n as:

$$\text{PlacementCost}_{b,n} = L_b + (L_\mu \times U_{ndlt}(n)) \tag{4}$$

where L_b reflects the latency cost of b if it were on nodelet n and $U_{ndlt}(n)$ represents the median utilization of nodelet n, assuming that b were to be placed on n.

With respect to nodelets, the Emu system is capable of running more than T_{max} threads on a nodelet, but every thread that exceeds T_{max} will be idle until it can be scheduled. Like the block performance impact, the model discourages over-utilization of a nodelet by increasing the block's placement cost by a percentage of the average block latency.

V. Optimization: Block Placement

The block placement optimization is straightforward: place block b on the nodelet n that minimizes b's cost as given by the model. The high-level algorithm is presented in Algorithm 1. In the following subsections, relevant portions of the algorithm are explained in detail, as well as its complexity.

A. Block Selection and Sorting

Because we are focusing on irregular memory access patterns, some blocks will likely have higher performance impact than others. If the optimization attempts to move all of the blocks, we have observed that layouts can be generated that lead to significant performance loss (see Section VI-B). In order to select only the most beneficial blocks for relocation, the optimization will ignore any block with a negative performance impact, as such blocks are dominated by local accesses and/or are significantly under-utilized. Line 4 in Algorithm 1 performs this block selection procedure.

Before performing the actual block placement, it is important to determine the order in which the blocks will be moved. To this end, blocks with high performance impact are given priority (Line 5 in Algorithm 1). Moving these "heavy" blocks first is likely to yield the largest performance gains. We address the performance impact of this sorting in Section VI-B.

B. Nodelet Comparisons

The main goal of the block placement optimization is to find the nodelet n that produces the minimum placement cost for a given block b. To achieve this, the algorithm first recomputes b's latency as if it were located on the prospective nodelet n. If this recomputed latency, referred to as $L_{b'}$ in Algorithm 1, is lower than the current "minimum" latency for b, then n can proceed as a prospective target nodelet. By doing this, nodelets which do not reduce the memory access latency for a given block are excluded from consideration. This strategy targets our original goal of reducing memory access latency.

If placement on n would produce a lower latency cost for b, then the algorithm accounts for the utilization on n if b were to be relocated there (Line 13). Line 14 then determines whether the placement cost of b on n is lower than the current minimum cost for b. If that is the case, then the algorithm updates the state to reflect that n is the nodelet that produces the minimum cost for b (Lines 15–17). Doing this allows for the optimization to discourage over-utilization of resources while still aiming to reduce memory access latency. This process is repeated for each nodelet, at which point the block placement map **P** is updated to map b to the nodelet that produces the lowest cost (Line 21).

Algorithm 1 Block Placement Optimization

INPUT: blocks, nodelets, cost model
OUTPUT: updated **P** map

```
1:  for each block b do
2:      compute PerformanceImpact_b
3:  end for
4:  pick candidate blocks with positive PerformanceImpact_b
5:  sort candidate blocks by decreasing PerformanceImpact_b
6:  for each candidate block b do
7:      MinLatency = L_b
8:      MinCost = PlacementCost_{b,p_b}
9:      MinNodelet = p_b
10:     for each nodelet n, n ≠ p_b do
11:         compute L_{b'} for b if it were on n
12:         if L_{b'} < MinLatency then
13:             compute PlacementCost_{b,n}
14:             if PlacementCost_{b,n} < MinCost then
15:                 MinLatency = L_{b'}
16:                 MinCost = PlacementCost_{b,n}
17:                 MinNodelet = n
18:             end if
19:         end if
20:     end for
21:     p_b = MinNodelet
22:     update cost model
23: end for
```

C. Cost Model Update

After each block placement, the cost model needs to be updated to reflect the changes induced by relocating the block (Line 22 in Algorithm 1). Because iteratively rerunning the program after each block placement to determine its new performance is prohibitively expensive, we instead infer the changes to the memory access profile using the metrics provided by the data-centric profiler and our understanding of the Emu architecture, as we now describe.

For a block b, its original nodelet n and each window w, threads that access b and do not make any other accesses on n

are subtracted from $T_{ndlt}(n, w)$. Similarly, for the new nodelet location n' and each window w, the threads that access b, and are not already active on n', are added to $T_{ndlt}(n', w)$.

By moving block b to a new nodelet, the number and source of the incoming memory accesses do not change. However, the latency cost of the accesses will change. This is reflected by updating the block latency map $\mathbf{L}$. Specifically, column b is recomputed to account for b's new location.

Relocating block b can change the source of incoming accesses for other blocks. Consider a scenario where some number of accesses were made to block b', which is located on a different nodelet from b itself. Furthermore, assume that those accesses were *outgoing* from b, which means they were made immediately after accessing block b. From the perspective of b', those accesses were incoming from b's original nodelet location. However, now that b has moved to another location, those accesses are now coming from b's new nodelet. If b were to be moved onto the same nodelet as b', then those accesses would be considered local, significantly changing the cost model. Fortunately, the data-centric profiler keeps track of these outgoing memory accesses for the blocks, which enables the cost model to be updated to account for such changes.

D. Complexity Analysis

Our analysis will focus on the loop on lines 6–23 in Algorithm 1, which dominates the complexity cost. The inner loop on lines 10–20 performs $\mathcal{O}(N)$ iterations and computing the latency of b on nodelet n (Line 11) requires $\mathcal{O}(N)$ operations. Determining the placement cost of block b on nodelet n (Line 13) is independent of the number of blocks and nodelets, so its cost is constant. Therefore, the entire inner loop requires $\mathcal{O}(N^2)$ operations. The outer loop on lines 6–23 performs $\mathcal{O}(B)$ iterations and updating the cost model on line 22 requires $\mathcal{O}(B + N)$ operations. Thus, the complexity of Algorithm 1 is $\mathcal{O}(BN^2)$.

Commonly, the number of blocks is the same as the number of nodelets. In that case, the complexity is simply $\mathcal{O}(N^3)$. This suggests that the algorithm may have scalability issues for a system with a large number of nodelets. However, the current Emu Chick system only has 32 nodelets. Addressing the scalability of the algorithm is left for future work.

VI. Case Study: Sparse Matrix Vector Multiply

To evaluate the cost model and block placement optimization, described in Sections IV and V, we chose sparse matrix-vector multiply (SpMV) as our case study. SpMV is a fundamental kernel in sparse linear algebra and is prevalent in graph analytics. Furthermore, our prior work investigated SpMV on Emu and demonstrated the importance of data layout choices for performance [10]. However, that work relied on existing matrix reordering techniques to control data layout, which differ from the application-independent framework presented in this work.

In the following sections, we briefly describe the implementation details of SpMV and which aspect of its operation we optimized. We then present performance results across a wide range of sparse matrices and discuss their implications.

A. Implementation Details

The SpMV implementation used in this study is taken directly from our prior work [10], where we formulate the operation as $\mathbf{Ax} = \mathbf{b}$. A sparse matrix $\mathbf{A}$ is stored in Compressed Sparse Row (CSR) format, where an equal number of contiguous rows are distributed to each nodelet and then further distributed to some number of threads that are spawned on each nodelet. The dense vectors $\mathbf{x}$ and $\mathbf{b}$ are divided into equal-sized blocks and distributed to the nodelets. A thread operates on the non-zeros within its assigned rows, potentially migrating to access elements of $\mathbf{x}$ that are not local. If a thread migrates to access $\mathbf{x}$, it will always migrate back to the "home" nodelet where its portion of the CSR structure is stored so it can process the next non-zero.

For the experiments in this work, we focus on optimizing the data layout for the input vector $\mathbf{x}$. For each experiment, we initialize the layout of $\mathbf{x}$ to the *default* layout, in which the blocks of $\mathbf{x}$ are distributed to the nodelets in a round-robin fashion, such that block i is stored on nodelet i, and each nodelet is given exactly one block.

B. Performance Evaluation

We evaluated 57 matrices from the SuiteSparse Matrix Collection [11]. The matrices are drawn from a wide range of domains, from graph analytics and optimization problems to computational fluid dynamics and circuit simulation. The number of non-zeros range from 200,000 to 37 million and the fraction of non-zeros varies from 0.5 to 1.81×10^{-7}.

The experiments were performed on all 8 nodes of the Emu Chick system using version 19.02 of the toolchain. For each experiment, the input vector $\mathbf{x}$ is partitioned into 32 blocks, where each of the 32 nodelets receives a single block as per the default layout. Each nodelet spawns 192 threads that operate on the rows assigned to it.

Of the 57 matrices, the cost model determined that 22 of them would benefit from a new data layout while the remaining 35 matrices were left unchanged. With respect to the 22 matrices that the cost model assigned new layouts, we observe runtime speed-ups, some as much as 50%, for 21 of the matrices. The remaining matrix suffers a small performance loss of 6%. Overall, the geometric mean of the speed-up of the 22 matrices is 12.5%. These results are presented in Figure 2.

C. Discussion

We hypothesize that poor data layouts were generated for the HEP-th-new matrix, as well as the matrices that achieved only minimal performance gains, due to thread migration hot spots. These matrices exhibit similar sparsity patterns, with dense columns of non-zeros that span most of the rows in the matrix. In particular, the execution time behavior of the cop20k_A matrix was evaluated in detail in our prior work [10], where we showed a migration queue became

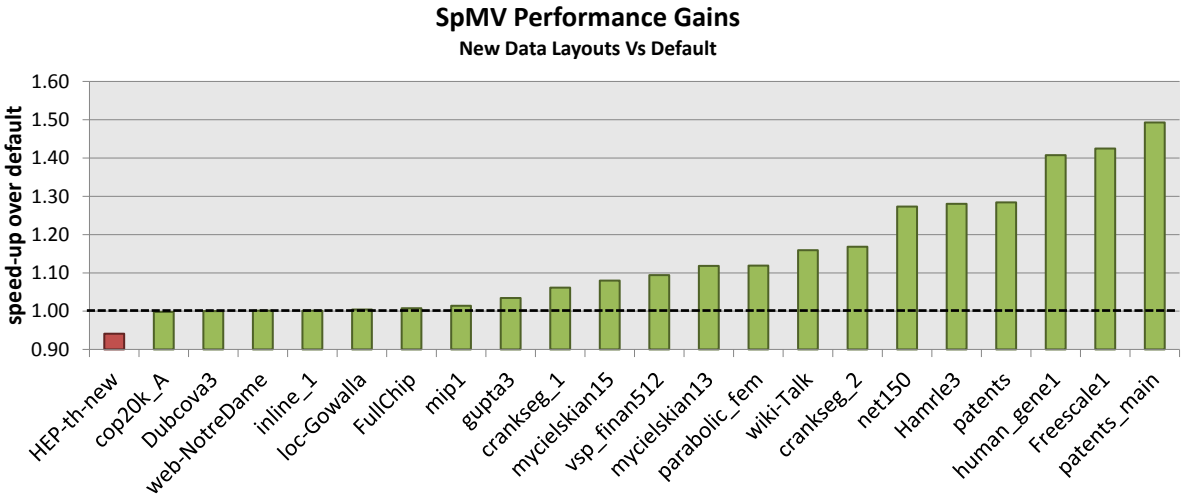


Fig. 2. Performance results on SpMV when using new data layouts. The vertical axis is runtime speed-up relative to the default layout for each respective matrix. The dotted horizontal line represents neutral performance. Green bars represent performance gain while red bars represent performance loss. The horizontal axis is sorted by increasing performance from left to right.

flooded with threads attempting to leave a nodelet at the same time. This happens because the first block in **x** is needed by almost every thread in the system at roughly the same time. Currently, our cost model does not fully capture the detailed behavior of the migration queue due to a lack of accurate information from the simulator.

We observe that the matrices that benefit the most from the data layout optimization are generally those whose non-zeros are not clustered around the main diagonal. As shown in our prior work, when the non-zeros are tightly clustered around the main diagonal, SpMV on the Emu architecture benefits greatly due to minimal migrations and good load balancing. Therefore, the data layout optimization is unlikely to find a higher performing layout given such a matrix.

As important as it is to achieve performance gains, it is equally important to avoid significant performance losses. Two algorithmic steps, down-selecting candidate blocks and sorting the blocks prior to the optimization, discourage selection of blocks that might reduce performance. To measure the impact of these steps, we ran the same set of experiments with those features removed. We found that by excluding sorting, the maximum performance loss for the same experiments is 23%. When the optimization does not select candidate blocks, but instead attempts to move all of the blocks, the maximum performance loss is 28%.

The targeted use-cases for the framework described in this work are those where the cost of profiling and optimizing data layouts can be amortized by executing many iterations of the application. For example, SpMV is the underlying kernel in conjugate gradient solvers [12], where only the values within the input vector change during each iteration. In such a case, the data layout would only need to be generated once.

VII. Related Work

The details of the Emu architecture, including hardware and software components, were presented by Dysart et al. [5]. In that work, benchmarks such as Breadth First Search (BFS), RandomAccess and SpMV were evaluated on the Emu simulator. It was not until the work of Hein et al. [8], [13] that the actual Emu Chick hardware was used in experiments, where they provided an initial characterization of the Emu Chick by evaluating several benchmarks and applications, including STREAM, pointer chasing, and BFS. Rolinger and Krieger [10] studied the impact of traditional optimizations for SpMV on the Emu Chick, showing that common techniques used to enforce hardware load balancing do not provide performance gains due to the migratory nature of Emu threads. Beyond direct application studies, Chatarasi and Sarkar [14] conducted a preliminary study of compiler transformations on the Emu system to reduce thread migrations, specifically targeting graph applications. Our work differs from previous work in that, rather than porting an application and evaluating its performance, we now focus on developing a framework to provide the means for application-independent data layout optimization.

Data partitioning is one of the fundamental techniques used in parallel and distributed computing and has been widely studied. Challenges for performance-driven optimization include gathering the appropriate metrics that can be used to guide optimization decisions, as well as determining the success of such decisions. Memory trace analysis can be very expensive, and simply re-running the entire application to evaluate an optimization is often not feasible. Rubin et al. [15] present a framework for finding high performing data layouts via profile-driven feedback on shared-memory cache-based systems. Their contribution was to simulate a candidate layout on representative memory traces rather than re-running the application. To address the problem of high overhead approaches to gathering memory access information, Yu et al. [16] proposed LWPTool, which uses light-weight address sampling and novel methods to determine memory access patterns to guide data layout optimization.

Our work differs from prior efforts by focusing on data layout optimization for irregular applications on a novel migratory thread architecture, which is fundamentally different from the architectures studied in prior works. The Emu system has no caches or special-purpose memories, and it supports thousands of independently migrating threads. Due to the migratory nature of the threads, efforts to statically distribute work evenly across the system are negated as soon as threads begin to migrate. Furthermore, as the system is relatively new and implemented on FPGAs, it lacks many of the low-level hardware features that would allow for the more sophisticated techniques used in the referenced prior work. Our work presents a preliminary cost model for understanding memory access performance on Emu as it relates to data layouts, as well as the necessary tools for obtaining the metrics used in the cost model.

VIII. Conclusions and Future Work

With the Emu migratory thread system, data layout choices play a crucial role in obtaining high performance. Optimizations targeting data layout for hierarchical memory systems are ineffective on Emu due to the fundamental differences in architecture. To address this gap, we have presented a preliminary framework for facilitating block-based data layout optimization on the Emu architecture.

To demonstrate the framework's utility, we have designed and implemented a straightforward block placement optimization. The optimization achieves speedups of up to 50% on the SpMV kernel, while never decreasing performance by more than 6%.

We are developing more data layout optimizations, such as adjusting block sizes and replicating blocks. We also believe that the cost model could be improved by incorporating more details of the underlying Emu architecture, such as the various queues. Furthermore, we will include memory usage statistics as part of the cost model, which will be crucial for the block replication optimization. As the Emu architecture matures, we plan to improve our tools by taking advantage of hardware performance counters and other features that would allow us to reduce the overhead of our framework, and perhaps move towards performing data layout optimizations at runtime.

References

[1] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," *Scientific Programming*, vol. 21, no. 3-4, pp. 137–148, 2013.

[2] J. Li, J. Sun, and R. Vuduc, "Hicoo: Hierarchical storage of sparse tensors," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 238–252.

[3] P. M. Kogge, "Execube-a new architecture for scaleable mpps," in *1994 International Conference on Parallel Processing Vol. 1*, vol. 1. IEEE, 1994, pp. 77–84.

[4] D. H. Yoon, M. K. Jeong, M. Sullivan, and M. Erez, "The dynamic granularity memory system," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 548–559. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337159.2337222

[5] T. Dysart, P. Kogge, M. Deneroff, E. Bovell, P. Briggs, J. Brockman, K. Jacobsen, Y. Juan, S. Kuntz, R. Lethin, J. McMahon, C. Pawar, M. Perrigo, S. Rucker, J. Ruttenberg, M. Ruttenberg, and S. Stein, "Highly scalable near memory processing with migrating threads on the Emu system architecture," in *Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms*. Piscataway, NJ, USA: IEEE Press, 2016, pp. 2–9. [Online]. Available: https://doi.org/10.1109/IA3.2016.7

[6] E. Petrank and D. Rawitz, "The hardness of cache conscious data placement," in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '02. New York, NY, USA: ACM, 2002, pp. 101–112. [Online]. Available: http://doi.acm.org/10.1145/503272.503283

[7] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of parallel and distributed computing*, vol. 37, no. 1, pp. 55–69, 1996.

[8] E. Hein, T. Conte, J. Young, S. Eswar, J. Li, P. Lavain, R. Vuduc, and J. Riedy, "An initial characterization of the Emu Chick," in *The 8th International Workshop on Accelerators and Hybrid Exascale Systems (AsHES)*, 2018.

[9] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global arrays: A nonuniform memory access programming model for high-performance computers," *The Journal of Supercomputing*, vol. 10, no. 2, pp. 169–189, 1996.

[10] T. B. Rolinger and C. D. Krieger, "Impact of traditional sparse optimizations on a migratory thread architecture," *2018 IEEE/ACM 8th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pp. 45–52, 2018.

[11] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011. [Online]. Available: http://doi.acm.org/10.1145/2049662.2049663

[12] J. Dongarra, M. A. Heroux, and P. Luszczek, "High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems," *The International Journal of High Performance Computing Applications*, vol. 30, no. 1, pp. 3–10, 2016.

[13] E. R. Hein, S. Eswar, A. Yasar, J. Li, J. S. Young, T. M. Conte, Ü. V. Çatalyürek, R. Vuduc, E. J. Riedy, and B. Uçar, "Programming strategies for irregular algorithms on the emu chick," *CoRR*, vol. abs/1901.02775, 2019. [Online]. Available: http://arxiv.org/abs/1901.02775

[14] P. Chatarasi and V. Sarkar, "A preliminary study of compiler transformations for graph applications on the Emu system," in *Proceedings of the Workshop on Memory Centric High Performance Computing*, ser. MCHPC'18. New York, NY, USA: ACM, 2018, pp. 37–44. [Online]. Available: http://doi.acm.org/10.1145/3286475.3286481

[15] S. Rubin, R. Bodík, and T. Chilimbi, "An efficient profile-analysis framework for data-layout optimizations," *SIGPLAN Not.*, vol. 37, no. 1, pp. 140–153, Jan. 2002. [Online]. Available: http://doi.acm.org/10.1145/565816.503287

[16] C. Yu, P. Roy, Y. Bai, H. Yang, and X. Liu, "Lwptool: A lightweight profiler to guide data layout optimization," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 11, pp. 2489–2502, Nov 2018.

Optimizing Post-Copy Live Migration with System-Level Checkpoint Using Fabric-Attached Memory

Chih Chieh Chou*[1], Yuan Chen†[2], Dejan Milojicic‡, A. L. Narasimha Reddy*, and Paul V. Gratz*
*Department of Electrical and Computer Engineering
Texas A&M University
Email: {ccchou2003, reddy, pgratz}@tamu.edu
†JD.com Silicon Valley R&D Center
Email: yuan.chen@jd.com
‡Hewlett Packard Labs
Email: dejan.milojicic@hpe.com

***Abstract*—Emerging Non-Volatile Memories have byte-addressability and low latency, close to the latency of main memory, together with the non-volatility of storage devices. Similarly, recently emerging interconnect fabrics, such as Gen-Z, provide high bandwidth, together with exceptionally low latency. These concurrently emerging technologies are making possible new system architectures in the data centers including systems with Fabric-Attached Memories (FAMs). FAMs can serve to create scalable, high-bandwidth, distributed, shared, byte-addressable, and non-volatile memory pools at a rack scale, opening up new usage models and opportunities.**

Based on these attractive properties, in this paper we propose FAM-aware, checkpoint-based, post-copy live migration mechanism to improve the performance of migration. We have implemented our prototype with a Linux open source checkpoint tool, CRIU (Checkpoint/Restore In Userspace). According to our evaluation results, compared to the existing solution, our FAM-aware post-copy can improve at least 15% the total migration time, at least 33% the busy time, and can let the migrated application perform at least 12% better during migration.

I. Introduction

The emerging Non-Volatile Memories (NVM), such as phase-change memory (PCM) [1], NVDIMM [2], and 3D XPoint [3], have byte-addressability and low latency, close to that of main memory, together with the non-volatility of storage devices. Similarly, recently emerging interconnect fabrics, such as Gen-Z [4], provide high bandwidth, together with exceptionally low latency. These concurrently emerging technologies are making possible new system architectures in the data centers including systems with Fabric-Attached Memories (FAMs) [5]. FAMs can serve to create scalable, high-bandwidth, distributed, shared, and byte-addressable NVM pools at a rack scale, opening up new usage models and opportunities. These technologies have great potential to improve the system/application performance as well as to provide scalability and reliability of applications/service. Many prior work focused on new system designs using NVM [6]–[11] and has already shown promising performance improvements.

Migration is a crucial technique for load balancing. Migration allows system administrators to remove some applications from stressed physical nodes in order to redistribute load, and therefore to increase overall system performance. In addition, migration can also provide power saving [12], and online maintenance. Traditional approaches to migration are non-live; that is, they require the application to be taken off-line while the migration occurs. Non-live migration can be divided into three steps: 1) the program is checkpointed at source, 2) the checkpointed data are copied from source to target, and 3) the program is restarted at target. The main drawback of non-live migration is that its application downtime (off-line time) is too long. To reduce this downtime, live migrations [13] are proposed to migrate most of (or all) pages before (pre-copy) or after (post-copy) "real" migration, and therefore to reduce the number of migrated pages during the downtime. However, the side effect of live migrations is that they require longer, compared to non-live migration, busy time of source node.

Here, we define the busy time as the duration from the beginning of the migration to the time that migrated applications can be killed at source node.[1] As far as we know, all prior works of post-copy focused on total migration time. However, we would like to make another point here that the busy time might be more important because it has direct impacts on the system performance; it decides when computing resources, such as CPU and memory, occupied by migrated applications to be released and be reallocated to remaining applications in source node. For example, when applications are suffering from swapping due to the lack of memory in a "hot" node, simply migrating an application with large memory footprint might be able to stop (after busy time) all remaining applications at source from swapping and therefore to improve the

[1] The author started this work as an intern at Hewlett Packard Labs.
[2] The author contributed while he was at Hewlett Packard Labs.

[1]Note: after migration completes, the migrated applications would continue to execute at target node.

overall system performance.

In this work, we consider that migration can greatly benefit from FAM. Although the non-live migration techniques can be easily improved with FAM by the removal of the copy phase (step 2), the optimal post-copy page fault handler, however, requires significant redesign. This new handler should rely on the both non-volatility and shareability of FAM to provide the optimal (shortest) busy time as well as total migration time. In particular, we introduce FAM-aware, checkpoint-based, post-copy live migration, which checkpoints the entire application to FAM in the beginning and transfers all pages through FAM, rather than through the network connection, which the traditional migration techniques use.

The contributions of this paper are as follows:

- Propose a new, checkpoint-based, page fault handler for post-copy migration using FAM to provide the best busy time and better total migration time.
- Implement our FAM-aware post-copy migration on a Linux open source checkpointing tool, CRIU (Checkpoint/Restore in Userspace).
- Evaluate our enhancements of CRIU with synthetic and realistic (YCSB+REDIS) workloads and show significant performance improvement.

The remainder of this paper is organized as follows. Section II describes the background and related work. Section III presents the motivation and design overview of our FAM-aware post-copy live migration. Section IV explains the implementation of our work by modifying and enhancing existing CRIU in more detail. Section V presents our results of evaluation of post-copy live migration using some macro benchmarks and realistic workloads. Finally, section VI concludes.

II. Background

A. Migration

Traditionally, in non-live migration of applications, at first the migration program needs to stop the applications. It then checkpoints their state (mostly as files) into local storage devices, copies these checkpointed data to the remote storage devices (at the target machine), and finally resumes them back to the checkpointed state at target. The downtime is mostly proportional to the amount of migrated memory.

Unlike non-live migration, live migration means that application is (or appears to be) responsive during the entire migration process. Post-copy [14], [15] transfers the processor state, register, etc., to target and resumes immediately at the target host. When resumed application accesses some pages which have not yet been transferred, page faults are triggered, and those pages are retrieved from source node through network.

Although post-copy has the almost minimum downtime, it would suffer from the network page fault overhead, and therefore degrade the migrated application performance during migration. Also, post-copy usually requires the longer migration time (which depends on page access pattern of application), compared with non-live one.

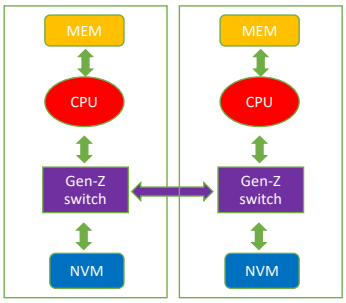


Fig. 1: Fabric-Attached Memory.

Sahni et al. [16] proposed a hybrid approach combining pre-copy and post-copy to take advantage of both types of live migration. The post-copy could suffer less page faults if the pages of the working set have already been sent by the pre-copy phase. Ye et al. [17] investigated the migration efficiency under different resource reservation strategies, such as CPU and memory reserved at target or source nodes. CQNCR [18] considered an optimal migration plan to migrate massive virtual machines in the data center by deciding a migration order to have less migration time and system impact.

These works, however, largely employ traditional networking as the only transfer media, thus they incur high access latencies. Besides, they only emphasize the total migration time, not busy time.

B. Fabric-Attached Memory

Fabric-Attached Memory (FAM) is a system architecture component in which high performance networking fabrics are used to interconnect NVM between multiple nodes in a rack to create a global, shared NVM pool. In our system model, we consider a cluster consisting of many nodes, each of which contains both DRAM and NVM. DRAM can only be accessed locally by local memory controller and serves as fast, local memory in each node. NVM and the processor (in each node) are connected to a switch fabric and these switches are interconnected to each other. Here we focus on Gen-Z [4] as one such fabric because it supports hundreds gigabyte per second bandwidth and memory semantics; however, any other fabric of similar latency and bandwidth could be used. The CPU can access NVM at other nodes with memory-semantics through the fabric interface and libraries. Therefore, all interconnected NVM can be treated as the slow, global memory pool in a rack scale [5]. This byte-addressable, shared, high-bandwidth, global NVM pool is so-called Fabric-Attached Memory (FAM) in this work. Fig. 1 shows the overall memory, NVM, CPU, and Gen-Z interconnections.

C. Checkpoint/Restore in Userspace (CRIU)

The Checkpoint/Restore in Userspace (CRIU) [19] is a Linux open source checkpoint tool which saves the current state of a running application into the local storage devices and restarts the application whenever necessary. To checkpoint, CRIU uses `ptrace` system call to inject a piece of parasite

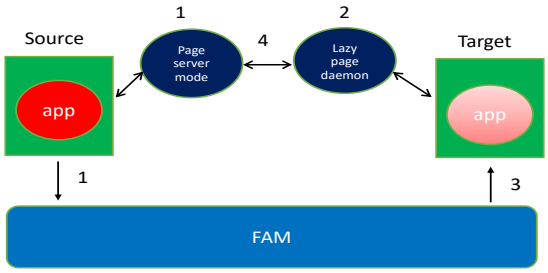


Fig. 2: Existing CRIU post-copy using FAM. (1) CRIU checkpoints all files except for page image file to FAM, and transforms itself as a page-server. (2) At target, CRIU creates a lazy-page daemon. (3) After (1) is completed, CRIU restores application immediately at target. (4) If restored application at target accesses a page and causes a page fault, it would notify the lazy-page daemon, which then requests to and obtains from page-server at source that faulting page.

code into the checkpointed application. Through the injected parasite, CRIU daemonizes the checkpointed application and lets it begin to accept commands sent by CRIU. Hereafter, CRIU starts the checkpoint process.

The most time-consuming part of checkpoint process is to dump pages of application. The page-dumping request asks the application to execute `vmsplice` system call, which maps the pages of VMAs of the process into pipes (kernel buffers). Finally, after all pages have already been mapped to the pipes, CRIU can access them directly (without the help of parasite) through `splice` system call, which copies dumped pages from pipes to a page image file in the storage devices.

CRIU also supports migration features, including non-live, pre-copy, and post-copy live migrations. For post-copy, the page fault handling is the main challenge. Like on-demand paging used in virtual memory systems, CRIU's post-copy employs the `userfaultfd` system call to allow paging in the user space. Fig. 2 shows the sequential steps of the existing CRIU post-copy implementation. To achieve the post-copy, like checkpoint, at first CRIU maps the pages of VMAs of applications, at the source node, into pipes and then places itself into a page-server mode. Then a lazy-page daemon at the target node is created to handle the page fault and other events requested during the following restore operations. Finally, the migrated application is resumed at the target node. When the restored application accesses a page which still remains in the source, the application is halted temporarily, and sends the page fault request to the lazy-page daemon, which in turn communicates with page-server at the source node to obtain that faulting page from pipes through network transfer. Although all other checkpointed state can be sent through FAM. The page transfer still needs to rely on socket interface if page fault handling is not redesigned.

We note that when using pure on-demand paging, migration process may never complete, since some pages may not be ever accessed. CRIU thus employs a timer to trigger the active pushing; that is, sequentially dumping all remaining pages from source to target. The timer is kept reset whenever a page fault event happens. Before the timer is expired, the lazy-page daemon only handles the page fault event, and it would start to actively push all remaining pages only after expiration.

III. Motivation and System Design Overview

In this section, we describe the motivation and design overview of our FAM-aware post-copy live migration.

A. Fabric-Attached Memory-Aware Post-Copy Live Migration

We propose an optimized FAM-aware post-copy migration, which exploits the properties of FAM to achieve low application downtime, low total migration time, low application degradation, low resume time, and especially low busy time.

To simplify the understanding of readers, we first briefly restate some key metrics of live migration proposed by Hines et al. [15] and we further introduce the concept of busy time.

- **Downtime:** The time that the application is stopped and cannot respond while the state of the processors and some pages are transferred. Post-copy, depending on implementation, might transfer a few (or no) pages. Non-live migration transfers all pages here, so its downtime is the longest.
- **Resume Time:** The time from the application starts executing to the end the entire migration process. This time is mainly required for post-copy to handle the page faults happening at the target node, and is near negligible to the non-live migration.
- **Busy Time:** The time from the beginning of the migration to the time that migrated applications can be killed and their resources (especially CPU and memory) can be released at source node. This may be the most important metric for migration since system administrators can only alleviate the loading of "hot" nodes after this time.
- **Migration Time:** The total time of downtime and resume time. Usually the migration time (of live migration) equals to the busy time; however, this is not the case if we employ FAM for migration. We will discuss this later.
- **Application Degradation:** The extent that application is slowed down during the operations of the migration. The application degradation of post-copy, because it handles the page fault at target and needs to get the faulting pages from source, might be the most severe.

B. Motivation and Design

Intuitively, the performance of non-live migration, especially migration time, could benefit from adopting the FAM simply because the copy phase can be eliminated through direct FAM accesses. Therefore, the entire migration process is now simplified as (1) checkpoint application to FAM at source and (2) restart application at target.

Post-copy migration is much more complicated than non-live one because the most critical part of post-copy is page fault handling (or network fault as termed by Hines et al. [15]) in the target. The behavior and implementation of the page

fault handler will greatly impact the resume time (application performance degradation) and busy time. Simply removing the copy phase, if applicable, is obviously not good enough. Therefore, our work focuses on optimizing FAM-aware post-copy migration based on the following three guidelines.

Checkpoint-Based Migration: The most apparent drawback of non-live migration is its very long downtime (which equals to the total migration time). However, the busy time of non-live migration is the best (compared to the live migrations) and is also much shorter than its total migration time because non-live migration first checkpoints everything to storage devices, and therefore, since a complete snapshot is saved in persistent media, the checkpointed application can be killed at the source.

Traditional live migrations have a long busy time (which equals to the total migration time) because they utilize memory to temporarily store pages and transfer pages by network. So, killing the application must wait until the completion of entire migration. Otherwise, if target crashes before migration completes, then the application will crash, too.

Due to the low-latency and non-volatility of FAM, storing migrated pages to FAM directly only slightly impacts performance (compared to DRAM), but it also saves the entire migrated state to persistent media. Therefore, application can be killed after being checkpointed and its busy time could be very close to that of non-live migration.

Accessing FAM as Shared Memory: Most existing live migration techniques [16]–[18], [20] still migrate their data content through communication network because they do not have a shared memory across multiple nodes. Therefore, their approaches will suffer the network overhead and network bandwidth. On the other hand, with the help of FAM, serving as a shared memory pool within the same rack, data could be simply migrated through memory semantics (`load/store` instructions), which bypass the significant networking protocol overhead. This could improve the critical page fault latency and therefore resume time and application degradation.

Retrieving Faulting Pages Synchronously: The page fault handler of the existing shared memory within a machine is controlled by the central operating system. The OS only needs to setup the page table of each process, then faulting pages can be mapped to virtual space of processes correctly. Our migration scheme, however, differs because a central controlling OS across multiple hosts does not exist. The FAM across machines can only be employed as a connecting media between nodes.

Besides, our page fault handler must contain two steps: first pages are written from source to FAM and then pages are read from FAM to target. So, page fault handling must be executed asynchronously through communication between the source (writing to FAM) and target nodes (reading from FAM); that is, the faulting address of pages must be sent to source first and then target must wait for the response to read that page. This communication impacts the page fault latency (even though all pages are already transferred by FAM) as well as migration performance, and must be avoided as much as possible by leveraging the information of the dumped pages. The source could notify target of the information of all dumped pages, and therefore the following faulting pages (if they have been dumped to FAM) can be accessed synchronously at target from FAM without the need of communication.

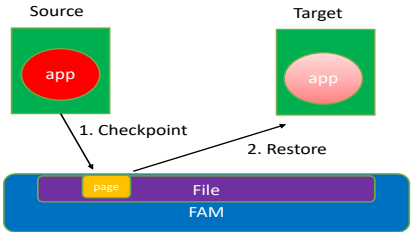


Fig. 3: Ideal migration method using FAM.

Thus, our post-copy optimization is mainly based on non-volatility and shared-ability of FAM, and can be divided into three parts.

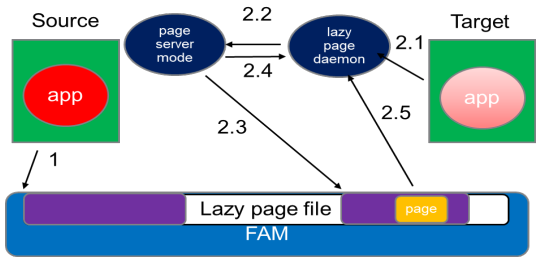
- *to achieve the shortest downtime*, we checkpoint the processor state, registers, etc., (excluding pages of VMA) of the victim application to FAM and resume victim application at the target.
- *to achieve low busy time*, at source, after checkpointing the necessary information, all the remaining pages continue to be dumped to FAM on the background. The victim application can be killed right after the page dumping is finished. This provides near optimal busy time, which is the checkpoint time of non-live migration.
- *to achieve low resume time and low application degradation*, all faulting pages at target will be served/received from FAM directly and the need of asynchronous communication is also tried to minimize. Therefore, with of help of FAM, the networking overhead as well as the page fault latency can be reduced significantly.

IV. Implementation

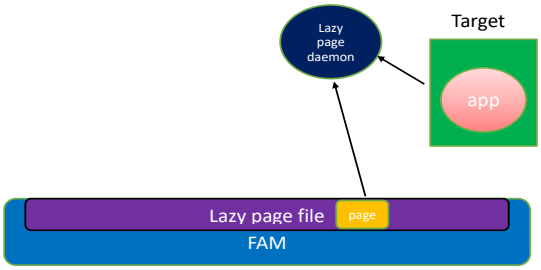
In this section, we explain our implementation of FAM-aware, checkpoint-based, post-copy live migration in detail.

The existing CRIU [19] migration tool is used as a baseline implementation, and augmented with our FAM-aware post-copy technique in this work. In particular, our case study implementation is based on modifying CRIU-dev branch version 3.2. Although our implementation is based upon CRIU, the technique developed is universal and may be easily implemented in other migration schemes.

Fig. 3 shows an ideal migration between nodes with FAM. First, an empty file is created in FAM, and the migration program at target can `mmap` this whole file and directly read dumped pages without having to wait until the whole file is dumped from the source. Ideally, (if the future can be predicted,) the source node would write a certain page to FAM each time before the page is needed at target. From the perspective of migration, which means that not only the restored application does not have to wait for the completion of page-dumping (that is, live migration), but it also avoids



(a) 1. Background thread dumps all lazy pages to FAM. 2.1. A page fault happens. A page fault event is sent to lazy-page daemon by kernel. 2.2. Lazy-page daemon requests page-server for that faulting page. 2.3. Page-server writes the entire vector, rather than a single faulting page, to FAM. 2.4. Page-server notifies the lazy-page daemon of the completion of dumping. 2.5. Lazy-page daemon directly reads faulting page from FAM.



(b) The page-server and application at source are killed after all lazy pages have been dumped. All remaining pages can be directly/synchronously accessed from target.

Fig. 4: FAM-aware post-copy live migration.

much of the network transmission (required by the existing CRIU and other previous implementations) through memory-semantics.

We have implemented the concept shown in Fig. 3 in our optimized post-copy migration in CRIU. Fig. 4 illustrates the main difference between our post-copy design versus the existing CRIU implementation (in Fig. 2).

Like the existing CRIU post-copy, we also employ a lazy-page daemon and page-server mode in our implementation. To migrate, first all required data are checkpointed as files to FAM except for the page image file, as the existing CRIU does. After that, CRIU at source enters a page-server mode, and launches a background thread to dump (checkpoint) all remaining "lazy" pages to FAM as a single lazy-page file (per process) (arrow with number 1 in Fig. 4(a)). Without having to wait for this background thread to finish its dumping job, system administrator can launch a lazy-page daemon and then can begin to restore the migrated application at the target node. To improve the checkpoint performance by batching and to avoid too long critical latency of page fault, we partition the VMAs of application as several I/O vectors with the maximum size of 2MB. Each I/O vector contains pages with contiguous virtual addresses. Therefore, the background thread writes the pages to FAM with at most 2MB of data at a time. When the restored application encounters a page fault, it sends the page fault event to the lazy-page daemon, which in turn sends page fault request to page-server. If the faulting page has not been dumped to FAM, the page-server waits for background thread to finish the dumping of current I/O vector, stops the background thread (by a spin lock), and dumps a specific I/O vector containing the requested faulting page. After dumping that (2MB) I/O vector, the page-server acknowledges page fault request and resumes the background thread. Arrows with number 2.1 to 2.4 in Fig. 4(a) illustrate this process. Alternately, if the faulting page has been dumped, the page-server acknowledges immediately.

The responses (arrow with number 2.4 in Fig. 4(a)) sent by the page-server not only indicate the "completion of dumping" of faulting pages, but they also contain some extra information for lazy-page daemon: the background thread's dumping progress (the largest virtual address of dumped pages) and the virtual address space of this entire (2MB) dumped I/O vector. The lazy-page daemon employs such information to construct a lookup table. (Actually, we build an LRU linked list). If the following faulting pages whose addresses can be found at the lookup table or are smaller than the dumping progress, they can be read from FAM synchronously without requesting to page-server. This can eliminate a lot of communications and overheads between source and target.

Once the background thread dumps all the lazy pages, the page-server actively notifies the lazy-page daemon of the completion of the checkpoint. Hereafter, the lazy-page daemon can synchronously read all faulting pages from FAM without any communication with page-server. Both migrated application and page-server can be killed at the source now as Fig. 4(b) shows.

Our implementation has some advantages: (1) The page-server only needs to handle faulting pages until the checkpoint of all "lazy" pages is finished. After that, all pages can be synchronously read from FAM. A lot of network transmissions of pages and protocol overhead can be eliminated. (2) The information of dumped pages is utilized to further reduce the communication between target and source nodes. It significantly minimizes the page fault latency and therefore improves performance. (3) After the background thread finishes dumping the lazy files, the application and page-server can be killed and their resources can be released; that is, the busy time would be much shorter than total migration time.

V. Evaluation

In this section, we experimentally evaluate the performance improvement of our optimized FAM-aware, checkpoint-based, post-copy live migration. Our platform contains 20GB DDR3-1600, 12GB NVM (emulated with DRAM [21]), and Intel i7-4770 four-core 3.4 GHz processor with hyperthreading enabled. Linux 4.15.0 is used in our platform.

A. Workloads and Experimental Setup

To examine the performance of our FAM-aware post-copy migration scheme, we leverage the NAS Parallel benchmarks (NPB) 3.3.1 [22] Class C, PARSEC 3.0 [23] native input, and

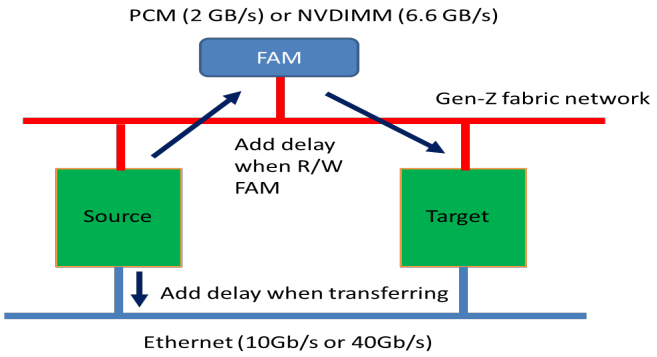


Fig. 5: The delay model of our evaluation.

TABLE I: Parameters for extra delay.

Media	Read lat. (us)	Write lat. (us)	BW. (GB/s)
PCM	1	1	2
10Gb Ethernet	0	0	1.25

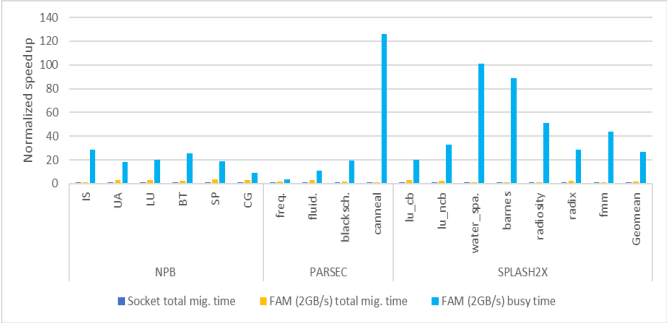
SPLASH-2X [24] native input benchmark suites. All workloads are migrated after executing twenty seconds, with the exception of "NPB IS" which migrated after five seconds for one thread and two seconds for four threads, and "SPLASH-2X radix" is migrated after five seconds for four threads.

We further examine the migration of REDIS [25] to investigate application performance degradation during live migration. REDIS is an in-memory data structure store and can be employed as database or in-memory cache. Through loading YCSB (Yahoo! Cloud Serving Benchmark) [26] records into REDIS, migrating REDIS to the target, and immediately accessing REDIS with YCSB at the target before the migration is finished, we can observe the impact of page fault overhead on REDIS performance with different YCSB workloads.

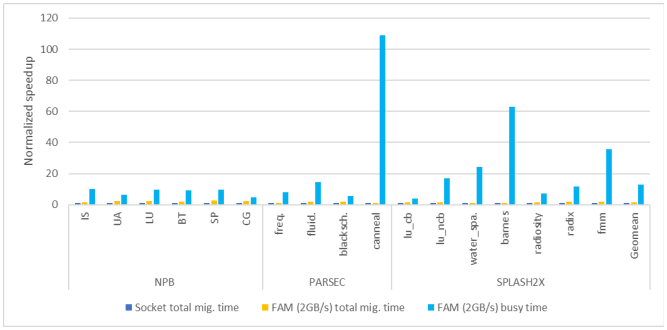
To evaluate FAM-aware migration, two limitations must be overcome. First, CRIU requires migrated applications to use the original pid they were checkpointed with. This means application cannot be "lively" migrated within the same physical host. Next, we do not have real FAM hardware, so we do not have a global, shared NVM across physical nodes. These two limitations seem to contradict with each other. Fortunately, with the help of a container virtualization technique, FAM can be emulated within a host machine by means of container and NVM: Two docker containers [27] are treated as target and source nodes; the emulated NVM, which is bind-mounted into containers so applications in containers can access NVM concurrently, can be emulated as FAM in our platform.

To compare the performance of our FAM-aware with existing CRIU post-copy, we assume all process states, except for memory pages, are migrated through FAM, so post-copy migration contains only 2 steps: checkpoint from source container and restart at target container. However, the methods of page transfer are different: one is by FAM, the other is through socket interface.

Furthermore, since NVM (FAM) is emulated from DRAM,



(a) One thread per benchmark. (Higher is better.)



(b) Four threads per benchmark. (Higher is better.)

Fig. 6: The busy time and total migration time performance of FAM-aware and existing CRIU post-copy migration using FAM. Expiration time is 100ms.

to correctly emulate the "slow" NVM, we use the same method proposed by Volos et al. [6] to add extra delay (via busy waiting) when accessing slow FAM (or Ethernet). Fig. 5 illustrates our delay model. The delay is added both when writing to and reading from FAM. The latency resulted from Gen-Z fabric is negligible compared to that of PCM [28]. Therefore, we only consider the access latency and bandwidth of NVM (PCM or NVDIMM). The delay calculation formula is as follows:

$$Delay_{R/W} = Latency_{R/W} + (data/bandwidth)$$

The parameters of NVM (and 10 Gigabit Ethernet used later) are shown at Table I. The bandwidth of NVM (PCM) is assumed to be 2GB/s [29]. Therefore, to access a single 4KB page from FAM, we have to wait around 3us and then can access FAM.

B. Evaluation Results

Fig. 6 (a) and (b) show the normalized total migration time and busy time performance of our FAM-aware vs. existing[2] CRIU post-copy with one and four threads. All the measurements are normalized to the total migration time of the existing post-copy (socket) of the same benchmark. FAM and socket (in Fig. 6) stand for the mechanism of page transfer by

[2]Note: for existing CRIU post-copy, the busy time equals to the migration time, so only total migration times are shown.

TABLE II: Average improvements of FAM-aware post-copy vs. existing CRIU post-copy.

# of threads	Mig. time imprvmt	Busy time imprvmt
1	2.00X	26.74X
4	1.63X	12.72X

FAM and by socket (network). We will keep using these terms hereafter. FAM (2GB/s) means the extra delay is added based on the 2GB/s bandwidth of PCM. The expiration time of active pushing timer is set as 100ms. Remember this timer would be reset whenever a page fault happens. We think 100ms should be a reasonable duration for workloads to access all working set pages before timer expires. So, the total migration time here should be a good indicator toward the different page fault overhead of these two mechanisms.

Tab. II summarizes the performance improvements of our FAM-aware post-copy. The number is the geometric mean of all benchmarks. From those results, we could conclude some useful observations.

First, instead of workload behavior, the busy time (also checkpoint time) of post-copy is impacted mostly by the dumped memory size. Thus, they are relatively small compared to total migration time.

Second, the improvements of total migration time (2X and 1.63X) come from faster demand paging handling, proving that our FAM-aware migration incurs less page fault overhead.

Third, the improvements of both total migration time and busy time reduce as the number of threads increases. For migration time, that is mainly because of the available bandwidth. We limit the FAM bandwidth as 2MB/s (PCM). On the other hand, although the existing CRIU acquires pages from source through socket, we do not apply any bandwidth limitation on it. Docker containers, in a single host, utilize Linux bridge component and bypass the NIC (network interface card) to communicate with each other. To estimate the bandwidth between docker containers, we use iperf3 [30] tool to measure and get the average 7.0 GB/s throughput (compared to 2GB/s at FAM-aware case). So, we could conclude that the reduction of improvements of total migration time when more threads are executed comes from the bandwidth limitations of FAM.

For busy time, the busy time of FAM-aware post-copy is the checkpoint time, which is almost the same if the memory footprint does not change, regardless of the number of executing threads. As to the existing post-copy, since the busy time is the total migration time, which would be improved as the number of threads increases because more threads will access pages more quickly. Therefore, the decreasing the busy time improvement results from the total migration time improvement of existing post-copy as more threads are executed.

Fig. 7 shows a similar comparison of FAM-aware and existing CRIU post-copy except that the expiration time is set as 0ms. Only the workloads whose migrated memory sizes are larger than 1GB are selected. All workloads are executed

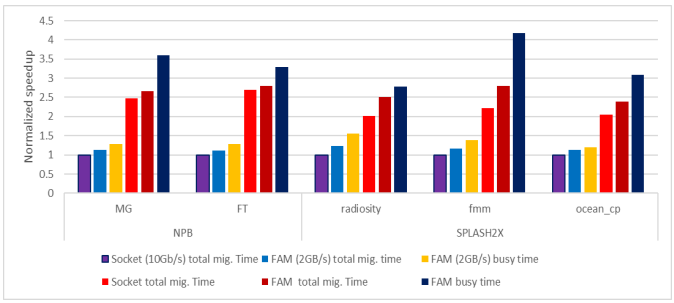


Fig. 7: The comparison of busy time and total migration time performance with the expiration time is 0ms. (Higher is better.)

TABLE III: Average improvements of FAM-aware post-copy for realistic case (FAM (2GB/s) v.s. socket (10Gb/s)) and ideal case (FAM v.s. socket).

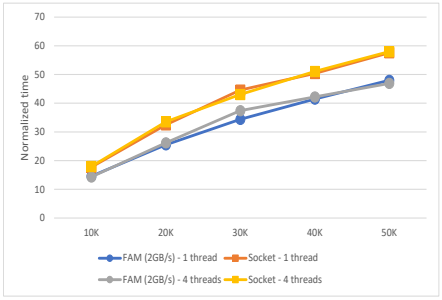
	Mig. time	Busy time
FAM (2GB/s) v.s. socket (10Gb/s)	15.44%	33.68%
FAM v.s. socket	15.16%	47.08%

with one thread and are migrated after executing twenty seconds. All the measurements are normalized to the total migration time of socket (10Gb/s) of the same benchmarks. Socket (10Gb/s) is assumed that the employed underlying network is 10 Gigabit Ethernet. FAM (without bandwidth limitation) means no delay is added when accessing NVM, which can be treated as the case of NVDIMM and whose performance is also the best among all cases. In our platform, the average throughput of accessing DRAM via file system `write` is around 6.6 GB/s (a little lower than 7 GB/s of socket throughput between containers).

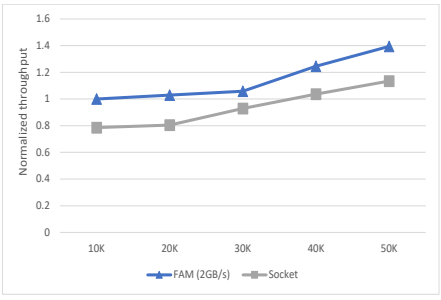
The 0ms expiration time means that the lazy-page daemon would actively push the pages from source from the beginning. Meanwhile, if a page fault happens, the daemon will also try to handle page fault event at best effort. From Fig. 7, we could conclude that (A) the bandwidth provided by the transmission media dominates the migration performance; (B) if the bandwidth is close to each other, FAM-aware is better than existing post-copy due to the lighter overhead. Tab. III summarizes the improvements.

Finally, REDIS and YCSB are utilized to investigate the migration performance, especially for performance degradation. 500K records are loaded by YCSB into REDIS at the source node, and each record is of 100 fields and the size of each field is fixed to 10B. This configuration will result in 939MB of pages to be migrated. After downtime, YCSB accesses REDIS at target immediately by different operation records (from 10K to 50K). The YCSB workloads are all configured as uniform distribution, readallfields and writeallfields are false, and R/W ratio are 70/30. The YCSB employs one and four threads to access REDIS. Fig. 8 shows the results. (a) to (c) employ 150ms expiration time and (d) to (f) employ 3ms.

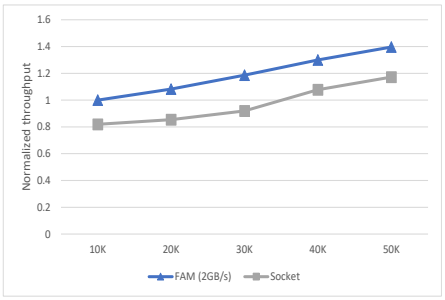
Fig. 8 (a) shows the measured total migration times normalized to the busy time of FAM (2GB/s). The busy times of one



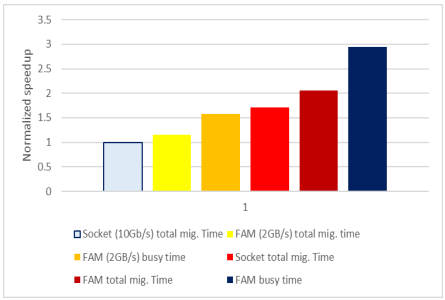
(a) Normalized total migration time. (Lower is better.)



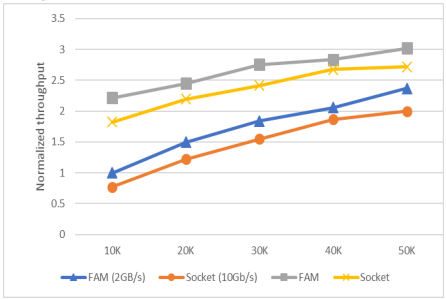
(b) Normalized throughput with one thread. (Higher is better.)



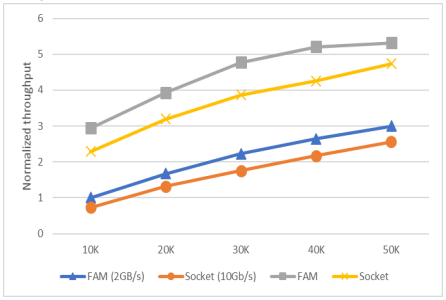
(c) Normalized throughput with four threads. (Higher is better.)



(d) Normalized total migration time and busy time performance. (Higher is better.)



(e) Normalized throughput with one thread. (Higher is better.)



(f) Normalized throughput with four threads. (Higher is better.)

Fig. 8: Migration performance of REDIS accessed by YCSB. (a) to (c): The expiration time is 150ms; (d) to (f): The expiration time is 3ms.

TABLE IV: Average REDIS throughput improvements of FAM-aware post-copy of real case (FAM (2GB/s) v.s. socket (10Gb/s)) and ideal case (FAM v.s. socket).

	1 thread	4 threads
FAM (2GB/s) v.s. socket (10Gb/s)	19.8%	25.69%
FAM v.s. socket	12.7%	21.79%

and four threads are almost the same since the same amount of data (939MB) are checkpointed. FAM-aware post-copy improves average total migration time 23.92% and 22.48% with one and four threads respectively. Fig. 8 (a) also indicates that the total migration time is not related to the number of threads of YCSB. The reason is that the REDIS is single-threaded, so more requests (threads) from YCSB cannot make the pages of REDIS be accessed faster. Fig. 8 (b) and (c) show the REDIS performance during migration. FAM-aware post-copy lets the REDIS perform 22.3% and 23.4% higher with one and four threads. respectively. Fig. 8 (a), (b), and (c) could prove that the total migration time and application degradation have some correlations because they are impacted mostly by the latency of page fault handling if the expiration time is larger enough.

Fig. 8 (d) shows the normalized total migration time and busy time performance with 3ms expiration time. Because those times at different operation counts of YCSB are almost the same, we only take the average. This result also looks like Fig. 7. Tab. IV summarizes the results of Fig. 8 (e) and (f).

Again, Fig. 8 (d), (e), and (f) also show that the total migration time and performance degradation are both influenced by the bandwidth of transmission media if the expiration time is too small and active pushing is triggered soon enough. The throughput of REDIS increases as the number of operations increases; this is because the chances of page fault reduce as more pages are migrated.

VI. Conclusion

We presented FAM-aware, checkpoint-based, post-copy migration. Through FAM, a global, shared NVM pool in a rack scale, we can map the entire migrated memory space onto FAM. So, the data migration can be simplified as memory-semantics to achieve a much lower page fault latency path. We have implemented our prototype at CRIU, and shown that our approach has lower busy time (at least 33%), lower total migration time (at least 15%), and migrated application can perform at least 12% better (i.e. lower application degradation) during migration process.

Acknowledgments

We first would like to thank Dr. Jaemin Jung, who was a postdoc at Texas A&M University, for his helpful comments and suggestions about this paper. We also thank the anonymous reviewers for their valuable and useful comments and feedback to improve the content and quality of this paper. Finally, we want to thank the National Science Foundation, which supports this work through grants I/UCRC-1439722 and

FoMR-1823403, and generous support from Hewlett Packard Enterprise.

REFERENCES

[1] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-change technology and the future of main memory," *IEEE Micro*, vol. 30, pp. 131–141, Mar. 2010.

[2] D. Narayanan and O. Hodson, "Whole-system persistence," in *ASPLOS '12*. London, England, UK: ACM, Mar. 2012, pp. 401–410.

[3] "Intel optane technology," https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

[4] Gen-Z speciticaions. https://genzconsortium.org/specifications/.

[5] K. Keeton, "Memory-driven computing," in *FAST '17*. Santa Clara, CA: USENIX, 2017.

[6] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *ASPLOS '11*. Newport Beach, CA: ACM, Mar. 2011, pp. 91–104.

[7] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," in *ASPLOS '11*. Newport Beach, CA: ACM, Mar. 2011, pp. 105 – 118.

[8] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "Nv-tree: Reducing consistency cost for nvm-based single level systems," in *FAST '15*. Santa Clara, CA: USENIX, Feb. 2015, pp. 167 – 181.

[9] V. Fedorov, J. Kim, M. Qin, P. V. Gratz, and A. L. N. Reddy, "Speculative paging for future nvm storage," in *MEMSYS '17*. Alexandria, Virginia: ACM, Oct. 2017, pp. 399 – 410.

[10] L. Liang, R. Chen, H. Chen, Y. Xia, K. Park, B. Zang, and H. Guan, "A case for virtualizing persistent memory," in *SoCC '16 Proceedings of the Seventh ACM Symposium on Cloud Computing*. Santa Clara, CA: ACM, Oct. 2016, pp. 126 – 140.

[11] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson, "Mojim: A reliable and highly-available non-volatile memory system," in *ASPLOS '15*. Istanbul, Turkey: ACM, Mar. 2015, pp. 3 – 18.

[12] K. Ye, D. Huang, X. Jiang, H. Chen, and S. Wu, "Virtual machine based energy-efficient data center architecture for cloud computing: A performance perspective," in *GREENCOM-CPSCOM '10*. IEEE, Dec. 2010, pp. 171–178.

[13] K. Chanchio and X.-H. Sun, "Communication state transfer for the mobility of concurrent heterogeneous computing," *IEEE Transactions on Computers*, vol. 53, pp. 1260–1273, 2004.

[14] E. R. Zayas, "Attacking the process migration bottleneck," in *SOSP '87*. Austin, TX: ACM, Nov. 1987, pp. 13–24.

[15] M. R. Hines, U. Deshpande, and K. Gopalan, "Post-copy live migration of virtual machines," *ACM SIGOPS Operating Systems Review*, vol. 43, pp. 14–26, Jul. 2009.

[16] S. Sahni and V. Varma, "A hybrid approach to live migration of virtual machines," in *CCEM '12*. Bangalore, India: IEEE, Oct. 2012.

[17] K. Ye, X. Jiang, D. Huang, J. Chen, and B. Wang, "Live migration of multiple virtual machines with resource reservation in cloud computing environments," in *CLOUD '11*. Washington, DC: IEEE, Jul. 2011.

[18] M. F. Bari, M. F. Zhani, Q. Zhang, R. Ahmed, and R. Boutaba, "CQNCR: Optimal VM migration planning in cloud data centers," in *IFIP '14*. Trondheim, Norway: IEEE, Jun. 2014.

[19] Checkpoint/Restore In Userspace (CRIU). https://www.criu.org/.

[20] C. Jo, E. Gustafsson, J. Son, and B. Egger, "Efficient live migration of virtual machines using shared storage," in *VEE '13*. Houston, TX: ACM, Mar. 2013, pp. 41–50.

[21] Emulate NVDIMM in Linux. https://nvdimm.wiki.kernel.org/.

[22] D. Baliley, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The NAS parallel benchmarks-summary and preliminary results," in *SC '91*. ACM, 1991, pp. 158–165.

[23] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications," in *PACT '08*. Toronto, Ontario, Canada: ACM, Oct. 2008, pp. 72–81.

[24] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *ISCA '95*. S. Margherita Ligure, Italy: ACM, Jun. 1995, pp. 24–36.

[25] Redis: An in-memory data structure store. http://redis.io/.

[26] B. F. Cooper, A. Silberstein, ErwinTam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *SoCC '10*. Indianapolis, Indiana: ACM, Jun. 2010, pp. 143–154.

[27] Docker container. https://www.docker.com/.

[28] "Gen-z overview," https://genzconsortium.org/wp-content/uploads/2018/05/Gen-Z-Overview.pdf.

[29] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic, "Optimizing checkpoints using nvm as virtual memory," in *IPDPS '13*. Boston, MA: IEEE, May 2013.

[30] iperf - the ultimate speed test tool for tcp, udp and sctp. https://iperf.fr/iperf-download.php.

Optimizing Memory Layout of Hyperplane Ordering for Vector Supercomputer SX-Aurora TSUBASA

Osamu Watanabe[†*], Yuta Hougi[†], Kazuhiko Komatsu[‡], Masayuki Sato[†], Akihiro Musa[‡*], Hiroaki Kobayashi[†]
[†]*Graduate School of Information Sciences, Tohoku University, Sendai, Japan.*
[‡]*Cyberscience Center, Tohoku University, Sendai, Japan.*
[*] *NEC Corporation, Tokyo, Japan.*
Email: {osamu.watanabe.t5, yuta.hougi.t8}@dc.tohoku.ac.jp, {komatsu, masa, musa, koba}@tohoku.ac.jp

***Abstract*—This paper describes the performance optimization of hyperplane ordering methods applied to the high cost routine of the turbine simulation code called "Numerical Turbine" for the newest vector supercomputer. The Numerical Turbine code is a computational fluid dynamics code developed at Tohoku University, which can execute large-scale parallel calculation of the entire thermal flow through multistage cascades of gas and steam turbines. The Numerical Turbine code is a memory-intensive application that requires a high memory bandwidth to achieve a high sustained performance. For this reason, it is implemented in a vector supercomputer equipped with a high-performance memory subsystem. The main performance bottleneck of the Numerical Turbine code is the time-integration routine. To vectorize the lower-upper symmetric Gauss-Seidel method used in this time integration routine, a hyperplane ordering method is used. We clarify the problems of the current hyperplane ordering methods for the newest vector supercomputer NEC SX-Aurora TSUBASA and propose an optimized hyperplane ordering method that changes the data layout in the memory to resolve this bottleneck. Through the performance evaluation, it is clarified that the proposed hyperplane ordering can achieve further improvement of the performance by up to 2.77×, and 1.27× on average.**

***Index Terms*—data structure, hyperplane ordering method, turbine simulation code, vector supercomputer, performance optimization**

I. Introduction

Thanks to advances in large-scale simulations, various phenomena in the real world can be reproduced more realistically by using supercomputer systems. However, there are still many issues to be addressed in the real world, and the impact of problems with social infrastructures on our society is immeasurable. There is no doubt that preventing these problems is beneficial for promoting a safe society. For example, gas and steam turbines are used for thermal power generation. However, a failure of these turbines will have serious social and economic impact.

To prevent such untoward effects, it is necessary to predict such failures in advance. However, this is very difficult for actual turbines. Therefore, it is necessary to conduct a numerical simulation of a turbine using a supercomputer to simulate various phenomena occurring in the turbine. Regarding these efforts, Industry 4.0 has been proposed internationally. For example, the manufacturing industry has rapidly moved to digitize everything from design to manufacturing and operation, and Industry 4.0 has been put to practical use at Siemens, GE, and so on.

The internal structure of a turbine is composed of a multistage stator and rotor blades, with the total number of blades exceeding 1,000. Experimentally designing these turbines in a short period is difficult in terms of cost and time. To design reliable and modern gas and steam turbines, various physical phenomena generated by thermal flow must be simultaneously addressed. To design an efficient and highly reliable turbine, it is necessary to develop a multiphysics computational fluid dynamics (CFD) technique for numerically analyzing the mathematical model that simulates these multiphysics as governing equations of thermal fluids. However, as multiphysics mutual interference in a turbine is caused by complex interactions with the total thermal flow field, it is necessary to analyze the total thermal flow field in the turbine.

To solve these problems, Tohoku University has developed a multiphysics CFD code called "Numerical Turbine" for large-scale simulations of unsteady wet steam flow with non-equilibrium condensation inside a turbine [1]. The code can be used to analyze the unsteady wet steam flow in the final multistage cascade of an actual steam turbine. The code also applies numerical solutions for analyzing the complex thermal flow generated inside the final stage of a steam turbine. The code incorporating these mathematical models can numerically elucidate the multiphysics interaction of the thermal flow inside a turbine, and we are able to determine in advance a catastrophic situation leading to turbine instability and blade destruction. Thus, multiphysics CFD is very useful for next generation turbine design.

To accurately reproduce the various phenomena occurring inside a turbine, it is necessary to reproduce conditions of galls and cracks on each blade in the turbine. Therefore, it is indispensable to simulate the whole turbine. The number of grids to be calculated exceeds 700 million, which is a huge amount of calculation to execute the simulation. To practically apply the analysis results, it is also necessary to complete the calculation within the required time.

To satisfy the various computational requirements, various processors of modern supercomputer systems have mainly adopted vector processing mechanisms such as single instruction stream, multiple data stream (SIMD) processing, for greatly improving computational performance. To use the high

computing power of a supercomputer by using such a vector process, the simulation code must be vectorized. Although the lower-upper symmetric Gauss-Seidel (LU-SGS) relaxation method [2] is used in the time integration routine, which is one of a high-cost parts of the Numerical Turbine code, it cannot be automatically vectorized. Therefore, a hyperplane ordering method [3] is adopted for vectorization. Thus, the Numerical Turbine code can be used to achieve high sustained performance by using the high vector computing capability of NEC's vector supercomputer SX series [4].

Although the computational performance of supercomputers continues to improve, improvement in memory performance has not kept up. The gap between the computational speed of the processors and data-transfer capability of memories has widened. To reduce the gap, the processors of modern vector supercomputers are equipped with an on-chip vector-cache mechanism [5] [6]. Therefore, to achieve high computational performance on such vector supercomputers, simulation codes need to effectively take into account the effect of the vector-cache. In this paper, we propose an optimized hyperplane ordering method that changes the data layout on the memory to further improve sustained performance.

The outline of this paper is as follows. In Section II, we present related work regarding optimization of hyperplane ordering methods. In Section III, we discuss the two vector supercomputers used in our study. In Section IV, we give an overview of the Numerical Turbine code and describe the conventional 2D and 3D hyperplane ordering methods and the performance results as the preliminary evaluation. In Section V, we describe our proposed optimized 3D hyperplane ordering method that changes the data layout to improve memory access. In Section VI, we discuss the performance result. We conclude the paper along with future work in Section VII.

II. Related Work

Regarding the hyperplane ordering methods implemented in the Numerical Turbine code, we previously reported a method suitable for SX-ACE [7]. In that report, we showed that the 3D hyperplane ordering method could not effectively use the vector cache implemented in SX-ACE because this method uses indirect memory accesses with long strides. Hence, we proposed a version of the 2D hyperplane ordering method as an alternative to the 3D hyperplane ordering method in that study. Although this 2D hyperplane ordering method has a shorter vector-loop length than the 3D hyperplane ordering method, the spatial locality of the data of the 2D hyperplane ordering method is higher than that of the 3D hyperplane ordering method. Thus, the vector cache can be used effectively even though its vector length is not sufficient for SX-ACE. The 2D hyperplane ordering method also does not need to use indirect memory access. As a result, its performance is 4.6 times higher than that of the 3D hyperplane ordering method. In our previous study, we argued that memory optimization is more important than calculation optimization for modern supercomputers.

TABLE I: Specifications of SX-ACE and SX-Aurora TSUBASA

	SX-ACE	***SX-Aurora TSUBASA***
Frequency	1.0 GHz	1.4 GHz
Theoretical performance / core	64 Gflop/s	268.8 Gflop/s
Number of cores	4	8
Theoretical performance / socket	256 Gflop/s	2.15 Tflop/s
Memory bandwidth	256 GB/s	1.22 TB/s
Memory capacity	64 GB	48 GB
Last level cache bandwith	1 TB/s	2.66 TB/s
Last level cache capacity	1 MB private	16 MB shared

Regarding the investigation of the optimal memory-access pattern with hyperplane ordering methods, Burger et al. showed that the memory-access performance improved by changing the data layout of the 3D array used in the 3D hyperplane ordering method [8]. This is a technique to improve the efficiency of memory access by continuously storing the data elements and creating a 2D array on each hyperplane. In the technique, each row on the 2D array is vectorized. Therefore, the vectorized loop length is still smaller than the loop length of the 3D hyperplane ordering method.

III. Modern Vector Supercomputers

The number of cores in modern vector supercomputer processors has been increasing, and the theoretical computational performance of the processors is also improving. Although the memory bandwidth has been improving, the improvement rate is lower than that of computational performance. To fill the performance gap between computation and memory, vector supercomputers are equipped with a cache mechanism for vector processing. Effectively using such a vector cache is indispensable for achieving a high computational performance in executing an application on a vector supercomputer. This section outlines the vector supercomputers SX-ACE and SX-Aurora TSUBASA, which are suitable for memory-intensive applications. Table I lists the hardware specifications of SX-ACE and SX-Aurora TSUBASA.

A. SX-ACE

SX-ACE is the previous generation vector supercomputer, launched by NEC in 2013 [5]. The SX-ACE processor is composed of four vector cores, each consisting of a vector processing unit (VPU) and assignable data buffer (ADB). The VPU is an important component of SX-ACE and can process up to 256 vector elements of 8 bytes, each with one vector instruction. The VPU is equipped with two multiply units and two addition units, which can operate independently with different vector instructions. The theoretical computational performance per core of SX-ACE is 64 Gflop/s, and the theoretical computational performance of one processor is 256 Gflop/s. Each core is connected to a memory control unit (MCU) via a memory crossbar network at a memory bandwidth of 256 GB/s, and the bandwidth is shared by four cores. Each core is equipped with a 1 MB ADB. This is a software-controllable data buffer that the VPU can access at a rate of 256 GB/s.

B. SX-Aurora TSUBASA

SX-Aurora TSUBASA is the newest vector supercomputer that NEC launched in 2017 [6]. The SX-Aurora TSUBASA system consists of a vector host (VH) and vector engines (VEs). A VE is mounted as a PCI Express (PCIe) card equipped with a vector processor, and the card is connected to the VH, which is a standard x86/Linux node, via the PCIe. An entire program with data executes on the VE with a dedicated high bandwidth memory, while the VH mainly provides OS functions to connected VEs. Up to eight VEs can be controlled by one VH. A VE consists of eight vector cores, a 16-MB last level cache (LLC), and six High Bandwidth Memory 2 (HBM2) memory modules. The VPU installed in the vector core has three vector-fused multiply-add (VFMA) units. The vector length of SX-Aurora TSUBASA is 256, same as that of SX-ACE, and 256 vector elements can be processed with one vector instruction. The theoretical computational performance per core of a VE is 268.8 Gflop/s, and the theoretical computational performance of one VE is 2.15 Tflop/s. The LLC is directly connected to the vector register of each core and shared by eight cores. The six HBM2 memory modules have a high memory bandwidth of 1.288 TB/s in total.

IV. Overview of Numerical Turbine code

The physical phenomenon in steam turbines is nonequilibrium condensation flows. The condensation observed in steam turbines is quite important in engineering. The phase change is governed by homogeneous nucleation and the nonequilibrium process of condensation. The latent heat of water is released to the surrounding non-condensed vapor, increasing its temperature and pressure. It is known that condensed water droplets affect the performance of a steam turbine. The blades of a steam turbine are occasionally damaged by erosion due to interaction with the condensed water droplets. However, the precise mechanism behind the erosion remains unknown. Transonic wet-steam flows in a steam-turbine cascade channel have been studied [9] [10] [11]. Young [11] calculated two-dimensional wet-steam turbine cascade flows by solving the Euler equation with the Lagrangian method for integrating the growth equation of a water droplet through each streamline.

A. Fundamental equations

The fundamental equations consist of the conservation laws of total density, momentum, total energy, liquid water density, and number density of water droplets with the shear stress transport (SST) turbulence model [12], as shown in the following equation.

$$\frac{\partial Q}{\partial t} + \frac{\partial F_i}{\partial \xi_i} + S + H = 0, \tag{1}$$

where Q, F_i, S, H, t, and ξ_i $(i = 1, 2, 3)$ are the vector of unknown variables, flux, the viscous term, source term, physical time, and general curvilinear coordinates, respectively. Also, the equations of state and the sound speed in wet steam, assuming that the condensate mass fraction β is sufficiently small ($\beta < 0.1$), are written as

$$p = \rho RT(1-\beta), \quad c^2 = \frac{C_{pm}}{C_{pm} - (1-\beta)R}\frac{p}{\rho}, \tag{2}$$

where R is the gas constant and C_{pm} is defined as the linear combination of the isobaric specific heats in the gas and liquid phases by using β. Here, Q, F_i, S, and H are defined in the following forms:

$$Q = J\begin{bmatrix} \rho \\ \rho w_1 \\ \rho w_2 \\ \rho w_3 \\ e \\ \rho\beta \\ \rho n \\ \rho k \\ \rho\omega \end{bmatrix}, \; F_i = J\begin{bmatrix} \rho W_i \\ \rho w_1 W_i + \partial\xi_i/\partial x_1 p \\ \rho w_2 W_i + \partial\xi_i/\partial x_2 p \\ \rho w_3 W_i + \partial\xi_i/\partial x_3 p \\ (e+p)\, W_i \\ \rho\beta W_i \\ \rho n W_i \\ \rho k W_i \\ \rho\omega W_i \end{bmatrix},$$

$$S = -J\frac{\partial\xi_i}{\partial x_j}\frac{\partial}{\partial\xi_i}\begin{bmatrix} 0 \\ \tau_{1j} \\ \tau_{2j} \\ \tau_{3j} \\ \tau_{kj}w_k + (\kappa+\kappa^t)\,\partial T/\partial x_j \\ 0 \\ 0 \\ \sigma_{kj} \\ \sigma_{\omega j} \end{bmatrix},$$

$$H = -J\begin{bmatrix} 0 \\ 0 \\ \rho\left(\Omega^2 x_2 + 2\Omega w_3\right) \\ \rho\left(\Omega^2 x_3 + 2\Omega w_2\right) \\ 0 \\ \Gamma \\ I \\ S_k \\ S_\omega \end{bmatrix},$$

where the nomenclature is as follows.

J	Jacobian for transformation
W_i	Relative contra-variant velocities
e	Total internal energy per unit volume
κ	Laminar thermal conductivity coefficient
β	Condensate mass fraction (wetness)
κ^t	Turbulent thermal conductivity coefficient
n	Number density of water droplets per unit mass
σ_{kj}	Diffusion term for k equation
k	Turbulent kinetic energy
$\sigma_{\omega j}$	Diffusion term for ω equation
ω	Turbulent kinetic energy dissipation ratio
Ω	Angular velocity of rotation
ρ	Total density
p	Static pressure
T	Static temperature
w_i	Relative physical velocities
x_i	Cartesian coordinates
τ_{ij}	Viscous stress tensors

B. Condensation model

The mass generation rate of liquid phase, Γ, is based on the classical condensation theory. Ishizaka et al. further simplified the following equation [13].

$$\Gamma = \frac{4}{3}\pi\rho_l \left(Ir_*^3 + 3\rho n r^2 \frac{dr}{dt} \right). \tag{3}$$

C. Numerical schemes

To calculate these equations, the Numerical Turbine code uses the fourth-order compact MUSCL TVD (Compact MUSCL) scheme [14] and the Roe approximate Riemann solver [15] for the finite-difference scheme in Eq. (1). The viscous term is calculated using the second-order central-difference scheme. The SST turbulence model is used for the turbulence modeling [16], and the LU-SGS method [2] is used for time integration.

Figure 1 shows the flowchart of the main iteration loop of the Numerical Turbine code. Its performance is primarily dominated by calculations of the space difference, time integration, and physical model routines.

D. 2D and 3D hyperplane ordering methods

Since the Numerical Turbine code is a memory-intensive code [4], it has been executed on vector supercomputers with high memory bandwidth. The space difference routine and physical model routine are automatically vectorized by the SX Fortran compiler because of non-dependency among computational data. The LU-SGS method used in the time integration routine cannot be automatically vectorized, because the Gauss-Seidel method has dependencies on the calculated results of its predecessor.

The Gauss-Seidel method has data dependency as shown in Fig. 2. As this figure shows, grid point $q(i,j,k)$ refers to grid points $q(i-1,j,k)$, $q(i,j-1,k)$, and $q(i,j,k-1)$ in the i-, j-, and k-directions for the calculation, and the calculation of grid point $q(i,j,k)$ depends on grid points $q(i-1,j,k)$, $q(i,j-1,k)$, and $q(i,j,k-1)$. Therefore,

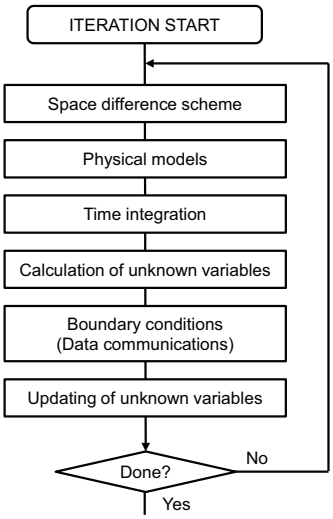


Fig. 1: Flowchart of the iteration loop of the Numerical Turbine code.

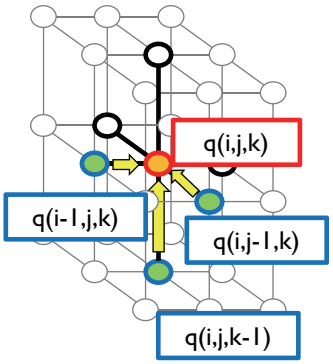
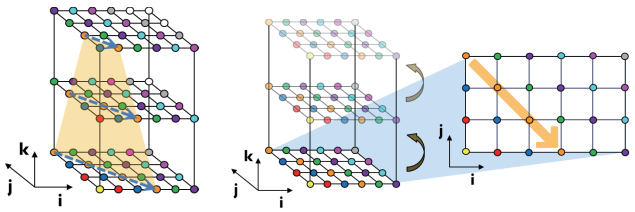


Fig. 2: Data dependency on the LU-SGS method.

the method cannot be vectorized. To vectorize this method used in the time integration routine of the Numerical Turbine code, a hyperplane ordering method is applied to avoid such data dependencies. Although the time integration routine is vectorized by applying a hyperplane ordering method, the main performance bottleneck in the iteration loop is still the time integration routine. Therefore, it is necessary to improve the computational performance of this routine.

The hyperplane ordering method is a parallelization method that avoids such data dependencies by changing the order of calculation. As described in Fujino et al.'s study [3], there are 2D hyperplane ordering and 3D hyperplane ordering methods. Figure 3(a) is a diagram of the ordering with the 3D hyperplane ordering method. In this figure, the orange plane is composed of grid points that are not dependent on each other, and each plane is a skew-cutting plane through the grid points that are ordered in the 3D space. Such a plane is called a "hyperplane." When calculating grid point $q(i,j,k)$ on this hyperplane, grid points $q(i-1,j,k)$, $q(i,j-1,k)$, and $q(i,j,k-1)$ are not updated. Therefore, the data dependency related to grid point $q(i,j,k)$ can be avoided, and the calculation of the grid points on the hyperplane can be vectorized.



(a) 3D hyperplane. **(b)** 2D hyperplane.

Fig. 3: Hyperplane ordering.

With the 3D hyperplane ordering method, a loop length for the vectorization can be generally increased. Since indirect memory access is used, however, long stride memory access occurs and the memory load increases. Vector supercomputers prior to SX-ACE had a high ratio of a memory bandwidth to a high floating-point operation ratio, known as Bytes per Flop ratio. Therefore, the 3D hyperplane ordering method is effective in increasing the loop length for exploiting the high sustained performance of such vector supercomputers.

Although the computational performance and memory bandwidth of the previous model SX-ACE have improved, the improvement rate of memory bandwidth is lower than that of computational performance. Therefore, the 3D hyperplane ordering method cannot exploit the high computational performance of SX-ACE due to the high memory load caused by the method.

To exploit the performance of SX-ACE, we have the 2D hyperplane ordering method to effectively use an ADB, which is an on-chip memory of SX-ACE. As shown in Fig. 3(b), a hyperplane of the 2D hyperplane ordering method does not consist of a set of grid points with no dependency in the 3D space. However, there is a set of grid points with no dependency on a 2D plane, and each hyperplane in the 2D hyperplane ordering method can be vectorized with the higher locality of data reference than that with the 3D hyperplane ordering method. Since the 2D hyperplane ordering method has a simple structure, it uses direct memory accesses to refer to the grid points by calculating the location of these points instead of the indirect memory accesses, resulting in reducing memory load.

E. Preliminary evaluation

To clarify the performance characteristics of the hyperplane ordering method, the preliminary evaluation is conducted. Two types of kernel codes of the 2D and 3D hyperplane ordering methods are created for this evaluation. The performance evaluation of these methods using these kernel codes is conducted using one node of each SX-ACE and SX-Aurora TSUBASA. One node of SX-ACE corresponds to one CPU, and one node of SX-Aurora TSUBASA corresponds to one VE.

To evaluate the performance of the hyperplane ordering methods in various problem sizes, the matrix size of the kernel codes is changed. The matrix size is N, N, and 181 in the i-, j-, and k-directions, respectively, and the range of N is from 31 to 141. These matrix sizes are set in consideration of the actual model sizes used in the Numerical Turbine code.

Figure 4 shows the results of the performance evaluation of the 2D hyperplane ordering and the 3D hyperplane ordering methods on SX-ACE and SX-Aurora TSUBASA. As shown in Fig. 4(a), in SX-ACE, when matrix size N is 54 or more, the performance of the 2D hyperplane ordering method is higher than that of the 3D hyperplane ordering method. On the other hand, on SX-Aurora TSUBASA, as shown in Fig. 4(b), the performance of the 3D hyperplane ordering method is better than that of the 2D hyperplane ordering method regardless of the matrix size. The high computational performance of SX-ACE can be exploited when the cache hit ratio and average vector length is balanced well. However, on SX-Aurora TSUBASA, the cache hit ratio is almost 100% regardless of any matrix size for the 2D hyperplane ordering method.

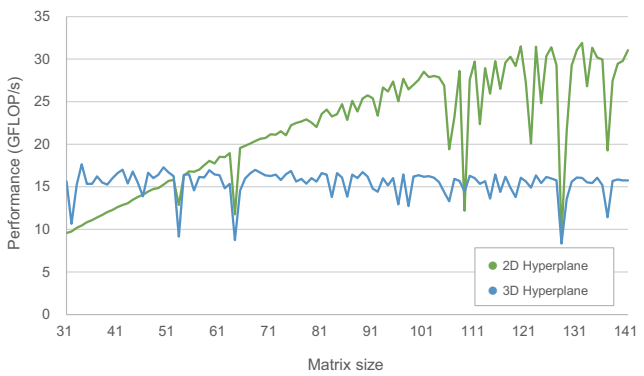
It is clear from Fig. 4(b) that there is a cross-point of the performance of the 2D hyperplane ordering method and the 3D hyperplane ordering method at a lager matrix size. However, the matrix size considered in the actual simulation is from 31 to 141. Therefore, the matrix size smaller than the cross-point is appropriate for the evaluation.

Figure 4 indicates that precipitous performance degradation appears at some matrix sizes in both SX-ACE and SX-Aurora TSUBASA . The memory system of both systems uses a multi-bank interleave memory, and the degradation is due to the bank conflict. With regard to the performance of the 2D hyperplane ordering method in SX-Aurora TSUBASA, the degradation is greatly mitigated. This is because the decrease in memory access is caused by the high cache hit ratio in SX-Aurora TSUBASA.

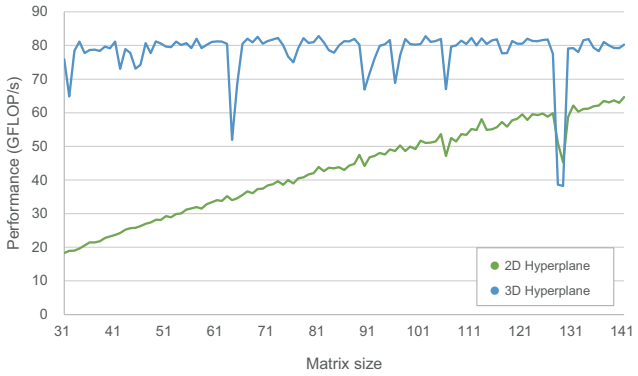
This preliminary evaluation suggests that SX-Aurora TSUBASA prefers the 3D hyperplane ordering method rather than the 2D one because the 3D hyperplane ordering method provides a longer vector length, even with an ineffective use of the cache. That is, on SX-Aurora TSUBASA, the increase in the cache capacity may diminish the effectiveness of the improvement in the cache hit ratio. In such a case, it has become obvious that the effect of other factors, such as vector length, on the performance of these hyperplane ordering methods increases. However, since the 3D hyperplane ordering method is accompanied by a high memory load, it is necessary to reduce indirect memory accesses causing memory load to further improve performance.

V. Optimizing Memory-Access Pattern of 3D Hyperplane Ordering Method

The 3D hyperplane ordering method is expected to be more effective than the 2D hyperplane ordering method on SX-Aurora TSUBASA. However, it has drawbacks: the distance of stride access in the memory is long and indirect memory access is used for accessing each grid point. It is necessary to mitigate the drawbacks to make calculation with the 3D hyperplane ordering method even faster. Therefore, we propose an optimized 3D hyperplane ordering method to improve

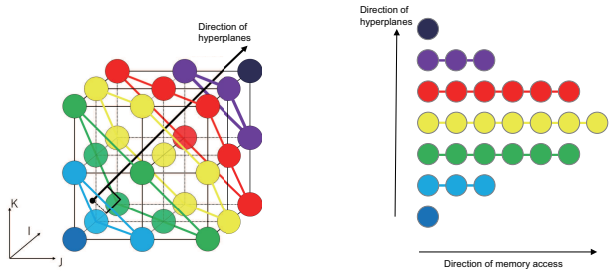


(a) Performance on SX-ACE.



(b) Performance on SX-Aurora TSUBASA.

Fig. 4: Performance of the 2D and 3D hyperplane ordering methods.



(a) Original 3D data layout. **(b)** New 1D data layout.

Fig. 5: Changing 3D data layout to 1D data layout.

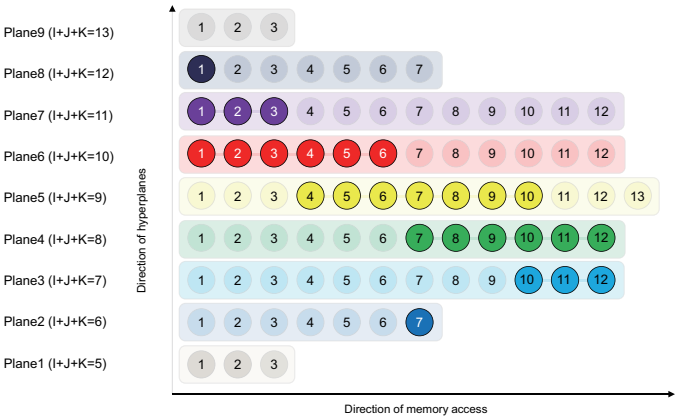


Fig. 6: 1D data layout and each plane number.

memory access performance by changing the data layout on the memory. With the proposed method, the grid points on each hyperplane of the 3D hyperplane ordering method are continuously stored in a 1D array so that the grid points into the 1D array corresponding to each hyperplane can be sequentially accessed.

A. Changing data layout on memory for sequential access

Figure 5(a) shows the data layout from the 3D hyperplane ordering method and Fig. 5(b) shows the new 1D data layout of the proposed method. As shown in Fig. 5(a), with the 3D hyperplane ordering method, each hyperplane is a skew-cutting plane through the 3D space. Thus, the distance to the adjacent grid points on the hyperplane on the memory is long, and indirect memory accesses are used for accessing grid points on each hyperplane. To reduce these indirect memory accesses in the new 1D data layout in the proposed method, grid points on each hyperplane are arranged in order of calculation. Thus, the grid points of each hyperplane are stored in a 1D array corresponding to the hyperplane, as shown in Fig. 5(b). Hence, the memory can be continuously accessed by using the grid points in each 1D array.

For each grid point (i, j, k) on a hyperplane, the sum of i, j, and k is the same value, and each hyperplane can be indexed using this sum. Therefore, as shown in Fig. 6, each 1D array corresponding to the hyperplane is arranged in the order of hyperplanes, and grid points $(i-1, j, k)$, $(i, j-1, k)$, and $(i, j, k-1)$, which are referred for the calculation of grid point (i, j, k) in a 1D array, are in the adjacent 1D array. To be described later, this data layout is created using an array of structures whose member is a 1D array, and each hyperplane is specified using the index of the array of this structure. In Fig. 6, for example, to access the fifth hyperplane (plane 5), the fifth element of the structure array is referred to. When storing grid points in each 1D array, it is necessary to store the grid points on the boundary area on each hyperplane as shown in Fig. 6. Translucent circles indicate grid points on the boundary areas.

B. Implementation of memory-efficient data layout

To create this data structure, the structure is dynamically allocated with the number of hyperplanes and number of grid points on each hyperplane. The specific allocation method is shown in Listing 1. Here, these 1D arrays are allocated in the hyperplane order shown in Fig 6.

Derived type `ARRAY1D` is declared as a hyperplane structure. This structure has member `ARRAY`, and this member is a dynamically allocatable array to store grid points on a hyperplane. Then, array `A_` in Listing 1 is declared as a

Listing 1: Define and allocate dynamic allocatable arrays

```
  ...
  TYPE ARRAY1D
    REAL*8,ALLOCATABLE,DIMENSION(:):: ARRAY
  END TYPE ARRAY1D
  TYPE(ARRAY1D),PUBLIC,ALLOCATABLE,DIMENSION(:):: A_
  ...
  ALLOCATE(A_(0:NUM_OF_PLANES+1))
  ...
  DO M=0,NUM_OF_PLANES+1
    ...
    ALLOCATE(A_(M)%ARRAY(MDIM1_MAX(M)))
    ...
  ENDDO
  ...
```

Listing 2: Copy data to the 1D layout arrays

```
  ...
  DO K=KK1,KKF
    DO J=JJ1,JJF
!$NEC vovertake
      DO I=II1,IIF
        LHP = LIST_3DA(I,J,K,L)
        ...
        M = I+J+K-(MMIN-1)
        A_(M)%ARRAY(LHP) = A(I,J,K)
      ENDDO
    ENDDO
  ENDDO
  ...
```

Listing 3: 3D hyperplane ordering using 1D layout arrays

```
...
DO LHP=1,LIST_3DC(M,L)
  I = LIST_3DI(LHP,M,L)
  J = LIST_3DJ(LHP,M,L)
  K = LIST_3DK(LHP,M,L)
  IM=LIST_3DA(I-1,J  ,K  ,L)
  JM=LIST_3DA(I  ,J-1,K  ,L)
  KM=LIST_3DA(I  ,J  ,K-1,L)
  ...
  AIM = (A_(M-1)%ARRAY(IM)*A_(M)%ARRAY(LHP))*0.5D0
  AJM = (A_(M-1)%ARRAY(JM)*A_(M)%ARRAY(LHP))*0.5D0
  AKM = (A_(M-1)%ARRAY(KM)*A_(M)%ARRAY(LHP))*0.5D0
  ...
ENDDO
...
```

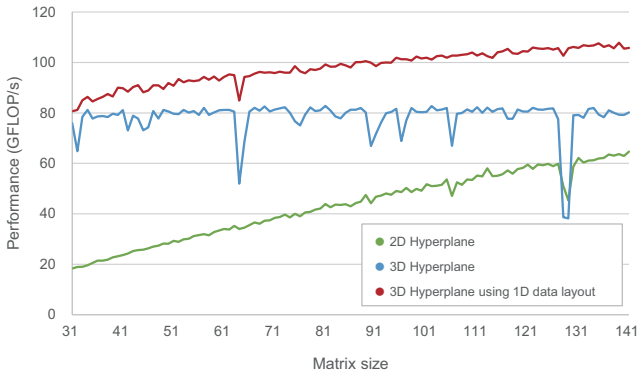


Fig. 7: Performance of three types of the hyperplane ordering method on SX-Aurora TSUBASA.

dynamically allocatable array of the structure. Next, the array is allocated according to the number of hyperplanes. Finally, structure member `ARRAY` is allocated to store each hyperplane according to the number of grid points on the hyperplane. Each element of array `MDIM1_MAX` stores the number of grid points on each hyperplane.

With the proposed method, as shown in Listing 2, it is necessary to copy data in the arrays into the 1D arrays. This data-copy is not, of course, necessary in the original 3D hyperplane ordering method. Array `LIST_3DA` stores the position of each grid point in each 1D array. Specifically, the number of the respective grid points is stored, as shown in Fig. 6. Such data copies are a time-consuming process because the copies occur every time the time integration routine is called. To shorten this time, the compiler directive `vovertake` of SX-Aurora TSUBASA is used. With this directive, all vector stores in the loop might be over-taken by the subsequent vector load so that the data copy time can be reduced.

Listing 3 shows the calculation part when using the proposed method. In Listings 1, 2, and 3, `M` indicates the hyperplane number. In the original code, an access to grid points `A(I,J,K)`, `A(I-1,J,K)`, `A(I,J-1,K)` and `A(I,J,K-1)` is an indirect memory access. On the other hand, in the code using the 1D array, accesses to `A_(M)%ARRAY(LHP)` corresponding to `A(I,J,K)` are sequential, and the number of indirect memory accesses can be reduced. In the code using a 1D array, since the 1D arrays are arranged in the order of hyperplane numbers as the array of structure, adjacent grid points can be found in the adjacent array of the structure, and the access distance to adjacent grid points is closer than that with the 3D hyperplane ordering method.

VI. Performance Evaluation

We evaluate the performance of the proposed method compared with the 3D hyperplane ordering method on SX-Aurora TSUBASA. Figure 7 shows the evaluation results. The horizontal axis indicates the matrix size, and the vertical axis indicates the computational performance of these methods. The proposed method using the 1D arrays performes better than the original 3D hyperplane ordering method for any matrix size. The performance improves by up to 2.77×, and 1.27× on average. The performance of the proposed method improves as the matrix size increases.

The reason why the performance of the proposed method increases as the matrix size increases is the ratios of calculation time to data-copy time for calculation preparation. Figure 8 shows these ratios as a function of the matrix size with the proposed method. The horizontal axis indicates the matrix size, and the vertical axis indicates these ratios. The blue portion indicates the ratio of calculation time and the red

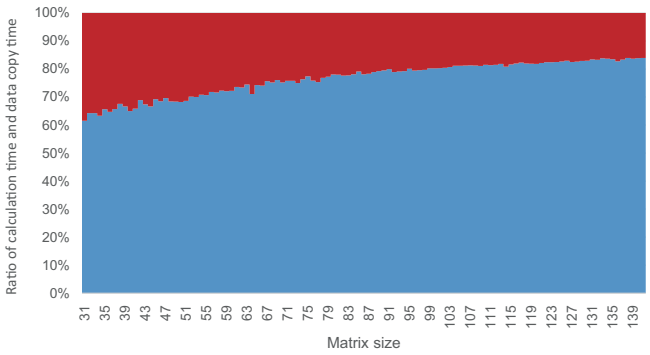


Fig. 8: Cost distribution of data copy and calculation with proposed method.

portion indicates that of data-copy time. These results indicate that as the matrix size increases, the ratio of calculation time increases; thus, the performance of the time integration routine improves. In other words, reducing the data-copy time can result in superior performance of the proposed method.

VII. Conclusions

We discuss two types of the hyperplane ordering methods used for the time integration routine of the Numerical Turbine code on the newest vector supercomputer SX-Aurora TSUBASA. With the 2D hyperplane ordering method, although the cache hit rate is almost 100%, computational performance of the short vector length harmfully affects the sustained performance. On the other hand, the 3D hyperplane ordering method provides a longer vector length, even with an ineffective use of the cache. Since the 3D hyperplane ordering method uses indirect memory accesses, long stride memory accesses occur and the memory load increases.

To further improve the performance of the 3D hyperplane ordering method to reduce the indirect memory accesses, we propose an optimized 3D hyperplane ordering method using a 1D data layout. To sequentially access the grid points, we rearrange the data layout to access the grid points of each hyperplane in a 1D array fashion. Thus, the memory for the grid points in a 1D array can be continuously accessed. To demonstrate the effectiveness of the proposed method, we evaluated its performance. The experimental results indicate that the proposed method reduces the execution time for the Gauss-Seidel kernel by up to 2.77×, and 1.27× on average, compared with the 3D hyperplane ordering method method without the 1D array layout, and its performance further improves as the matrix size becomes larger. We confirm that indirect memory accesses can be reduced by enabling sequential memory accesses for the 3D hyperplane ordering method, resulting in further performance improvement.

Although we evaluate the proposed method using a kernel code in this study, we will implement the method in whole Numerical Turbine code and evaluate the effect of the method on the performance. We will also examine the effect of the method on widely used SIMD architectures with larger cache capacity such as Intel Xeon and AMD EPYC processors.

Acknowledgment

This research was supported in part by MEXT as "Next Generation High-Performance Computing Infrastructures and Applications R&D Program," entitled "R&D of A Quantum-Annealing-Assisted Next Generation HPC Infrastructure and its Applications." The authors thank Satoru Yamamoto, Takashi Furusawa, and Hironori Miyazawa of Tohoku University for their fruitful discussions and variable comments.

References

[1] S. Miyake, I. Koda, S. Yamamoto, Y. Sasao, K. Momma, T. Miyawaki, and H. Ooyama, "Unsteady Wake and Vortex Interaction in 3-D Steam Turbine Low Pressure Final Three Stages," Proc. ASME Turbo Expo 2014, Düsseldorf, Germany, GT2014-25491, 2014.

[2] S. Yoon and A. Jameson, "Lower-upper Symmetric-Gauss-Seidel method for the Euler and Navier-Stokes equations," AIAA Journal, 26 (1988), 1025-1026.

[3] S. Fujino, M. Mori, and T. Takeuchi, "Performance of hyperplane ordering on vector computers," Journal of Computational and Applied Mathematics, 38 (1991), 125-136, North-Holland.

[4] H. Kobayashi, "Feasibility Study of a Future HPC System for Memory-Intensive Applications: Final Report," Sutained Simulation Performance 2014, Springer, (2014), 3-16.

[5] R. Egawa, K. Komatsu, S. Momose, Y. Isobe, A. Musa, H. Takizawa, and H. Kobayashi, "Potential of a Modern Vector Supercomputer for Practical Applications - Performance Evaluation of SX-ACE," The Journal of Supercomputing 73(9), 3948-3976 (2017), DOI: 10.1007/s11227-017-1993-y.

[6] K. Komatsu, S. Momose, Y. Isobe, O. Watanabe, A. Musa, M. Yokokawa, T. Aoyama, M. Sato, and H. Kobayashi, "Performance Evaluation of a Vector Supercomputer SX-Aurora TSUBASA," Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis,2018.

[7] *26th International Conference on Parallel Computational Fluid Dynamics, Parallel CFD 2016*, http://www.conf.kit.ac.jp/parcfd2016/conference.html.

[8] M. Burger and C. Bischof, "Optimization the memory access performance of FASTEST's sipsol routine," 6th European Conference on Computational Fluid Dynamics, ECFD VI, July 2014, Barcelona, Spain. DOI: 10.13140/RG.2.2.32568.14089.

[9] F. Bakhtar and M.T. Mohammadi Tochai, "An Investigation of Two-Dimensional Flows of Nucleating and Wet Steam by the Time-Marching Method," Int. J. Heat Fluid Flow, Vol.2, No.1(1980), pp.5-18.

[10] M. Moheban and J. B. Young, "A Study of Thermal Nonequilibrium Effect in Low-Pressure Wet-Steam Turbine Using a Blade-to-Blade Time-Marching Technique," Int. J. Heat and Fluid Flow, Vol.6, (1985), pp.269-278.

[11] J. B. Young, "Two-Dimensional, Nonequilibrium, Wet-Steam Calculations for Nozzles and Turbine Cascades," Trans. ASME, J. Turbomachinery, 114 (1992), 569-579.

[12] F. R. Menter, "Two-equation Eddy-viscosity Turbulence Models for Engineering Applications," AIAA Journal, 32 (1994), 1598-1605.

[13] K. Ishizaka, T. Ikohagi, and H. Daiguji, "A High-Resolution Finite Difference Scheme for Supersonic Wet-steam Flow," Proceedings of 6th International Symposium on Computational Fluid Dynamics (in Japan), Vol.1, (1995), pp.479-484.

[14] S. Yamamoto and H. Daiguji, "Higher-Order-Accurate Upwind Schemes for Solving the Compressible Euler and Navier-Stokes Equations," Computers and Fluids, 22 (1993), 357-372.

[15] P. L. Roe, "Approximate Riemann Solvers, Parameter Vectors, and Difference Schemes," J. Comp. Phys., 43 (1981), 357-372.

[16] F. R. Menter, "Two-equation Eddy-viscosity Turbulence Models for Engineering Applications," AIAA Journal, 32 (1994), 1598-1605.

Generalized Sparse Matrix-Matrix Multiplication for Vector Engines and Graph Applications

Jiayu Li
Intelligent Systems Engineering
Indiana University
Bloomington, USA
jl145@iu.edu

Fugang Wang
Intelligent Systems Engineering
Indiana University
Bloomington, USA
fuwang@indiana.edu

Takuya Araki
Data Science Research Laboratories
NEC
Kanagawa, Japan
t-araki@dc.jp.nec.com

Judy Qiu
Intelligent Systems Engineering
Indiana University
Bloomington, USA
xqiu@indiana.edu

***Abstract*—Generalized sparse matrix-matrix multiplication (SpGEMM) is a key primitive kernel for many high-performance graph algorithms as well as for machine learning and data analysis algorithms. Although many SpGEMM algorithms have been proposed, such as ESC and SPA, there is currently no SpGEMM kernel optimized for vector engines (VEs). NEC SX-Aurora is the new vector computing system that can achieve high performance by leveraging high bandwidth memory of 1.2TB/s and long vector of VEs, where the execution of scientific applications is limited by memory bandwidth. In this paper, we demonstrate significant initial work of SpGEMM kernel for a vector engine and implement it to vectorize several essential graph analysis algorithms: Butterfly counting and Triangle counting. We propose a SpGEMM algorithm with a novel hybrid method based on sparse vectors and loop raking to maximize the length of vectorizable code for vector machine architectures. The experimental results show that the vector engine has advantages on more massive data sets. This work contributes to the high performance and portability of the SpGEMM kernel to a new family of heterogeneous computing systems, which is Vector Host (VH) equipped with different accelerators or VEs.**

***Index Terms*—Sparse Linear Algebra Kernel, NEC Vector Engine, Graph**

I. Introduction

Generalized sparse matrix-matrix multiplication (SpGEMM) is a primitive kernel for many high-performance Graph analytics and Machine Learning algorithms. Although many SpGEMM algorithms have been proposed, there is currently no SpGEMM kernel optimized for vector engines. The NEC SX-Aurora TSUBASA is a vector processor of the NEC SX architecture family[1], a CPU Machine with Vector Engine (VE) for accelerated computing using vectorization. The concept is that the full application runs on the high-performance Vector Engine, and the operating system tasks are taken care of by the Vector Host (VH), which is a standard x86 server.

As shown in Figure 1, a NEC SX-Aurora node, also called a ***Vector Island (VI)***, is comprised of a ***Vector Host (VH)*** and one or more ***Vector Engines (VEs)***. The VH is an x86 server with one or more standard server CPU running Linux operating system. One or multiple VEs are connected to each VH CPU. Inside each VE, it has 8 cores and dedicated memory.

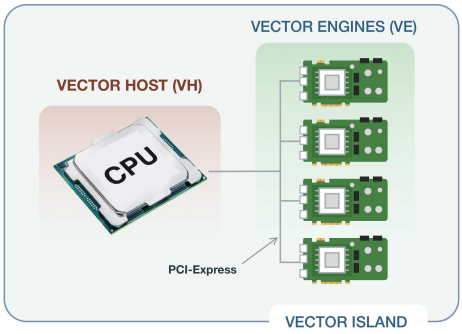


Fig. 1. Hardware configuration of NEC SX-Aurora VH and VEs

Figure 2 shows the detailed architecture of a VE. The Vector Engine Processor integrates 8 vector-cores and 48 GB of high bandwidth memory (HBM2), providing a peak performance of up to 2.45 TeraFLOPS. The computational efficiency is achieved by the unrivaled memory bandwidth of up to 1.2 TB/s per CPU and by the latency-hiding effect of the vector architecture. The single Vector Engine Processor core arithmetic unit can execute 32 double-precision floating-point operations per cycle with its vector registers holding 256 floating-point values. With 3 fused-multiply-add units (FMA), each core has a peak performance of 192 FLOP per cycle or up to 307.2 GigaFLOPS (double precision). The Vector Engine has a peak performance of up to 2.45 TeraFLOPS.

In this paper, we design a new SpGEMM kernel for the vector engine (VE) as an addition to the family of accelerators and further study two subgraph counting algorithms: Triangle counting [2] and Butterfly counting [3]. Our main contributions in this paper are:

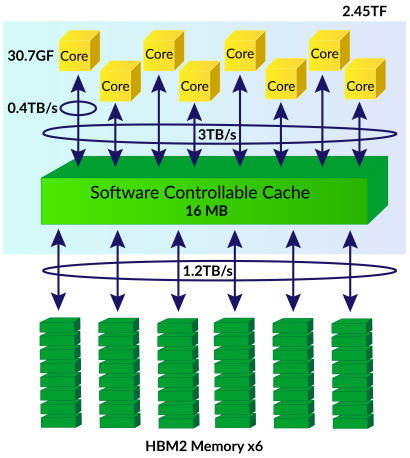


Fig. 2. NEC SX-Aurora VE architecture

- We propose a unique hybrid method that enlarges vector length for non-zeros values and leverages the High Bandwidth Memory (HBM). This enables vector architectures to exert their potentials at 1.2TB/s of HBM and 256 elements of long vector length.
- We deploy loop raking to vectorize a loop and increase the memory access efficiency.
- The SpGEMM kernel is used to implement several important graph analysis algorithms on the vector machine.

The experimental results show that the vector engine achieves high performance on large data sets. We implemented the algorithms in C++ and have made the open-source code available on Github[1][4].

II. Related work

Counting subgraphs from a large network is fundamental in graph problems. It has been used in real-world applications across a range of disciplines, such as in bioinformatics [5], social networks analysis, and neuroscience [6]. Many graph algorithms have been previously presented in the language of linear algebra [7], [8]. The matrix-based triangle counting algorithm we use is mainly based on the text [2]. [3] proposed a fast butterfly counting algorithm, which we converted into a matrix operation form. We make it highly parallel and take full advantage of vector machines.

SX-Aurora TSUBASA is a vector engine developed by NEC. Comparison of SX-Aurora TSUBASA with other computing architectures includes: [9][10]. The hybrid SpGEMM algorithm we propose is based on the following related work: ESC Algorithm [11], Hash-based SpGEMM [12] [13], and SPA Algorithm [14]. They will be explained in the next section, together with our implementation.

III. Implementation of SPGEMM on Vector Engine

In the Compressed Sparse Row (CSR) format, we represent a matrix $\mathbf{M}_{m\times n}$, by three 1-D arrays or vectors called as `A`, `IA`, `JA`. Let NNZ denote the number of non-zero elements in $\mathbf{M}$.

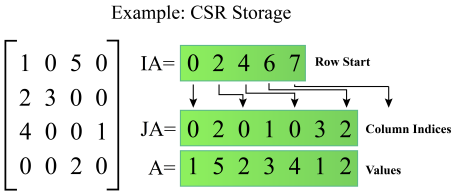


Fig. 3. CSR representation of a sparse matrix

The `A` vector is of size NNZ, and it stores the values of the non-zero elements of the matrix. The values appear in the order of traversing the matrix row-by-row. The `JA` vector stores the column index of each element in the `A` vector. The `IA` vector is of size $m+1$ stores the cumulative number of non-zero elements up to (not including) the i-th row. For example, to calculate the number of non-zero elements in row 0, just calculate `IA[1]-IA[0] = 2` $= NNZ_{row0}$.

SpGEMM can be used for various kinds of graph algorithms, including triangle counting and butterfly counting that are addressed in this paper. Since both of the matrices are sparse, its implementation is not straightforward; there exist several algorithms called ESC algorithm, the hash-based algorithm, and sparse accumulator (SPA).

Figure 4 shows basic structure of SpGEMM algorithm. It shows $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$; we assume that the sparse matrices are in CSR format.

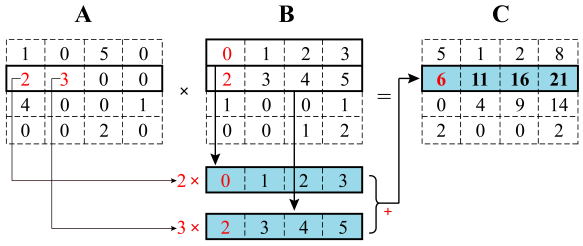


Fig. 4. Basic structure of SpGEMM algorithm

Here, each row of $\mathbf{A}\cdot\mathbf{B}$ creates each row of $\mathbf{C}$; we focus on *kth* row. The *kth* row of $\mathbf{A}$ has two non-zero elements, whose column indices are *i* and *j*. Because other elements are zero, we only need to care about *ith* and *jth* row of the matrix $\mathbf{B}$. They are multiplied by the corresponding non-zero elements of the *kth* row of $\mathbf{A}$ and added to create the *kth* row of $\mathbf{C}$. There are several algorithms to add these sparse vectors.

We implemented these algorithms on SX-Aurora TSUBASA using the technique called loop raking, and propose a novel hybrid method. To the best of our knowledge, this is the first attempt to implement SpGEMM on a vector architecture.

A. Loop raking

Loop raking is a long-forgotten technique that was proposed in the early '90s to implement radix sort [15]. However, it is

[1]https://github.com/dsc-nec/frovedis_matrix

an essential technique to enhance vectorization and enlarge vector length. The key idea of loop raking is viewing each element of a vector register as a virtual processor. Here we take the union of sets as an example to introduce the loop raking technique.

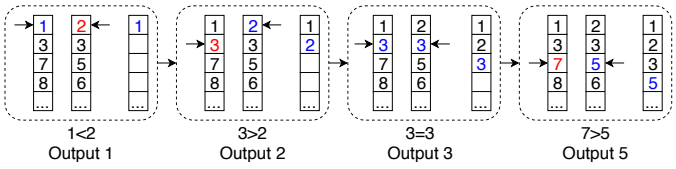
Set operations (union, intersection, merge, difference, etc.) of sorted integer can be easily implemented, but vectorizing them is not trivial.

The traditional algorithm compares two lists one by one, and the result of the comparison will determine which pointer is increased. It contains the following steps:

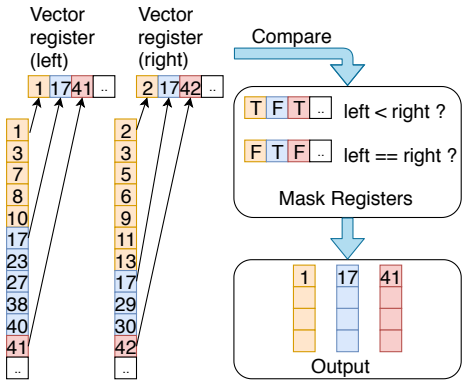
1) Set the pointers to the first elements of the sets.
2) Compare the data of the pointers.
3) Output smaller data and increase the pointer of it; If the data are the same, output it and increase both pointers.
4) Goto 2).

In contrast, the loop raking method divides the data set into many groups (in our case, 256 groups, since the length of vector register is 256), and comparing the first element in all groups at the same time. It consists of the following steps:

1) Divide the data into groups.
2) The first unused element of each group is placed in the vector register.
3) Compare the two vector registers (vectorized).
4) Goto 2).



(a) Sequential algorithm



(b) Loop Raking

Fig. 5. Comparison of sequential algorithm and loop raking.

Loop raking makes it possible to vectorize a loop that cannot be vectorized otherwise. Besides, it can be used to enlarge vector length. However, it has several drawbacks. One drawback is that memory access becomes non-contiguous. Especially, if the memory access becomes scatter or gather (e.g., access like a[b[i]]), the performance of memory access becomes non-optimal. Another drawback is the performance of the branch. If the computation of each virtual processor becomes complex, it might contain a branch. A loop that contains a branch can be vectorized, but it is implemented using a mask register. That is, the value of the condition is stored in the mask register, and the instruction is executed regardless of the condition; the result is reflected to register or memory according to the mask value. Therefore, if the condition becomes complex, many of the results end up unused.

B. ESC Algorithm

ESC algorithm [11] is proposed by Bell et al. for GPU. It consists of three phases, which are expansion, sorting, and compression. In the expansion phase, it creates sparse vectors multiplied by non-zeros of $\mathbf{A}$, as explained above. This phase is done for all the rows of $\mathbf{A}$ (and $\mathbf{C}$) in parallel. In the sorting phase, the resulting non-zeros of the sparse vectors are sorted according to the row and column indices. In the compression phase, the non-zeros that have the same row and column index are added to one non-zero value. The result becomes the matrix $\mathbf{C}$. Each phase of this algorithm has a high parallelism. As for the sorting phase, which is the most time-consuming part of this algorithm, we used radix sort based on loop raking that is proposed in [15].

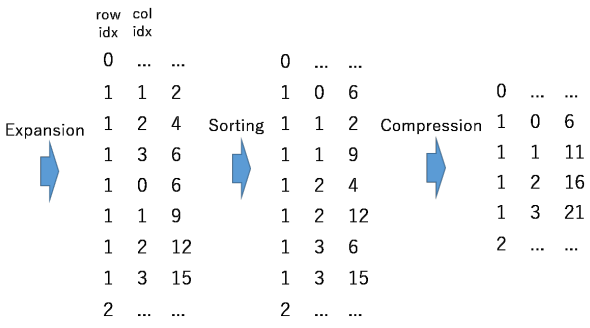


Fig. 6. Esc Algorithm

In the implementation of vector architecture, we separate the matrix A into blocks and do the steps block by block to utilize the cache (LLC), which is similar to the strategy proposed by Dalton et al. [16].

Besides, we added individual case support for matrices that have only 0 or 1 as the values (which means non-zero values are always 1). This is typical if the matrix is an adjacency matrix, and the edge weights of the graph are always 1. In this case, we can speed up the sort phase, because we only sort indices instead of pairs of index and value.

C. Hash based Algorithm

The SpGEMM algorithm in the cuSPARSE library uses the hash table for the addition of sparse vectors [12]. Where the column index becomes the key of the hash table; the value is inserted if the key is not stored in the hash table. Otherwise, the value is accumulated to the already stored value. After this process, we can get the result row by extracting the stored key

value pairs from the hash table. We can do this process for all the rows in parallel.

We used loop raking technique to implement hash based algorithm; each virtual processor (element of vector register) processes different rows. In this case, each virtual processor updates the hash table for the corresponding rows sequentially; we do not have to worry about the parallel update of the hash table. To handle the collision, if a collision occurs, the key is stored in a different array; the contents of the array is processed again by adding 1 to the hash value, which realizes open address linear probing.

The column indices of the result matrix are not sorted in the case of the hash-based algorithm. Since some algorithms assume that they are sorted, we added an option to sort them, though it decreases the performance.

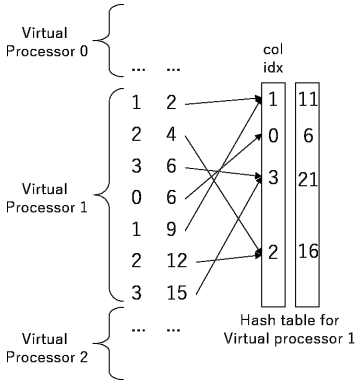


Fig. 7. Hash based Algorithm

D. Sparse Accumulator (SPA)

Sparse accumulator (or SPA) is a classic algorithm proposed by Gilbert et al. [14]. It uses dense vectors whose size is the same as the number of columns of C. The sparse vectors are added into the dense vector. There is another flag vector that contains the information if the index of the vector contains a value or not. If the corresponding index of the flag vector is false, it is set to true, and the index is saved into another vector; this vector stores the non-zero part of the dense vector after the process. By using this vector, the non-zero part can be known without scanning the flag vector or the dense vector that contains the value.

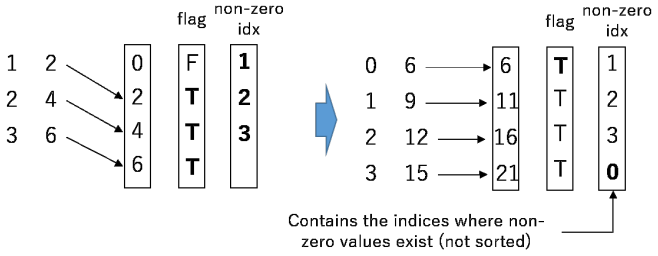


Fig. 8. SPA Algorithm

Though SPA is a quite efficient algorithm, implementations of SpGEMM for highly parallel architectures, such as GPUs, usually avoid it. This is because if multiple rows of **A** are processed in parallel, the required number of the dense vectors is the number of parallelisms, which is too large and not affordable memory size.

In the implementation of vector architecture, adding sparse vectors into the dense vector is processed in a vectorized way without loop raking technique to avoid using too many dense vectors. In this case, parallelism (which corresponds to vector length) is limited by the number of non-zeros of each sparse vector. Saving the non-zero index is done by loop raking manner; separate memory space is assigned to each virtual processor, and indices are stored independently. Like the hash-based algorithm, The column indices of the result matrix is not sorted. We also added an option to sort them.

E. Hybrid Algorithm

As described above, The parallelism of SPA is limited by the number of non-zero elements of each sparse vector. In practical applications, the number of non-zero elements per row will vary greatly, as they usually follow the power-law distribution. Therefore, we propose a novel hybrid method. It combines SPA and other methods according to the average numbers of the non-zeros of intermediate sparse vectors. For example, that of *kth* row of **A** in Figure 4 is (3 + 4) / 2 = 3.5.

First, the average number of non-zeros of each row is calculated, and the rows are sorted according to this value. Then, the matrix is divided by the user-defined threshold; the part with a higher average number of the non-zeros is processed by SPA, another part is processed by ESC or hash-based algorithm.

Liu et al. proposed a method that separates the matrix into bins and uses different algorithms for each bin [17]. It is similar to our method in that it uses multiple algorithms. However, our method is unique in that it uses an efficient SPA algorithm for the part where the average number of the non-zeros is high to enlarge vector length for vector architecture.

F. Parallelization of SpGEMM with Vector Engine

SpGEMM can be parallelized by dividing the matrix by row and assigning them to each processor. However, to get better scalability, it is essential to assign tasks evenly to the processors for load balancing. In our implementation, we count the number of non-zero of intermediate sparse vectors and divide the matrix according to the number, which achieved better load balancing.

Figure 9 illustrates this process. For the two operand matrices of SpGEMM - A and B, we applied 1D decomposition [18] by row for the left side matrix, i.e., we leave the right side matrix B untouched, but for the left side matrix A we split it into row blocks. However, to achieve better load balancing, we do not split the rows evenly but based on the number of non-zeros (denoted by a solid dot). In this example, matrix A is split into 4 slices. Although the number of rows is not the same for each slice, getting the non-zeros evenly distributed can help

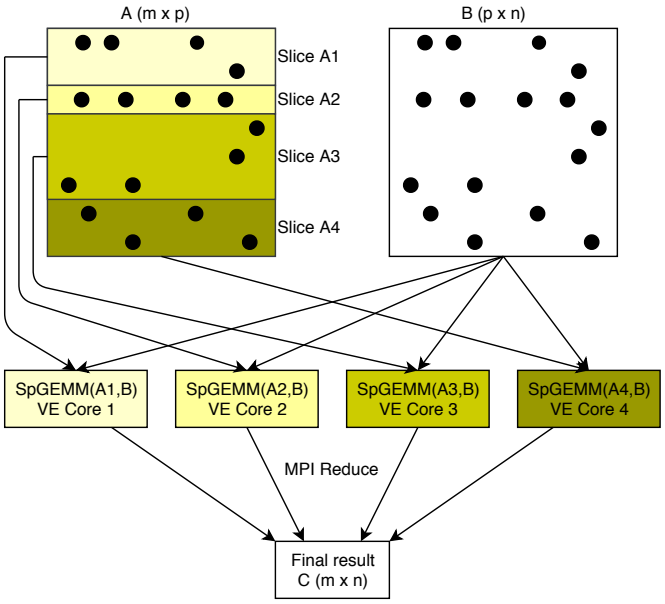


Fig. 9. Parallelizing SpGEMM using MPI with partitioned row blocks

achieve better load balance. We utilize MPI for parallelization, and each slice is assigned to a different MPI process so that the work can be conducted in parallel on the VE cores. For actual datasets, the split number could be in thousands.

G. Evaluation

We evaluated our implementation using sparse matrices from the SuiteSparse Matrix Collection [19] that are commonly used in papers like [13]. Table I shows the matrix used for evaluation.

TABLE I
MATRICES FOR EVALUATION

	nnz/row	max nnz/row	intermed. nnz
Protein	119.3	204	555,322,659
FEM/Accelerator	21.7	81	79,883,385
webbase	3.1	4700	69,524,195
cit-patents	4.4	770	82,152,992
wb-edu	5.8	3841	1,559,579,990
Circuit	5.6	353	8,676,313

The performance is evaluated by $\mathbf{A}^2$. The FLOPS is calculated as (the number of non-zero of intermediate sparse vectors) * 2 / (execution time), which is commonly used as SpGEMM evaluation. The execution time does not contain I/O but contains the counting cost for load balancing.

We measured the performance using 1, 2, 4, and 8 VEs. Since SX-Aurora TSUBASA (A300-4) contains 4 VEs per server. We used two servers for evaluation of 8 VEs, which are connected via InfiniBand. We used MPI also for parallelization within the VE (flat-MPI); in the case of 8 VEs, there are 64 ranks in total.

To compare absolute performance, we also evaluated the performance on Xeon using Intel MKL. We used 1 and 2 sockets of Xeon 6126 Gold. The API we used is `mkl_sparse_spmm` and `mkl_sparse_order`; the API `mkl_sparse_order` is for sorting the index of the result. We utilized shared memory parallelization for MKL that is provided by the library.

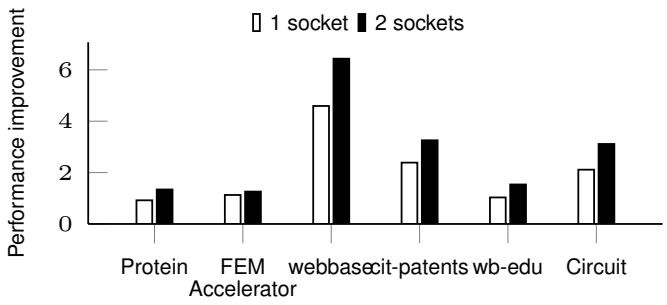


Fig. 10. Performance improvement over CPU. Calculated by $\frac{\text{NEC-Hybrid GFLOPS}}{\text{CPU GFLOPS}}$. NEC-Hybrid has an average performance improvement of 139% over CPU, with a maximum performance improvement of 6.43x.

Figure 11 shows the evaluation result. Hybrid is a hybrid of ESC and SPA method. We used the single-precision floating-point as the value and 32bit integer as the index. All the results include the sorting time of the index.

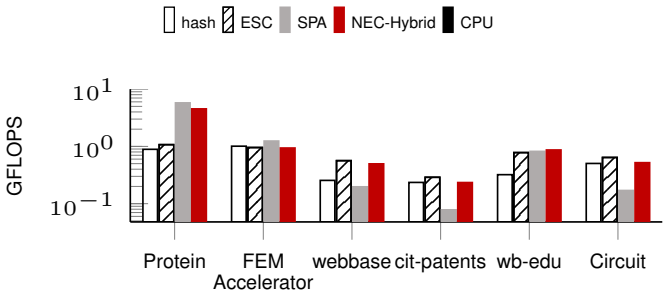
The matrices are grouped into three categories. As for Protein and FEM/Accelerator, the NNZ per row is relatively large. Therefore, SPA performs better than other methods. As for webbase and cit-patents, NNZ per row is small; Therefore, the ESC shows better performance than the SPA. For wb-edu, the network size is relatively large, and the maximum NNZ per row is much larger than average. Therefore, our hybrid method performs the best. Our hybrid method shows stable performance. Although it is not the best performer in all situations, it avoids some of the noticeable shortcomings of ESC and SPA. In all the cases, the hash-based method does not show better performance than other methods. Since it consists of a complex branch in the loop raking technique to implement the hash table; the overhead of loop raking caused poor performance. Our implementation shows better performance than CPU and also shows good scalability.

IV. Vectorization of Graph Algorithms with SpGEMM

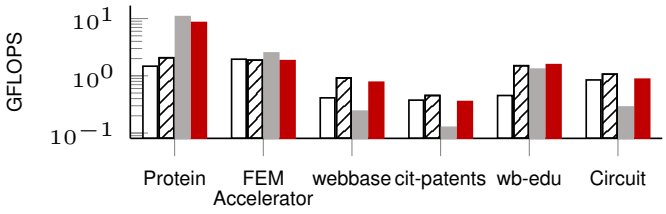
We have implemented a high-performance linear algebra kernel SpGEMM on the vector engine. In this section, we will introduce two important graph analysis algorithms: triangle counting and butterfly counting. We will use linear algebra operations, mainly SpGEMM, to implement these algorithms so they can take advantage of the NEC vector engine.

A. Triangle Counting

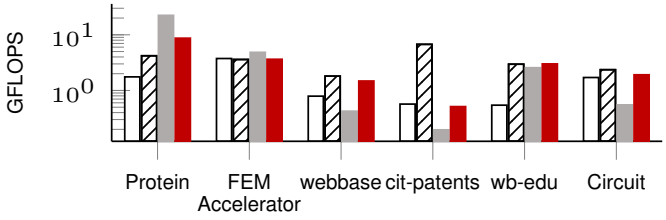
A triangle is a special type of a subgraph that is commonly used for computing important measures in a graph. The triangle counting algorithm consists of the following steps:



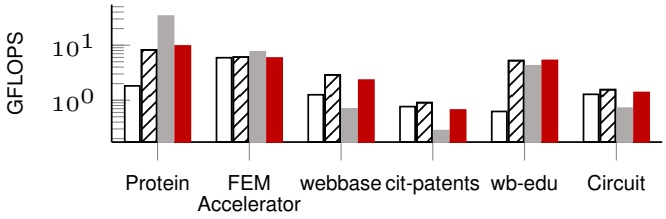
(a) 1 socket



(b) 2 sockets



(c) 4 sockets



(d) 8 sockets

Fig. 11. Evaluation of SpGEMM kernels on VE

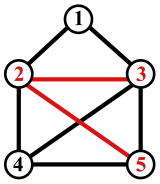
1) Split $\mathbf{A}$ *into a lower triangular* $\mathbf{L}$ *and an upper triangular* $\mathbf{U}$*:* Given an adjacency matrix $\mathbf{A}$, as shown in figure 12 (a) and (b), the algorithm splits $\mathbf{A}$ into a lower triangular and an upper triangular pieces via $\mathbf{A} = \mathbf{L} + \mathbf{U}$.

2) Calculate $\mathbf{B} = \mathbf{LU}$*:* In graph terms, the multiplication of $\mathbf{L}$ by $\mathbf{U}$ counts all the wedges of (i, j, k) form where j is the smallest numbered vertex. As shown in Figure 12 (c), only one wedge $(5, 2, 3)$ between node 5 and node 3 satisfies $2 < 5$ and $2 < 3$. Correspondingly, $\mathbf{B}_{5,3} = 1$.

3) Calculate $\mathbf{U} . * \mathbf{B}$*, the element-wise multiplication of* $\mathbf{A}$ *and* $\mathbf{B}$*:* The final step is to find if the wedges close by doing element-wise multiplication with the original matrix.

B. Butterfly Counting

Butterfly refers to a loop of length 4 in the bipartite graph. It is the simplest cohesive higher-order structure in a



(a) Input graph G. There is only one 2-path (3,2,5) between node 3 and node 5 that satisfies $2 < 3$ and $2 < 5$. Path (3,4,5) is not considered since $4 > 3$.

$$\begin{array}{l} Vertex1 \\ Vertex2 \\ Vertex3 \\ Vertex4 \\ Vertex5 \end{array} \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

(b) Adjacency matrix $\mathbf{A}$ of G. The lower triangular part $\mathbf{L}$ and the upper triangular part $\mathbf{U}$ are marked in blue and red.

$$\mathbf{B} = \mathbf{LU} = \begin{bmatrix} & & & \\ 1 & & & \\ 1 & 1 & & \\ 0 & 1 & 1 & \\ 0 & 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 0 & 0 \\ & 1 & 1 & 1 \\ & & 1 & 1 \\ & & & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 2 & 1 & \boxed{1} \\ 0 & 0 & 1 & 2 & 2 \\ 0 & 0 & \boxed{1} & 2 & 3 \end{bmatrix}_{5\times5}$$

(c) Line 3 of algorithm 1 : $\mathbf{B} = \mathbf{LU}$ using SpGEMM kernel. $\mathbf{B}_{5,3}$ and $\mathbf{B}_{3,5}$ corresponds to the number of 2-paths between node 5 and 3 satisfying $2 < 3$ and $2 < 5$.

Fig. 12. Adjacency matrix

Algorithm 1: Triangle counting

input : Graph $G = (V, E)$
output: *number of triangles in* G
1 Generate the adjacency matrix $\mathbf{A}$
2 Split $\mathbf{A}$ into a lower triangular $\mathbf{L}$ and an upper triangular $\mathbf{U}$
3 $\mathbf{B} = \mathbf{LU}$ `// SpGEMM Kernel`
4 $\mathbf{C} = \mathbf{U} . * \mathbf{B}$ `// Element wise multiplication`
5 **return** $\sum \mathbf{C}_{i,j}$

bipartite graph. [3] presented exact algorithms for butterfly counting, as shown in algorithm 2, which can be considered as state-of-the-art. Although this algorithm is fast, it is a loop-based, sequential algorithm. To achieve parallelization and vectorization, we have improved algorithm 2 to algorithm 3 which fully utilize the linear algebra kernels. Our Butterfly counting algorithm consists of the following steps:

1) Create the adjacency matrix $\mathbf{A}$*:* Let $G = (V = (L, R), E)$ be a bipartite graph, where L and R are its left and right parts, respectively. Suppose $L = \{L_1, L_2, \ldots, L_m\}$, $R = \{R_1, R_2, \ldots, R_n\}$. Then we can represent the adjacency matrix of G as $\mathbf{A}_{m\times n}$. $\mathbf{A}_{i,j} = 1$ if and only if L_i and R_j is connected, otherwise, $\mathbf{A}_{i,j} = 0$. In this case, $\mathbf{A}$ is sometimes called the biadjacency matrix.

2) Calculate $\mathbf{AA}^\intercal$*:* According to the properties of the adjacency matrix[20], we have: If $\mathbf{B} = \mathbf{AA}^\intercal$, then the matrix $B_{i,j}$ gives the number of walks of length 2 from vertex L_i to vertex L_j. As shown in Figure 13, there are three paths of length 2 between L_1 and L_2, which are marked in three colors: red, blue, and green.

3) Set the element on the diagonal of matrix $\mathbf{B}$ *to 0 and add up:* Since each butterfly is made up of two paths of length 2, the two paths share endpoints in L (or R). Thus, if there are $B_{i,j}$ paths between L_i and L_j, then the number of butterflies

Algorithm 2: ExactBFC: Sequential Butterfly Counting

input : Graph $G = (V = (L, R), E)$
output: $Butterfly(G)$
1 **if** $\sum_{u \in L}(d_u)^2 < \sum_{v \in R}(d_v)^2$ **then**
2 $A \leftarrow R$
3 **else**
4 $A \leftarrow L$
5 **for** $v \in A$ **do**
6 $C \leftarrow hashmap$
7 **for** $u \in Neighbour(v)$ **do**
8 **for** $w \in Neighbour(u)$ **do**
9 $C[w] \leftarrow C[w] + 1$
10 **for** $w \in C$ **do**
11 $Butterfly(G) \leftarrow Butterfly(G) + \binom{C[w]}{2}$
12 **return** $Butterfly(G)/2$

Algorithm 3: Vectorized butterfly counting

input : Graph $G = (V = (L, R), E)$
output: *number of butterflies in* G
1 Generate the adjacency matrix $\mathbf{A}$
2 $\mathbf{B} \leftarrow \mathbf{A}\mathbf{A}^\intercal$ `// SpGEMM kernel`
3 Set the element on the diagonal of matrix $\mathbf{B}$ to 0.
4 **return** $\frac{1}{2}\sum_{i,j}\binom{\mathbf{B}_{i,j}}{2}$

with L_i, L_j as the endpoint is $\binom{B_{i,j}}{2}$. Note that we only count 2-paths that differ from the start and endpoints, so we set the elements on the diagonal of matrix B to 0 to exclude those paths where the start and endpoints are the same. For example, the matrix in Figure 13 is transformed into the following form

$$\mathbf{B} = \begin{bmatrix} 3 & 3 & 1 \\ 3 & 4 & 2 \\ 1 & 2 & 2 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & \binom{3}{2} & \binom{1}{2} \\ \binom{3}{2} & 0 & \binom{2}{2} \\ \binom{1}{2} & \binom{2}{2} & 0 \end{bmatrix} = \begin{bmatrix} 0 & 3 & 0 \\ 3 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}_{3\times 3}$$

Finally, we calculate $\frac{1}{2}\sum_{i,j}\binom{B_{i,j}}{2}$ to get the exact number of "butterflies".

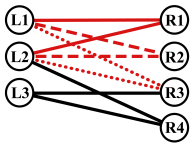
V. EXPERIMENTS ON SUBGRAPH COUNTING AND MACHINE LEARNING ALGORITHMS

A. System Architecture and Implementation

In the experiments, we use: 1) a single node of dual-socket Intel(R) Xeon(R) Gold 6126 (architecture Skylake), 2) a single node of a dual-socket Intel(R) Xeon(R) CPU E5-2670 v3 (architecture Haswell), 3) a single NEC Aurora node with 4 VEs. More details of the testbed hardware can be seen in Table II.

B. Execution Time Breakdown

We have applied the NEC SpGEMM algebra kernels in our triangle counting and butterfly counting implementation. We instrumented the code to show the normalized time spent breakdown on different portions of the code with different



(a) Input bipartite graph and 2-paths between L_1 and L_2.

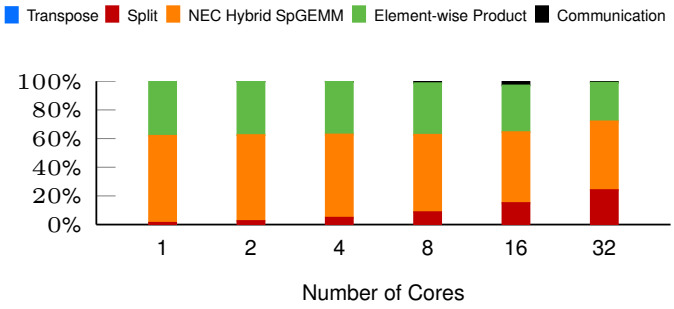
$$\begin{array}{c} \\ L_1 \\ L_2 \\ L_3 \end{array} \begin{array}{c} \begin{array}{cccc} R_1 & R_2 & R_3 & R_4 \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix} \end{array}$$

(b) Biadjacency matrix $\mathbf{A}$ of input graph.

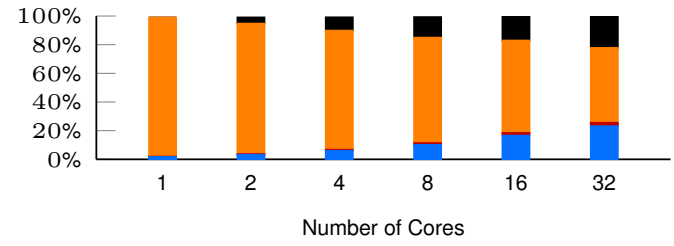
$$\mathbf{B} = \mathbf{A}\mathbf{A}^\intercal = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 3 & \boxed{3} & 1 \\ 3 & 4 & 2 \\ 1 & 2 & 2 \end{bmatrix}_{3\times 3}$$

(c) Line 2 of algorithm 3: $\mathbf{B} = \mathbf{A}\mathbf{A}^\intercal$. $\mathbf{B}_{1,2}$ and $\mathbf{B}_{2,1}$ represents the number of 2-paths between L_1 and L_2.

Fig. 13. Adjacency matrix of bipartite graph



(a) Triangle counting



(b) Butterfly counting

Fig. 14. Normalized execution time breakdown.

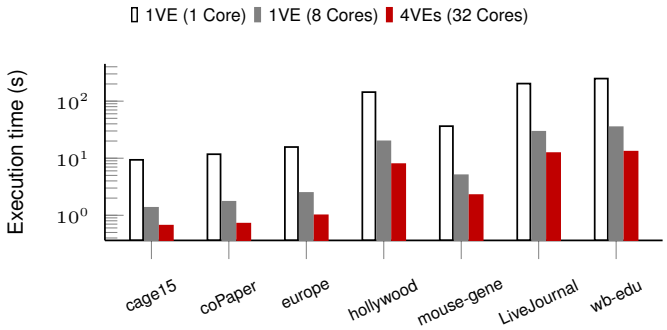
datasets. The results are shown in Figure 14 for both triangle counting and butterfly counting. The fact that the two computation kernels (SpGEMM and Element-wise Product) consume the majority of the execution time was the motivation why we have implemented and optimized the algebra kernels on the NEC Aurora platform. We can see from the figure that with the increasing number of cores used, the two kernels are consuming less proportional time which suggests that the parallel execution of the kernels has decreased the overall execution time. On another hand, the proportion of the time spent on non-parallelized code, which includes the preparation of the data and other overhead introduced with handling multi-processes (split, transpose operations, and MPI communication and synchronizations) is increasing.

TABLE II
HARDWARE SPECIFICATIONS AND DATASETS USED

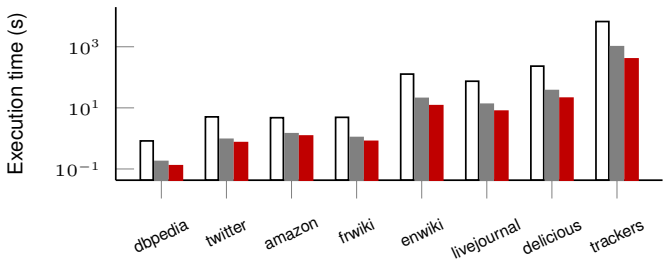
Arch	Model	frequency (GHz)	physical cores	Vector register width (bits)	Vector register	Peak Performance	memory bandwidth (GB/s)	memory capacity (GB)	L1 cache (KB)	L2 cache (MB)	L3 cache (MB)
CPU Skylake	Xeon Gold 6126	2.6	12	512	2	998GF(SP)	119	125	768	12	19.25
CPU Haswell	E5-2670 v3	2.3	12	256	1x24	883GF(SP) 441GF(DP)	95	125	768	3	30
Vector Engine	SX-Aurora TSUB-ASA	1.4	8	16384	256	4.91TF(SP) 2.45TF(DP)	1200	48	32x8	2	16

TABLE III
GRAPH DATASETS USED IN THE EXPERIMENTS

Data	Vertices	Edges	Avg Deg
mouse-gene	29.0M	28.9M	2.00
coPaperDBLP	540k	30.5M	112.83
soc-LiveJournal1	4.8M	85.7M	35.36
wb-edu	9.8M	92.4M	18.78
cage15	5.2M	94.0M	36.49
europe-osm	50.9M	109.1M	4.25
hollywood-2009	1.1M	112.8M	197.83
DBPedia-Location	172K+53K	293K	1.30
Wiki-fr	288K+3.9M	22M	5.25
Twitter	175K+530K	1.8M	2.55
Amazon	2.1M+1.2M	5.7M	1.73
Journal	3.2M+7.4M	112M	10.57
Wiki-en	3.8M+21.4M	122M	4.84
Deli	833K+33.7M	101M	2.92
web-trackers	27.6M+12.7M	140M	3.47



(a) Triangle counting



(b) Butterfly counting

Fig. 15. Execution Time of Triangle counting and Butterfly counting

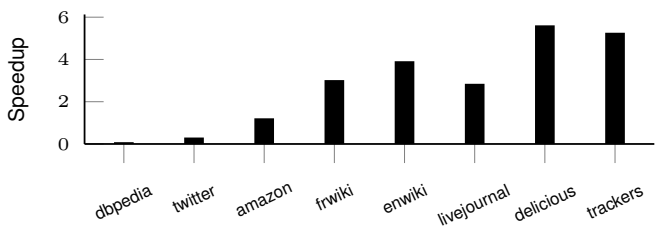


Fig. 16. Butterfly counting speedup (1VE vs BFC_Exact on CPU)

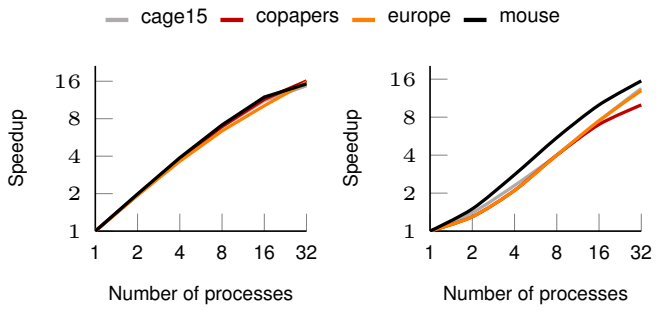
C. Execution Time Evaluation

We used the datasets in Table III to evaluate the performance on a different number of processes utilizing the up to 32 cores of the 4 VEs on the single NEC Aurora node. For butterfly counting, we also compared the execution with the reference BFC_exact [3] running on Haswell CPU. Figure 15 shows the execution time of triangle counting and butterfly counting for the different datasets on single VE (1 core and 8 cores) and 4 VEs (32 cores being utilized). Figure 16 shows the speedup of our algorithm while using one single VE comparing to the BFC_Exact result (normalized to 1). We can see from this figure that for the larger datasets, single VE (with all 8 cores being utilized) achieved better performance with a factor of 3-5 comparing to the BFC_Exact algorithm running on Haswell CPU.

D. Strong Scaling Evaluation

The execution time breakdown shown in Figure 14 has already suggested that the algebra kernels helped achieve good strong scaling. Here we are showing this more clearly in Figure 17 and Figure 18. Figure 17 shows the speedup when utilizing multiple VE cores comparing to one core on the left side; and on the right side, it shows the results from the work in [2]. For the scaling on VE most datasets showing close to linear scalability till the number of processes reached over 10, which is better than the right side chart. Figure 18 shows similar results for Butterfly counting, although it decreases after 8 processes when comparing to the Triangle counting

results.



(a) VE (b) CPU

Fig. 17. Triangle counting scalability test on single process

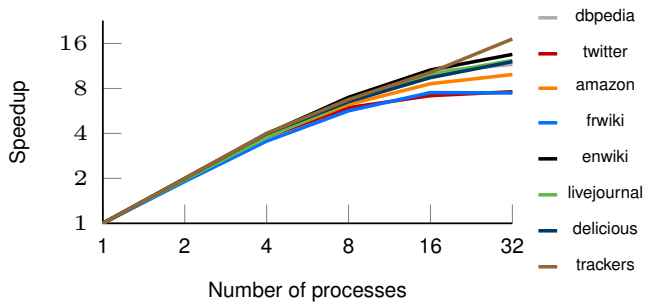


Fig. 18. Butterfly counting scalability with increasing number of processes

E. SpGEMM Kernel Algorithms and Parameter Options

The SpGEMM kernel provided multiple algorithms to choose from, among HASH, SPA, SPA_SORT, ESC, and hybrid of any two of these. For the hybrid algorithm, a user can change the default threshold parameters as part of the function call. The evaluation we have done was using the hybrid algorithm of ESC and HPA/HPA_SORT with default parameters. In this section, we are showing the results of alternative algorithms and parameters. We used triangle counting as an example on three different datasets to show the impact of the algorithms and parameters options. For the algorithms, We tested the ESC alone, SPA_SORT, and the hybrid of the two; For parameters choice of the hybrid mode, a user can specify 3 parameters N1, N2, N, meaning number of columns to process at a time for the first separated matrix, the number of columns to process for the second matrix, and the threshold to separate the two matrices, respectively. The default values for these 3 parameters are 256, 4096, and 64, in that order. We used default values, the increased values by timing 4, and the decreased values by dividing 4 from the default parameters. The results are shown in Figure reffig:spgemm-options. The columns are the SpGEMM breakdown time from the overall execution time of running triangle counting on the testing datasets. The left-most column for each dataset is the hybrid algorithm with default parameters setting. We can see from figure 19 that while this may not be the best-performed choice among all, the hybrid mode with default parameters can produce balanced and good enough results without the need to get more details of the dataset beforehand. As mentioned before in the SpGEMM kernel evaluation section, knowing the characteristics of the dataset, e.g., the distribution of the NNZ per row, could help choose the best algorithm/parameter options, but our experiments were done with hybrid mode and default parameters already show good results. For the SPA and SPA_SORT options, the difference is that with SPA_SORT the column indices of the result matrix are sorted while for SPA, they are not. While sorting added more overhead compared to the non-sorted version, this feature is especially useful if the result matrix is to be chained as input to another algorithm that assumes the column indices are sorted. E.g., in our triangle counting application, we have an element-wise multiplication step after SpGEMM, which utilized another kernel that requires the input matrices to be sorted with column indices. Thus we would need to use the hybrid algorithm of ESC and SPA_SORT. While for butterfly counting, the order of the column indicators does not affect the counting result. Thus we use the hybrid algorithms of ESC and SPA to get better performance.

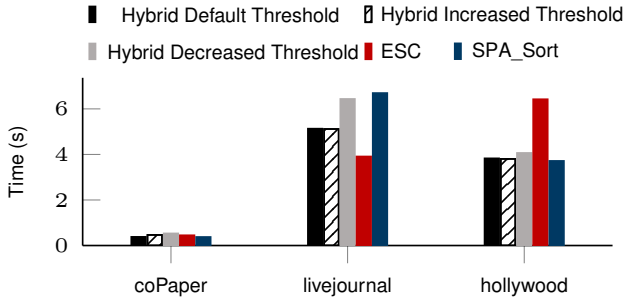


Fig. 19. SpGEMM Time from Triangle Counting Execution Breakdown using Different SpGEMM Algorithms and Parameter Options

VI. CONCLUSION

In this paper, we have introduced a new vectorized SpGEMM algorithm for butterfly counting in bipartite graphs and also adapted another vectorized triangle counting algorithm, on the NEC Aurora platform. The algorithms are all vectorized, makes it very suitable to run on the vector engine of the NEC Aurora system.

In the SpGEMM kernel evaluation (section III-G), NEC-Hybrid has an average performance improvement of 139% over CPU, with a maximum performance improvement of 6.43x. In the test of the triangle counting algorithm, our implementation shows high scalability compared to related work [2]. In the test of the butterfly counting algorithm, our implementation on large datasets has achieved up to 6 times faster performance even with a single VE, and more than 10 times faster when multiple VEs are used from one node. With the optimized linear algebra kernels such as SpGEMM, both

Graph algorithms demonstrate good performance and scalability. This work can be extended to support other applications and architecture-specific code optimizations in future work.

VII. Acknowledgment

We gratefully acknowledge the support from NEC Corp., NSF CIF21 DIBBS 1443054: Middleware and High Performance Analytics Libraries for Scalable Data Science, Science and NSF EEC 1720625: Network for Computational Nanotechnology (NCN) Engineered nanoBio node, NSF OAC 1835631 CINES: A Scalable Cyberinfrastructure for Sustained Innovation in Network Engineering and Science, and Intel Parallel Computing Center (IPCC) grants. We would like to express our special appreciation to the FutureSystems team.

References

[1] "Nec sx-aurora tsubasa - vector engine." https://www.nec.com/en/global/solutions/hpc/sx/vector_engine.html. Accessed: 2019-09-05.

[2] A. Azad, A. Buluç, and J. Gilbert, "Parallel triangle counting and enumeration using matrix algebra," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW)*, pp. 804–811, IEEE, 2015.

[3] S.-V. Sanei-Mehri, A. E. Sariyuce, and S. Tirthapura, "Butterfly counting in bipartite networks," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 2150–2159, ACM, 2018.

[4] "Spgemm on nec vector engine." https://github.com/dsc-nec/frovedis_matrix. Accessed: 2019-09-05.

[5] N. Alon, P. Dao, I. Hajirasouliha, F. Hormozdiari, and S. C. Sahinalp, "Biomolecular network motif counting and discovery by color coding," *Bioinformatics*, vol. 24, no. 13, pp. i241–i249, 2008.

[6] F. Battiston, V. Nicosia, M. Chavez, and V. Latora, "Multilayer motif analysis of brain networks," *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 27, no. 4, p. 047404, 2017.

[7] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra*. SIAM, 2011.

[8] L. Chen, J. Li, A. Azad, L. Jiang, M. Marathe, A. Vullikanti, A. Nikolaev, E. Smirnov, R. Israfilov, and J. Qiu, "A graphblas approach for subgraph counting," *arXiv preprint arXiv:1903.04395*, 2019.

[9] I. V. Afanasyev, V. V. Voevodin, V. V. Voevodin, K. Komatsu, and H. Kobayashi, "Analysis of relationship between simd-processing features used in nvidia gpus and nec sx-aurora tsubasa vector processors," in *International Conference on Parallel Computing Technologies*, pp. 125–139, Springer, 2019.

[10] K. Komatsu, S. Momose, Y. Isobe, O. Watanabe, A. Musa, M. Yokokawa, T. Aoyama, M. Sato, and H. Kobayashi, "Performance evaluation of a vector supercomputer sx-aurora tsubasa," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 685–696, IEEE, 2018.

[11] N. Bell, S. Dalton, and L. N. Olson, "Exposing fine-grained parallelism in algebraic multigrid methods," *SIAM J. Sci. Comput.*, vol. 34, no. 4, pp. C123–C152, 2012.

[12] J. Demouth, "Sparse matrix-matrix multiplication on the gpu," in *Proceedings of the GPU Technology Conference*, 2012.

[13] Y. Nagasaka, A. Nukada, and S. Matsuoka, "High-performance and memory-saving sparse general matrix-matrix multiplication for nvidia pascal gpu," in *Proceedings of 46th International Conference on Parallel Processing (ICPP)*, 2017.

[14] J. R. Gilbert, C. Moler, and R. Schreiber, "Sparse matrices in MATLAB: Design and implementation," *SIAM J. Matrix Anal. Appl.*, vol. 13, no. 1, pp. 333–356, 1992.

[15] M. Zagha and G. E. Blelloch, "Radix sort for vector multiprocessors," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pp. 712–721, ACM, 1991.

[16] S. Dalton, L. Olson, and N. Bell, "Optimizing sparse matrix—matrix multiplication for the gpu," *ACM Trans. Math. Softw.*, vol. 41, pp. 25:1–25:20, Oct. 2015.

[17] W. Liu and B. Vinter, "A framework for general sparse matrix-matrix multiplication on gpus and heterogeneous processors," *J. Parallel Distrib. Comput.*, vol. 85, pp. 47–61, Nov. 2015.

[18] A. Buluç and J. R. Gilbert, "Highly parallel sparse matrix-matrix multiplication," *arXiv preprint arXiv:1006.2183*, 2010.

[19] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, pp. 1:1–1:25, Dec. 2011.

[20] R. P. Stanley, "Algebraic combinatorics," *Springer*, vol. 20, p. 22, 2013.

A Distributed Deep Memory Hierarchy System for Content-based Image Retrieval of Big Whole Slide Image Datasets

Esma Yildirim
Department of Mathematics and Computer Science
Queensborough Community College of CUNY
Bayside, NY
eyildirim@qcc.cuny.edu

Shaohua Duan
Rutgers Discovery Informatics Institute
Rutgers University
Piscataway, NJ
shaohua.duan@rutgers.edu

Xin Qi
Rutgers Cancer Institute of New Jersey
Rutgers University
New Brunswick, NJ
qixi@cinj.rutgers.edu

***Abstract*—Whole slide images (WSIs) are very large (30-50GB each in uncompressed format), multiple resolution tissue images produced by digital slide scanners, and are widely used by pathology departments for diagnostic, educational and research purposes. Content-based Image Retrieval (CBIR) applications allow pathologists to perform a sub-region search on WSIs to automatically identify image patterns that are consistent with a given query patch containing cancerous tissue patterns. The results can then be used to draw comparisons among patient samples in order to make informed decisions regarding likely prognoses and most appropriate treatment regimens, leading to new discoveries in precision and preventive medicine.**

CBIR applications often require repeated, random or sequential access to WSIs, and most of the time the images are preprocessed into smaller tiles, as it is infeasible to bring the entire WSI into the memory of a computer node. In this study, we have designed and implemented a distributed deep memory hierarchy data staging system that leverages Solid-State Drives (SSDs) and provides an illusion of a very large memory space that can accommodate big WSI datasets and prevent subsequent accesses to the file system. An I/O intensive sequential CBIR workflow for searching cancerous patterns in prostate carcinoma datasets was parallelized and the I/O paths were altered to include the proposed memory system. Our results indicate that the parallel performance of the CBIR workflow improves and our deep memory hierarchy, staging framework produces negligible overheads for the application performance even when the number of staging servers and their memory sizes are limited.

***Index Terms*—whole slide images, distributed data staging, big data analytics, content-based image retrieval**

I. Introduction

Content-based Image Retrieval workflows require expensive computational operations such as low-level transformations, object segmentation, feature computation, filtering and subsampling [1], [2]. In most studies [1], [2], [3], [4], [5], the workflow starts with the preprocessing of WSI data into smaller tiles so that they can fit into the available memory on the system, and the computation is easily parallelized by distributing these tiles. The very few parallel data access methods, do not scale well [6] or introduce additional overhead [1], [5] . In a recent study [7], we have developed efficient, highly scalable parallel and distributed data access methods for WSI datasets, however these methods are still impaired by disk I/O performance because the WSI data is very large and can only be accessed from the file system.

DataSpaces [8] is a distributed data staging framework designed for scientific workflows. Data, represented in a multi-dimensional tensor format, is indexed and distributed among the memories of distributed data staging servers in a cluster of computers and is accessed using simple *get()* and *put()* operations, given the partial coordinate plane information with respect to the dimension sizes in the tensor data object. One downside of this approach is that the memory space of each node is limited and it would require a large number of data staging nodes to accommodate the entire WSI dataset. Such a system could benefit from deep memory hierarchies that include SSDs, which are faster than disk systems. SSD performance has improved even more with the introduction of NVM-e technology. While SATA SSDs perform much better than hard-disk drives(HDDs), NVMe SSDs present stellar performances in terms of bandwidth, I/O operations per second and latency surpassing the performance of SATA SSDs [9], [10]. Current HPC systems are turning into installing SSDs as node-local storage to improve application I/O performance rather than having none at all or depend on a shared parallel disk system.

In this study, we design and implement the necessary constructs and algorithms for a distributed deep memory hierarchy staging system that leverages NVMe SSDs to improve the I/O performance of content-based image retrieval workflows, even in the presence of limited memory space. Our proposed architecture is coded as modules into DataSpaces. As a case study, we have parallelized a sequential CBIR workflow that searches for glandular cancerous structures in prostate carcinoma datasets, using MPI [11] and OpenMP [12]. The experimental results show that the hierarchical memory system introduces only slight increase in data read performances

compared to the pure memory scenario and it performs very well compared to disk access scenario.

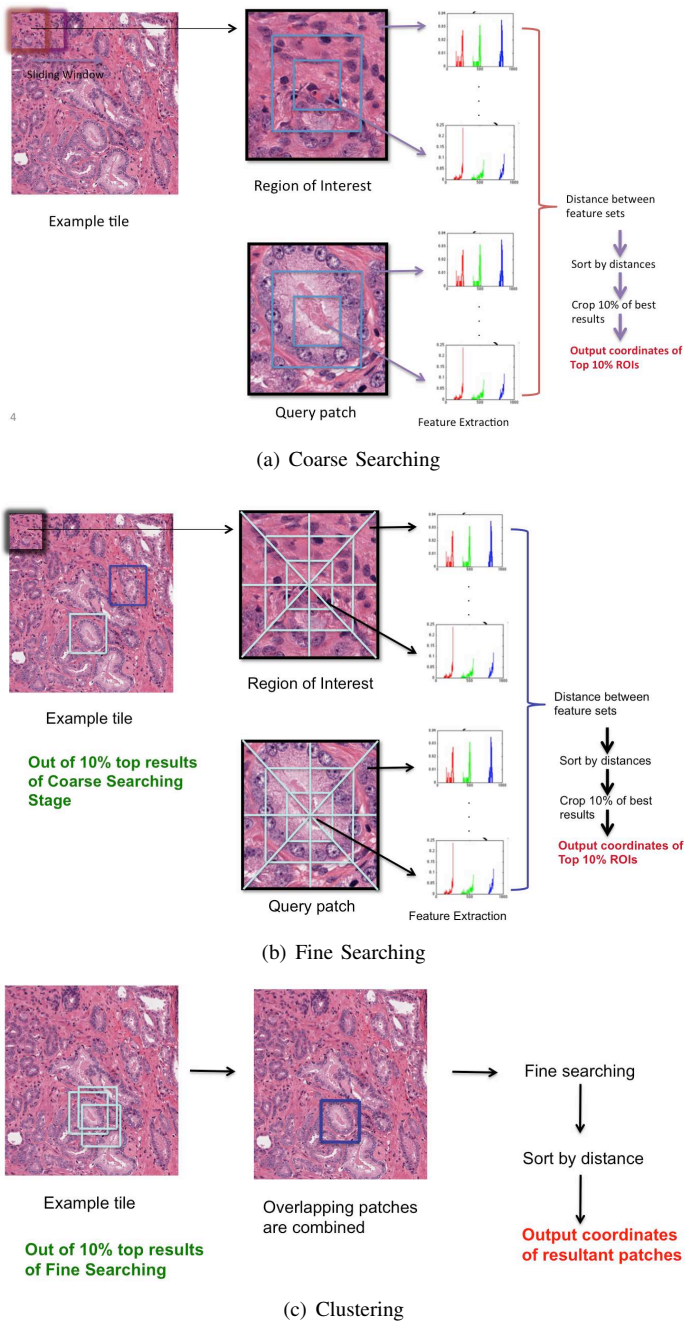
II. Case Study: A Prostate Carcinoma CBIR Workflow

The example CBIR workflow is based on the feature extraction and clustering stages of the work of Qi et al. [2]. We first parallelized the stages using MPI and OpenMP and then altered the I/O paths to integrate DataSpaces into the workflow.

The CBIR workflow consists of multiple feature extraction and clustering stages to find the coordinates of areas (Regions Of Interest (ROIs)) in a WSI dataset that show similar characteristics to the query patch at hand. The workflow starts with the Coarse Searching stage(Fig.1.a). In this stage, a sliding window approach is used to compare the query patch image to the window (a.k.a ROI) in the WSI tile. The size of the sliding window is equal to the size of the query patch and also each tile is a cropped piece from a much larger WSI. For each comparison pair (query patch, ROI), the pair of images selected are divided into rectangular rings. As a result of a feature extraction operation(e.g. colour histograms, texture), the distance between the feature vectors are calculated and sorted in ascending order. Then top 10% of the ROI coordinates with best distance results are selected and fed into a Fine Searching stage (Fig.1.b), where accesses to the WSI data is random, because the best coordinate results can come from anywhere from the dataset. This percentage is variable parameter and can be decided by the user. In the Fine Searching stage, the comparison pair is also divided into segments in addition to the rectangular rings. After a Clustering (Fig.1.c) operation, in which overlapping ROI coordinates are combined on the top 10% of the results of the Fine Searching stage, the distance values of the clustered coordinates are re-calculated, which again requires a Final Fine Searching stage.

The parallelisation process starts with constructing a list of tile coordinates with given width and height (e.g.1024x1024 pixels) values from the entire WSI dataset. The coordinates of the tiles are then shared among multiple MPI processes. Each process accesses the parallel file system (GPFS) and reads the tiles assigned to itself. Then another level of parallelism is applied once the tile is in the memory. The feature calculations of rectangular rings are shared among OpenMP threads. The highest possible level of parallelism that can be achieved in Coarse Searching and Fine Searching Stages is # of processes x # threads, which corresponds to the number of computer nodes of the cluster and number of cores per node respectively. In the Clustering stage, only MPI processes are used to share tiles due to the restrictions on the clustering algorithm.

We have redesigned the I/O paths of the workflow, so that only in the Coarse Searching stage, the WSI dataset is read from the parallel file system by the application and then it is written into DataSpaces using *dspaces_put()* interface. It is accessed from there using *dspaces_get()* interface by the subsequent stages of the application (Fig. 2). Therefore the performance of Fine Searching and the Final Fine Searching



(a) Coarse Searching

(b) Fine Searching

(c) Clustering

Fig. 1: CBIR workflow stages.

stages improves dramatically because memory access is much faster than disk access.

III. System Architecture

One downside of distributed data staging is that the memory space of each DataSpaces server can be limited and it would require a large number of data staging servers to accommodate if the WSI dataset is very big. Such a system could benefit from deep memory hierarchies that include SSDs, which are faster than disk systems. The proposed framework consists of

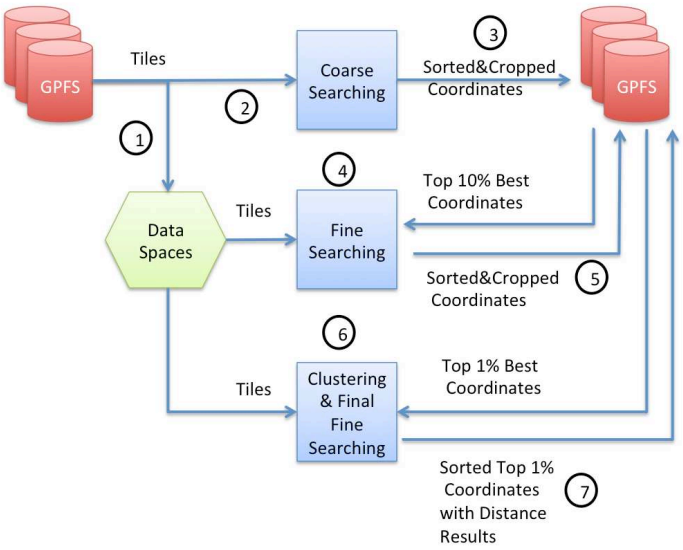


Fig. 2: CBIR I/O Path with DataSpaces

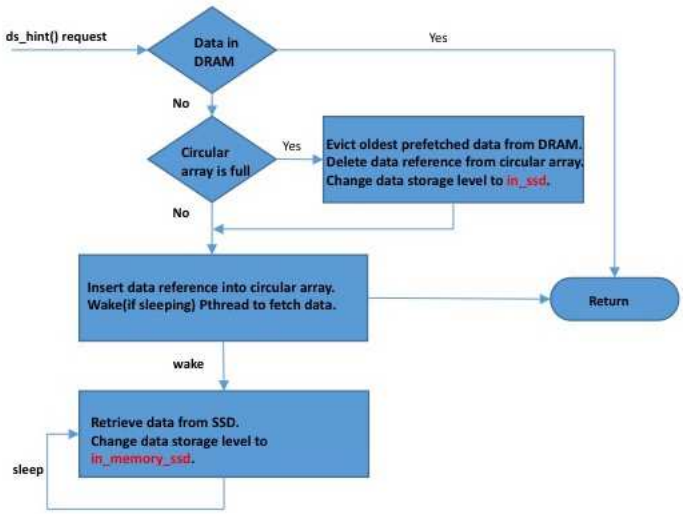


Fig. 3: Flowchart of a *ds_hint()* call

Persistence, *Caching*, and *Prefetching* modules that are coded into the DataSpaces architecture.

A. Persistence Module

The *Persistence Module* provides a light-weight approach for exploiting SSD as a secondary memory partition so that Caching and Prefetching Modules can explicitly allocate memory regions therein and read/write data. The POSIX mmap() interface offers a viable way to map files in SSD giving the illusion that they reside in memory. We build management structures for the efficient use of node-local SSD memory of each DataSpaces server. The module creates one big mapped address space during initialisation then, it manages allocation/deallocation and fragmentation of space through doubly linked list data structures.

B. Caching Module

The *Caching Module* implements a Least Recently Used(LRU) algorithm to keep recently requested data in DRAM memory and evict less recently used data back to SSD. Data could be located in DRAM memory, SSD or both, therefore we mark data storage status with three types: *In_memory*, *In_ssd*, and *In_memory_ssd*.

C. Prefetching Module

The *Prefetching Module* presents an illusion of infinite memory by making data available in DRAM memory before the application uses it, thereby masking the latency of the slower SSD memory. We offer a new user interface *dspaces_hint()* for the user application. By using this interface, the user can hint the system of the upcoming data coordinate planes the application would like to access, hence the *Prefetching Module* can bring data from SSD beforehand. A circular array data structure is used to keep the prefetching requests. Unlike the previous modules, the Prefetching module operates as a concurrent thread under the main DataSpaces server thread to maximise the performance by pipelining the prefetching I/O operations between SSD and DRAM memory and the I/O operations happening between client application and DataSpaces server. Figure 3 presents the actions performed by the prefetch module after a call to *ds_hint()* is made.

IV. Interfaces

The framework offers the following updated and newly introduced interfaces:

- **dspaces_init()** calls persistence module to create the mapped file in SSD and launches the Prefetching thread.
- **dspaces_get()** queries DataSpaces server to retrieve data. If the data for the requested coordinates is in DRAM memory, the server returns it to the client. If it is in SSD, it caches it into DRAM and changes data storage status into *In_memory_ssd*. If DRAM has no space, then it calls the cache replacement algorithm in the *Caching Module* to evict some data into SSD before bringing requested data into DRAM.
- **dspaces_put()** inserts data in DRAM and changes data storage status into *In_memory*. If DRAM has no space then it calls the cache replacement algorithm before inserting data into DRAM.
- **dspaces_hint()** queries DataSpaces servers to check if data is in SSD. If so, the hint is inserted into prefetching circular array. It wakes prefetching thread if the thread is sleeping to fetch data from SSD to DRAM. After the operation is complete data storage status is changed into *In_memory_ssd*.

Listing 1 presents a code snippet on how to use *dspaces_hint()* function in the Final Fine Searching stage. The coordinate information regarding the current tile is stored in *vectors[i]*. Before loading that tile into memory, the program

hints about the tile that will be accessed next and the information about the next tile is stored in *vectors[i+1]*. The *lb* and *ub* variables contain the dimension information while the *hint_name* is used to refer to a data object. After adding an entry to the circular prefetching array structure *dspaces_hint()* function returns immediately.

```
1
2 sprintf(hint_name,"%s-%lld-%lld",vectors[i+1][0].
    file_name,vectors[i+1][0].tile_x,vectors[i
    +1][0].tile_y);
3 uint64_t ub[2],lb[2];
4 lb[0] = 0;
5 ub[0] = lb[0] +  vectors[i+1][0].tile_width *
    vectors[i+1][0].tile_height - 1;
6 lb[1] = ub[1] = 0;
7 dspaces_hint(hint_name, 0, sizeof(uint32_t), ndim,
    lb, ub, buffer);
```

Listing 1: Code snippet for use of *dspaces_hint()*

V. Experimental Results

In this section, we present the performance results of the proposed deep memory hierarchy system on the I/O operations of the CBIR workflow. The input dataset consists of 100 WSI images selected from the Cancer Genome Atlas [13] Prostate Carcinoma dataset in the second resolution level (48GB). A 500x500 pixel query patch with cancerous tissue patterns is searched in the dataset. The experimental testbed is a cluster residing at Rutgers Discovery Informatics Institute, which consist of 128 24-core nodes with 256GB of memory connected via FDR Infiniband. In this cluster, only a few nodes were installed with the state-of-the-art NVM-e based SSDs and we had limited access to them. Therefore, we were able to use at most 4 NVM-e nodes to test our system.

Fig.4.a and b present the read performance of the fine searching and final fine searching stages in which the data is randomly accessed from 2 DataSpaces servers using pure DRAM memory and 24G/16G/8G total DRAM + SSD memory settings and they are compared to the pure disk access version. The number of cores represents the parallelism level of the CBIR workflow and is ranged between 24-192. The first observation is that the staging approach performs much better than the parallel file system(disk) access. The effect of introducing a hierarchical memory system produces negligible overhead and this overhead slightly increases as we decrease the DRAM memory size limit from 24GB to 8GB. This result proves our claim that using SSD is a viable solution when the memory size per staging node is limited and the number of staging nodes are scarce for applications that stages big data.

The total execution time of the workflow (Fig.4.c) also improves even though an extra stage of writing to DataSpaces is introduced compared with the disk version results. The CBIR application is a compute-intensive application: a very large percentage of total execution time is spent on computation rather than I/O. Therefore, although significant I/O improvements were achieved with our system (Fig.4.a and b) compared with the pure disk version, that showed itself have a little impact on the total execution time and the difference between the pure DRAM and DRAM+SSD performances seems negligible.

In another setting, we increased the number of DataSpaces servers from 2 to 4 (Fig. 5) but kept the total DRAM sizes equal. We observed that it only helped improve the performance when the number of the parallel client processes were very high (e.g. 192). For 24 core results which are not presented here, increasing the number of servers negatively impacted the performance. This indicates that increasing the number of staging servers is good only if the client application I/O access load is very high. Otherwise the indexing and extra I/O messaging of the staging system only causes performance degradation.

Another observation was that although we doubled the number of servers, it did not result in twice the speed up in most cases. The reason for that is although we increased the number of servers, we kept the total DRAM size equal in both 2 server and 4 server settings. While an 8GB DRAM refers to 4GB per server in 2-server setting, it refers to 2G per server in 4-server setting. In Figure 6, the difference between the performances of 2-server and 4-server settings is much more significant when both settings have a 4G/server DRAM size.

VI. Conclusion

The proposed deep memory hierarchy system to stage big WSI data reduced the execution time of a content-based image retrieval workflow while only introducing very little overhead. Combining SSDs with DRAM provided the illusion of a very large data staging space even in the presence of limited DRAM memory, and performed very well compared to the scenario where data was purely accessed from the filesystem.

Acknowledgment

The results published or shown here are in whole or part based upon data generated by the TCGA Research Network: http://cancergenome.nih.gov/. This research was funded, in part, by grants from the National Institutes of Health through contract 5R01CA156386-10 and 7R01CA161375-06 from the National Cancer Institute; and contract 4R01LM009239-08 from the National Library of Medicine.

References

[1] G. Teodoro, T. Kurc, J. Kong, L. Cooper, and J. Saltz, "Comparative performance analysis of intel (r) xeon phi (tm), gpu, and cpu: a case study from microscopy image analysis," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014, pp. 1063–1072.

[2] X. Qi, D. Wang, I. Rodero, J. Diaz-Montes, R. H. Gensure, F. Xing, H. Zhong, L. Goodell, M. Parashar, D. J. Foran *et al.*, "Content-based histopathology image retrieval using cometcloud," *BMC bioinformatics*, vol. 15, no. 1, p. 287, 2014.

[3] L. Hou, D. Samaras, T. M. Kurc, Y. Gao, J. E. Davis, and J. H. Saltz, "Efficient multiple instance convolutional neural networks for gigapixel resolution image classification," *arXiv preprint*, 2015.

[4] Y. Xu, T. Mo, Q. Feng, P. Zhong, M. Lai, I. Eric, and C. Chang, "Deep learning of feature representation with multiple instance learning for medical image analysis," in *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*. IEEE, 2014, pp. 1626–1630.

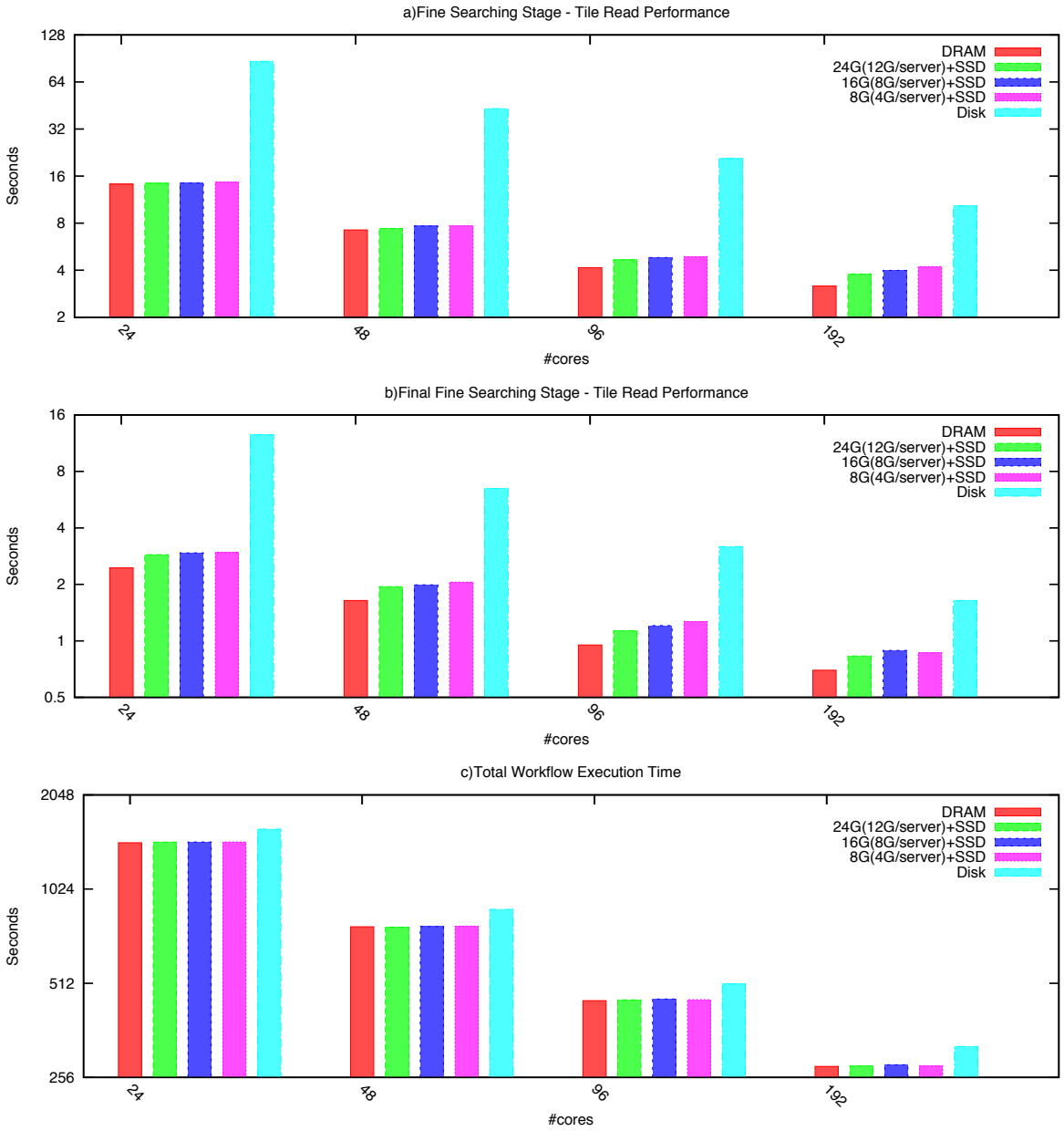


Fig. 4: Performance Results - #DSpaces servers=2

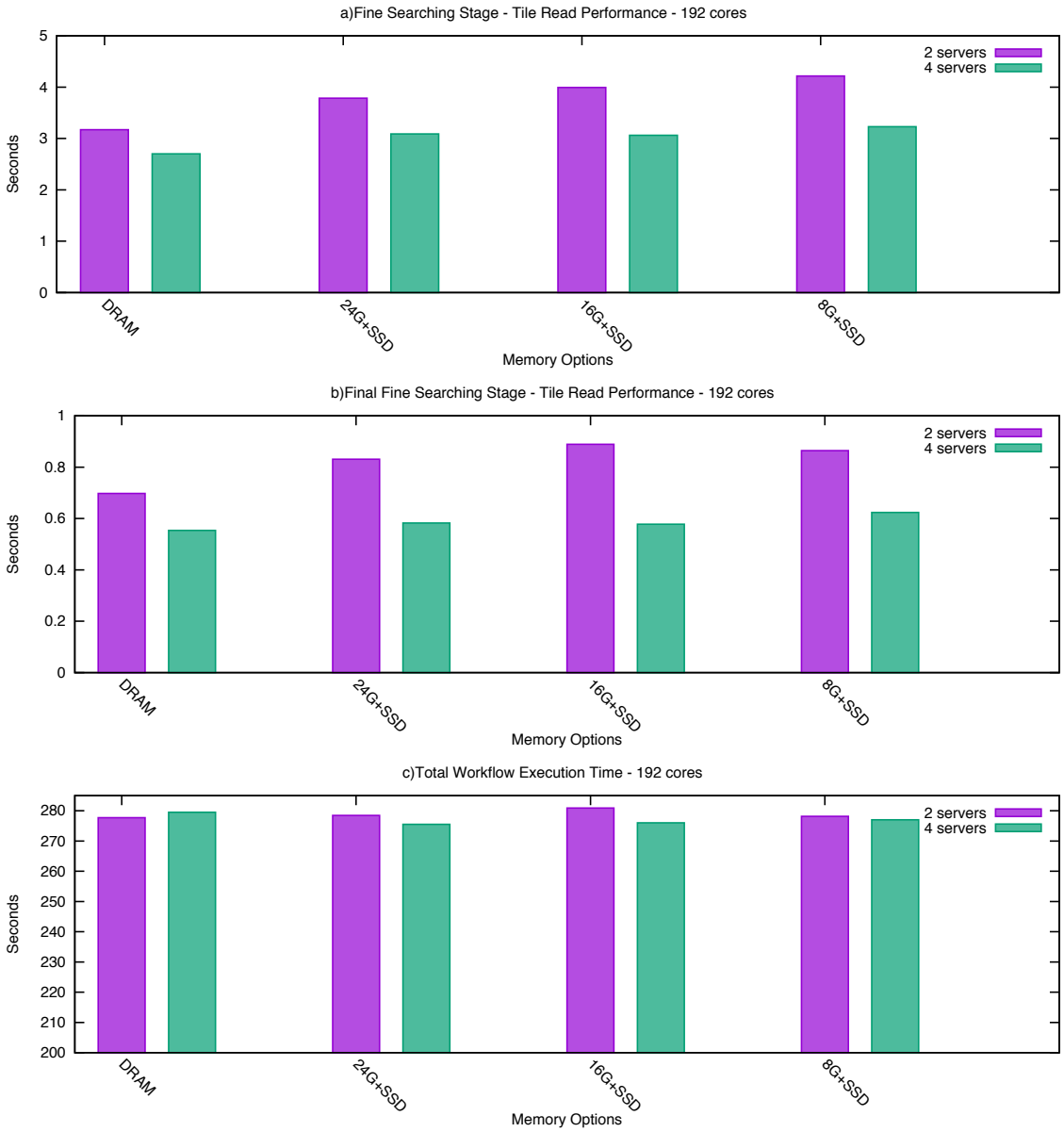


Fig. 5: Effect of Number of Staging Servers on Performance

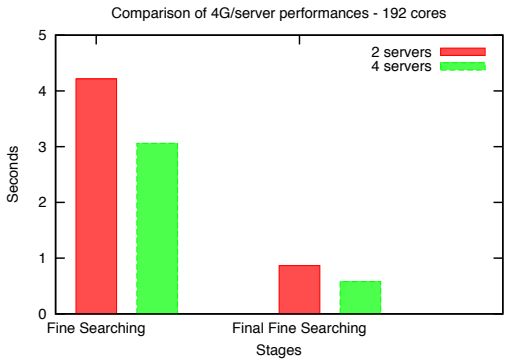


Fig. 6: 4G/server performances of Fine Searching and Final Fine Searching stages in DRAM+SSD setting

[5] N. Zerbe, P. Hufnagl, and K. Schlüns, "Distributed computing in image analysis using open source frameworks and application to image sharpness assessment of histological whole slide images," in *Diagnostic pathology*, vol. 6, no. 1. BioMed Central, 2011, p. S16.

[6] G. Bueno, R. Gonzalez, O. Déniz, M. García-Rojo, J. Gonzalez-Garcia, M. Fernández-Carrobles, N. Vállez, and J. Salido, "A parallel solution for high resolution histological image analysis," *Computer methods and programs in biomedicine*, vol. 108, no. 1, pp. 388–401, 2012.

[7] E. Yildirim and D. J. Foran, "Parallel versus distributed data access for gigapixel-resolution histology images: Challenges and opportunities," *IEEE journal of biomedical and health informatics*, vol. 21, no. 4, pp. 1049–1057, 2017.

[8] C. Docan, M. Parashar, and S. Klasky, "Dataspaces: an interaction and coordination framework for coupled simulation workflows," *Cluster Computing*, vol. 15, no. 2, pp. 163–181, 2012.

[9] (2019) Nvm express overview. [Online]. Available: https://nvmexpress.org/wp-content/uploads/NVMe_Overview.pdf

[10] (2019) Nvme vs ssd vs hdd. [Online]. Available: https://www.netweaver.uk/nvme-vs-ssd-vs-hdd/

[11] (2019) Mpi standard. [Online]. Available: http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf
[12] (2019) Openmp standard. [Online]. Available: https://www.openmp.org
[13] R. L. Grossman, A. P. Heath, V. Ferretti, H. E. Varmus, D. R. Lowy, W. A. Kibbe, and L. M. Staudt, "Toward a shared vision for cancer genomic data," *New England Journal of Medicine*, vol. 375, no. 12, pp. 1109–1112, 2016.

Performance Evaluation of Advanced Features in CUDA Unified Memory

Steven W. D. Chien
KTH Royal Institute of Technology
Stockholm, Sweden
wdchien@kth.se

Ivy B. Peng
Lawrence Livermore National Laboratory
Livermore, USA
peng8@llnl.gov

Stefano Markidis
KTH Royal Institute of Technology
Stockholm, Sweden
markidis@kth.se

***Abstract*—CUDA Unified Memory improves the GPU programmability and also enables GPU memory oversubscription. Recently, two advanced memory features, *memory advises* and asynchronous *prefetch*, have been introduced. In this work, we evaluate the new features on two platforms that feature different CPUs, GPUs, and interconnects. We derive a benchmark suite for the experiments and stress the memory system to evaluate both in-memory and oversubscription performance.**

The results show that *memory advises* on the Intel-Volta/Pascal-PCIe platform bring negligible improvement for in-memory executions. However, when GPU memory is oversubscribed by about 50%, using *memory advises* results in up to 25% performance improvement compared to the basic CUDA Unified Memory. In contrast, the Power9-Volta-NVLink platform can substantially benefit from *memory advises*, achieving up to 34% performance gain for in-memory executions. However, when GPU memory is oversubscribed on this platform, using *memory advises* increases GPU page faults and results in considerable performance loss. The CUDA *prefetch* also shows different performance impact on the two platforms. It improves performance by up to 50% on the Intel-Volta/Pascal-PCI-E platform but brings little benefit to the Power9-Volta-NVLink platform.

***Index Terms*—CUDA Unified Memory, UVM, CUDA memory hints, GPU, memory oversubscription**

I. Introduction

Recently, leadership supercomputers are becoming increasingly heterogeneous. For instance, the two fastest supercomputers in the world [17], Summit and Sierra, are both equipped with Nvidia V100 GPUs [6], [12] for accelerating workloads. One major challenge in programming applications on these heterogeneous systems arises from the physically separate memories on the host (CPU) and the device (GPU). Kernel execution on GPU can only access data stored on the device memory. Thus, programmers either need to explicitly manage data using the memory management API in CUDA or relying on programming systems, such as OpenMP 4.5 [7] and RAJA [5], for generating portable programs. Today, a GPU can have up to 16 GB memory on top supercomputers while the system memory on the host can reach 256 GB. Leveraging the large CPU memory as a memory extension to the relatively small GPU memory becomes a promising and yet challenging task for enabling large-scale HPC applications.

CUDA Unified Memory (UM) addresses the challenges as mentioned above by providing a single and consistent logical view of the host and device memories on a system. UM uses the virtual memory abstraction to hide the heterogeneity in GPU and CPU memories. Therefore, pages in the virtual address space in an application process may be mapped to physical pages either on CPU or GPU memory. Based on UM, CUDA runtime can leverage page faults, which is supported on recent GPU architectures, e.g., Nvidia Pascal and Volta architectures, to enable automatic data migration between device and host memories. For instance, when a device accesses a virtual page that is not mapped to a physical page on the device memory, a page fault is generated. Then, the runtime resolves the fault by remapping the page to a physical page on the device memory and copying the data. This procedure is also called on-demand paging. Now with the hardware-supported page fault and the runtime-managed data migration on UM, oversubscribing the GPU memory becomes feasible. For instance, when there is no physical memory available on the device for newly accessed pages, the runtime evicts pages from GPU to CPU and then bring the on-demand page.

CUDA has introduced new features for optimizing the data migration on UM, i.e., *memory advises* and *prefetch*. Instead of solely relying on page faults, the *memory advises* feature allows the programmer to provide data access pattern for each memory object so that the runtime can optimize migration decisions. The prefetch proactively triggers asynchronous data migration to GPU before the data is accessed, which reduces page faults and, consequently, the overhead in handling page faults.

In this paper, we evaluate the effectiveness of these new memory features on CUDA applications using UM. Due to the absence of benchmarks designed for this purpose, we developed a benchmark suite of six GPU applications using UM. We evaluate the impact of the memory features in both in-memory and oversubscription executions on two platforms. The use of *memory advises* results in performance improvement only when when we oversubscribe the GPU memory on the Intel-Volta/Pascal-PCI-E systems. On Power9-Volta-NVLink based system, using *memory advises* leads to performance improvement only for in-memory executions. With GPU memory oversubscription, it results in substantial performance degradation. Our main contributions in this work are as follows:

- We survey state-of-art practice in UM *memory advises*, *prefetch*, and GPU memory oversubscription.

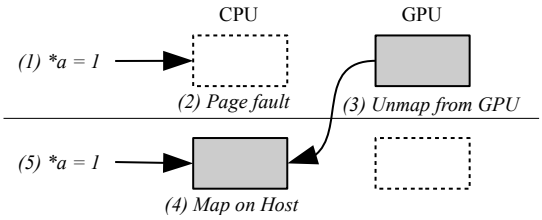


Fig. 1: CPU writes to a page resident on the GPU, triggering a page fault and the page is migrated to CPU.

- We design a UM benchmark suite consisting of six applications for evaluating advanced memory features.
- We evaluate the performance impact of *memory advises*, *prefetch* on two systems with Intel, Nvidia Pascal, and Volta GPUs connected via PCI-E and a system with IBM Power9 and Nvidia Volta GPU connected via NVLink.
- Our results indicate that using *memory advises* improves application performance in oversubscription execution on the Intel platform and in-memory executions on the IBM platform.
- We show that UM *prefetch* provides a significant performance improvement on the Intel-Volta/Pascal-PCI-E based systems while it does not show a performance improvement on the Power9-Volta-NVLink based system.

II. Unified Memory

In this section, we introduce the underlying mechanism in GPU UM, and the three memory advises. We also describe the prefetching and memory oversubscription.

A. CUDA Unified Memory

UM creates a unified logical view of the physically separate memories across host and GPU. Currently, modern CPUs support 48-bit memory addresses while Unified Memory uses 49-bit virtual addressing, which can address both host and GPU memories [14]. One of the main goals of Unified Memory is to provide a consistent view of data between devices. The system ensures a memory page can only be accessed by one process at a time. When a process accesses a page that is not resident of its memory system, a page fault occurs. The memory system holding the requested page will unmap it from its page table, and the page will be migrated to the faulting process. Figure 1 illustrates an example when the CPU accesses a page on GPU memory, and the page is migrated to CPU memory. Similarly, when GPU accesses a page not physically stored on GPU memory, the page will be moved to GPU.

UM was first introduced in CUDA 6.0 [21]. Only until the recent Nvidia Pascal microarchitecture that has hardware support for page faults, bi-directional on-demand page migration becomes feasible [14]. Resolving a page fault has high overhead, and memory thrashing that moves the same pages back and forth between the memories is even a performance bottleneck. The massive parallelism on GPU further exacerbates the page fault overhead because processes stall when page faults are being resolved, and multiple threads in different warps accessing the same page can cause multiple duplicated faults [18].

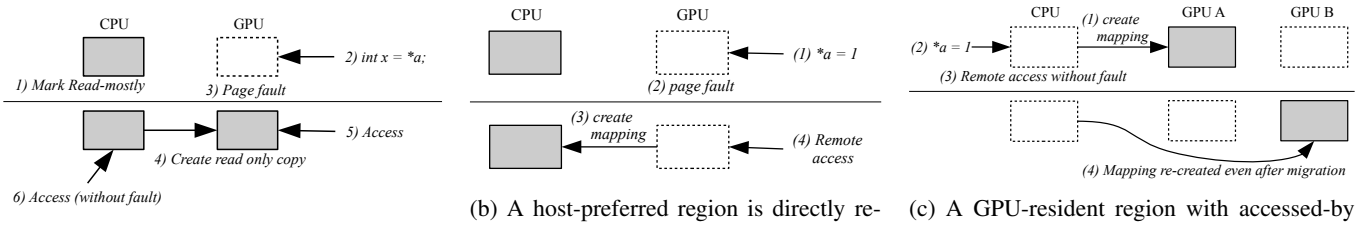
B. Data Movement Advises

CUDA 8.0 introduces a new programming interface, called memory advise [15]. The concept is similar to `posix_madvise` in Linux, which uses application knowledge about access patterns to make informed decisions on page handling [1]. The UM advise focuses on data locality, i.e., whether a page is likely to be accessed from the host or device. The main objective is to reduce unnecessary page migration and their associated overhead. Currently, developers can specify three access patterns to the CUDA runtime:

cudaMemAdviseSetReadMostly implies a read-intensive data region. In the basic UM, accessing a page on a remote side triggers page migration. However, with `cudaMemAdviseSetReadMostly`, a read-only duplicate of the page will be created on the faulting side, which prevents page faults and data migration in the future. Figure 2a illustrates an example, where the second access (step 5) has no page fault and is local access. This mechanism, however, results in a high overhead if there is any update to this memory region because all copies of the corresponding page will be invalidated to preserve consistency between different copies. Thus, this advice is often used in read-only data structures, such as lookup tables and application parameters.

cudaMemAdviseSetPreferredLocation sets the preferred physical location of pages. This advice pins a page and prevents it from migrating to other memories. Figure 2b illustrates a page preferred on the host side, and GPU uses remote mapping to access the page. This advice established a direct (remote) mapping to the memory page. When accessing the page remotely, data is fetched through the remote memory instead of generating a page fault. If the underlying hardware does not support the remote mapping, the page will be migrated as in the standard UM. `cudaMemAdviseSetPreferredLocation` is useful for applications with little data sharing between CPU and GPU, i.e., part of the application is executed completely on the GPU, and the rest of the application executes on the host. Data that is being used mostly by the GPU can be pinned to the GPU with the advice, avoiding memory thrashing.

cudaMemAdviseSetAccessedBy establishes a direct mapping of data to a specified device. Figure 2c illustrates an example of a physical page on GPU being remotely access from the host. When *cudaMemAdviseSetPreferredLocation* is applied, CUDA runtime tries to build a direct mapping to the page to avoid data migration so that the destination can access data remotely. Differently from *cudaMemAdviseSetPreferredLocation*, this *cudaMemAdviseSetAccessedBy* does not try to pin pages on a specific device; instead, its main effect is to establish mapping on the remote device. This advice takes effect on the creation of the memory pages. The mapping will be re-established after the pages are migrated.



(a) A read-mostly region is duplicated to the GPU to avoid page faults in the future.

(b) A host-preferred region is directly remote mapped to allow remote access from the GPU.

(c) A GPU-resident region with accessed-by CPU advise can be accessed by CPU through remote memory access.

Fig. 2: Page fault mechanism and effects of the three *Memory Advise* in Unified Memory.

C. Prefetching

The CUDA interface introduces an asynchronous page prefetching mechanism, i.e., *cudaMemPrefetchAsync()* [15], to trigger data migration. The data migration occurs in a background CUDA stream to avoid stalling the computation threads. One natural optimization for prefetching a large number of pages is to split into multiple streams, i.e., a bulk transfer, to prefetch pages in a batch of streams concurrently. If the page is prefetched to the device memory before the data access, no page faults will occur, and the GPU benefits from the high bandwidth on its local memory.

The behavior of the prefetching mechanism might change when used in combination with CUDA memory advises. For example, when *cudaMemAdviseSetReadMostly* is set, a read-only copy will be immediately created. Also, when prefetching a region with *cudaMemAdviseSetPreferredLocation* set to another destination memory, the pages will no longer be pinned to the preferred location. Thus, our evaluation considers the interplay between these two types of memory features.

D. Oversubscription of Device Memory

GPU memory has a relatively small capacity compared to the system memory on CPU. One major limitation when porting large-scale applications to GPUs is to overcome their memory capacity to enable larger problems. UM in the post-Pascal page fault capable GPUs can oversubscribe GPU memory, allowing GPU kernels to use more memory than the physical capacity on the device. The memory oversubscription is achieved through the traditional virtual memory management, i.e., selected memory pages on the device are evicted to CPU to make space for newly requested pages. Currently, the CUDA runtime uses the Least Recently Used (LRU) replacement policy to select victim pages when running out of space [19]. Some work also proposed pre-eviction to start page eviction early to avoid stalling on the critical path [3].

III. Methodology

We develop a benchmark suite for evaluating UM and different data migration policies. Although several porting efforts have been reported for specific applications, there lacks a suite of diverse kernels for controlled experiments across platforms. Thus, we extend the memory management in popular GPU benchmark and applications to utilize UM with advanced advise and prefetching features.

A. Application and Benchmarks

Our benchmark suite includes six applications, as specified in Table I. These applications include numerical solvers, financial application, image processing, and graph problems. The benchmark suite is available at a repository [1].

For each application, we develop four versions in addition to the original version that uses explicit GPU memory allocation. Our benchmarks use long data types to support large input problems in oversubscription executions. We use GPU kernel execution time as the figure of merit.

We present detailed tracing results for BS, CG, and FDTD3d on selected platforms to study the implications of data movement. BS is a financial application that performs option pricing. BS features good data reuse because the same input data set is used in multiple iterations in the application lifetime. CG is a linear solver that solves a linear system $Ax = b$ on the GPU. An error is computed on the host using the results from GPU computation after the solving iteration finishes. FDTD3d is a finite difference solver that reads and writes to two arrays in an interleaving manner. Both arrays are being initialized using the same data. The output eventually resides in one of the arrays.

1) UM: The first version is an implementation that uses UM with minimal changes. We simply replace the memory allocation in applications from *cudaMalloc()* to *cudaMallocManaged()* and eliminate explicit data copy, i.e., *cudaMemcpy()*, between host and device. After the completion of a GPU kernel, if the application has no subsequent host computation using the GPU results, an explicit data copy by *memcpy()* is inserted to simulate a CPU computation using the results.

2) UM Advise: The second version is UM with Advise. This version is based on the basic UM version and applies memory advises to data structures in the application. A stall in GPU execution, e.g., for resolving page fault, has a significant impact on performance due to massive parallelism. Thus, the main consideration for memory advises is to keep data used by GPU close to GPU memory. Therefore, we set a *cudaMemAdviseSetPreferredLocation* and specify the preferred location to GPU memory after the memory allocation of a data structure that is accessed by GPU in the computation. If the data structure is initialized by the CPU,

[1] https://github.com/steven-chien/um-apps

TABLE I: Applications and data input sizes on different platforms.

Name	Description	Input size Intel-Pascal (Approximate GB)		Input size Intel-Volta & P9-Volta (Approximate GB)	
		In-memory	Oversubscribe	In-memory	Oversubscribe
Black-Scholes (BS)	A financial application that performs option pricing.	4	6.4	15.2	26
Matrix Multiplication (cuBLAS)	A general matrix matrix multiplication in single precision using cuBLAS.	3.9	6.3	15.2	25.4
Conjugate Gradient (CG)	A conjugate gradient solver that solves a sparse linear system using cusparse.	3.8	6.4	15.4	25.4
Graph500	Breadth-first search (BFS) kernel of Graph500.	3.63	7.62	8.52	N/A
Convolution 0 (conv0)	A FFT-based image convolution using Real-to-Complex and Complex-to-Real FFT plans.	2.8	6.4	11.6	25.6
Convolution 1 (conv1)	A FFT-based image convolution using Complex-to-Complex FFT plan.	3.5	6.7	13.6	25.5
Convolution 2 (conv2)	A FFT-based image convolution using Complex-to-Complex FFT plan.	3.0	6.4	11.6	25.5
Finite Difference Time Domain (FDTD3d)	A finite difference solver in three dimension.	3.8	6.4	15.2	25.3

we set a *cudaMemAdviseSetAccessedBy* CPU to keep the data physically on GPU but establish a remote mapping on CPU. With this optimization, the host data initialization performs remote accesses to initialize data in GPU memory directly. For constant data structures, the *cudaMemAdviseSetReadMostly* advice is set after data initialization. This optimization will only have page fault at the first access but keep all subsequent accesses local.

3) UM Prefetch: The third version is UM with prefetch. We apply `cudaMemPrefetchAsync` to trigger page migration at appropriate sites explicitly. We prefetch large data structures that will be accessed by GPU kernels in a background stream while the GPU kernel is launched in the default stream. After completing the GPU kernel execution, we prefetch the arrays containing results to the host memory in the default stream. One advantage of bulk transfer in prefetch, compared to resolving individual page fault groups, is high memory bandwidth to utilize the hardware capability fully. Explicitly triggering page pages in bulk improves transfer efficiency. Furthermore, to prefetch pages avoids page faults as data already resides in the physical memory when the kernel starts executing.

4) UM Both: Finally, in the fourth version, we combine memory advises and prefetch to examine the mutual effects of both techniques.

B. Test Environment

We evaluate our benchmark applications on three platforms:

1) **Intel-Pascal** is a single node system with Intel Core i7-7820X processor and 32 GB of RAM. It has one GeForce GTX 1050 ti GPU with 4GB memory. The GPU is connected through PCIe. The operating system is Ubuntu 18.10 and the host compiler is GCC 8.3.
2) **Intel-Volta** is a GPU node on Kebnekaise at HPC2N in Umeå. It has an Intel Xeon Gold 6132 processor with 192 GB of RAM. The node has two Tesla V100 GPU with 16 GB memory and the GPU is connected through PCIe. The operating system is Ubuntu 16.04 and the host compiler is GCC 8.2.
3) **P9-Volta** is a node with an IBM Power9 processor and 256 GB of RAM. The system has four Tesla V100 GPUs with 16 GB of HBM. The GPU is connected through NVLINK to CPU.

Our platforms consist of two Intel systems that use Pascal and Volta GPUs, and a Power9 system that uses Volta GPU. All the systems use CUDA 10.1. We only use one GPU in the experiments. For each application variation, we perform benchmark runs up to five times and present the average GPU kernel execution time and standard deviation. An exception is Graph500, where we report the average and standard deviation of BFS iterations. We separate our experiments into two cases: when problem size fits into GPU memory and when oversubscription of memory is required. Their problem sizes are selected to be approximately 80% and 150% to GPU memory, respectively. A detailed list of sizes is presented in Table I. Due to the limitation in implementation for input data size, we only examine Graph500 with oversubscription on Intel-Pascal. However, the input size does not follow the 150% data size rule.

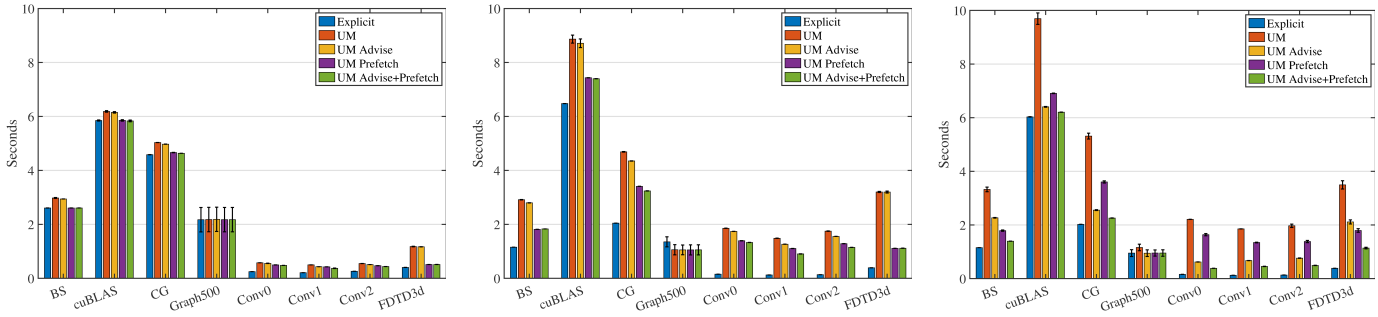
Apart from benchmark executions, we perform profiling runs using *nvprof* for selected applications. We obtain the trace by *–print-gpu-trace*. By selecting entries with *Unified Memory Memcpy HtoD* and *Unified Memory Memcpy DtoH*, we can build a time series of data movement. Through a comparison of the time series and time spent on memory movement, it is possible to compare and characterize the intensity of data movement between different application variations.

IV. Results

In this section, we present the performance and profiling results of the applications in four configurations: basic UM (UM), UM with Advise (UM Advise), UM with Prefetch (UM Prefetch) and UM with both Advise and Prefetch (UM Both). Each application is evaluated in each configuration with two problem sizes: one that fits into GPU memory (in-memory execution) and one that oversubscribes GPU memory (oversubscription execution). We report the average and standard deviation of GPU kernel execution time for each application.

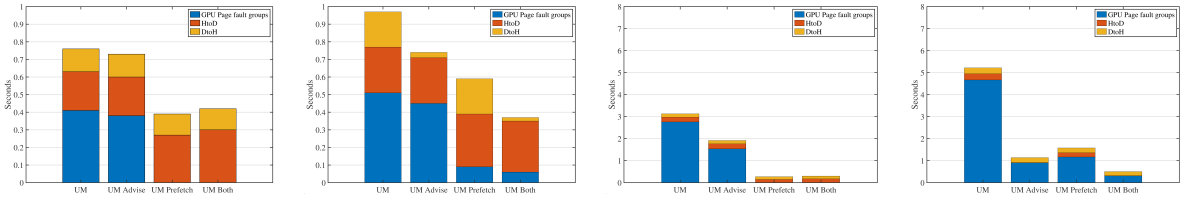
A. In-Memory Execution

We present the GPU kernel execution time of the applications in Fig. 3. The performance of all applications decreases when using basic UM instead of explicit data movement between CPU and GPU memories. Performances on Volta GPUs platforms have a larger performance decrease. In particular, our convolution and FDTD3d applications exhibit a drastic increase in execution time. The execution time of conv2 and FDTD3d are $14\times$ and $9\times$ higher respectively on P9-Volta. Performance change is similar to Intel-Volta. Performance decrease is less drastic but still considerable on Intel-Pascal. The execution of both applications is $2-3\times$ slower than the execution time of applications using explicit data movement.



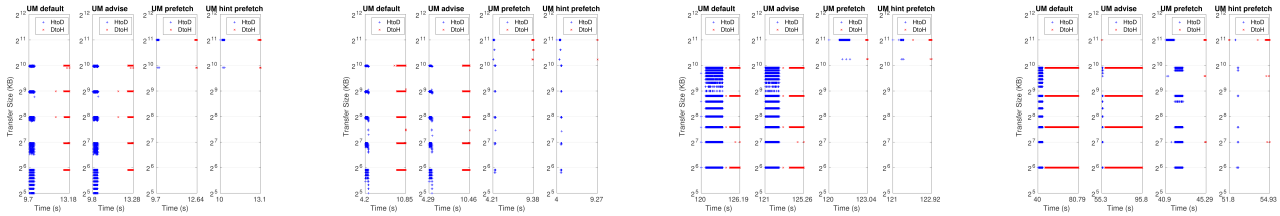
(a) Intel-Pascal (b) Intel-Volta (c) Power9-Volta.

Fig. 3: GPU kernel execution time of applications where data fits in GPU memory.



(a) BS on Intel-Pascal (b) CG on Intel-Pascal (c) BS on P9-Volta (d) CG on P9-Volta

Fig. 4: Breakdown of total time spent handling page faults and data movement when applications are running in-memory.



(a) BS Intel-Pascal (b) CG Intel-Pascal (c) BS P9-Volta (d) CG P9-Volta

Fig. 5: UM data transfer traces when running in-memory.

After applying advises, the performance of our applications generally improves. It is possible to improve execution time up to 15% on Intel-based platforms. The impact of advises is higher for the three FFT based convolution applications on Intel platforms. Advises have a significant impact on all the applications, and execution time can be improved by up to 70% on the P9 platform. Applications, such as CG and cuBLAS, results in similar execution time to the original version with explicit memory allocation. This implies that some advises are more effective than others on the P9 platform.

Expensive page fault handling can be avoided by prefetching data to the GPU before execution. Our results show that prefetch has a much higher impact on Intel-based platforms than P9-based platforms. Application performance generally improves when prefetch is used: our results show that it has a much higher impact on Intel-based platforms than the case advise is used. The performance of FDTD3d improves by up to 56% on the Intel-Pascal system. The performance of Black-Scholes application is close to on-par with the application version using explicit data transfer. As for Intel-Volta, the performance of FDTD3d improves by up to 65%. Performance improves by 50% on the P9-Volta system. However, the improvement is less than when only advises are applied. Despite that, we notice that when both advises and prefetch are used together, it generally outperforms the performance of applications using only advises or prefetch.

To better understand the difference in terms of data movement between the versions, we plot the total time spent on different UM events in Fig. 4 as stacked bar plots. They show the total time spent on GPU page fault group handling and data transfer, respectively. In particular, we have selected BS and CG for the comparison. The bar plot reveals two important information for comparison: the time spent on data movement, which correlates to the amount and efficiency of data transferred, as well as stalls due to page fault, which correlates to the number of page fault and efficiency of fault resolution.

Since the Black-Scholes application uses the same input dataset repeatedly over iterations, when data size fits in memory, the first iteration will be slower due to page migration.

Subsequent iterations should be able to execute at full speed as data already resides in device memory. For this particular application, the advise *cudaMemAdviseSetReadMostly* is applied to the input arrays. No other advise is applied. The same goes for prefetch. Figs. 4a and 4c show the break down of total time spent on data-related activities on the two platforms for the Black-Scholes application. Comparing to Intel-Pascal, the data transfer is much faster on P9-Volta, while the impact of stalling is less profound on Intel-Pascal. This can be attributed to the larger input data used and a faster interconnect on P9-Volta. For UM Advise, the time spent on data transfer is similar while the time spent stalling due to page fault has reduced. This suggests that page fault handling becomes more efficient when the advises are applied. The observation is similar for both Intel-Pascal and Intel-Volta similarly. When prefetch is used, the same amount of data is being transferred while the stall due to page fault is eliminated. This implies the complete elimination of page faults. By prefetching pages in bulk, data can be transferred at a fast pace to avoid future page faults when accessed. The observation can be confirmed by Figs. 5a and 5c, where the detailed transfers are plotted as a time series. When prefetch is applied, data is transferred as a block at a much higher rate.

The Conjugate Gradient application solves a linear system $Ax = b$ iteratively. When applying advises, we set the preferred location of matrix A and vector b to GPU memory. We also set a read-mostly advise on the sparse matrix after completing initialization. The breakdown of time spent on Intel-Pascal and P9-Volta are shown in Figs.4b and 4d, respectively. The use of advises results in similar time spent on data transfer from host to device but a slight reduction in time on stalls on Intel-Pascal system. A considerable reduction in time spent on the host to device transfer and stall is observed on the P9-Volta system. One reason is the use of preferred location advise, where the data arrays are initialized from the host on GPU memory through remote memory access. On Power9, it is possible for the CPU to access GPU memory while this is not possible on Intel platforms. At the same time, time spent on transfer from device to host is largely eliminated on Intel-Pascal. One possible reason is due to the read-mostly advise. Instead of migrating pages to the GPU from host memory, a read-only copy is copied to the GPU. This means that a copy of data exists in both memory systems. When the Ax is being computed, A can be fetched directly in host memory. Since P9-Volta initializes data directly in GPU memory, a copy has to be fetched back to the host. In this case, the naive use of prefetch results in a reduction of time spent stalling. Despite the fact that more data is transferred from device to host, the use of prefetch results in a higher transfer rate. The data transfer trace is presented in Figs. 5b and 5d. When used in combination with advises, it results in a reduction of time for data transfer and stall.

B. Oversubscription Execution

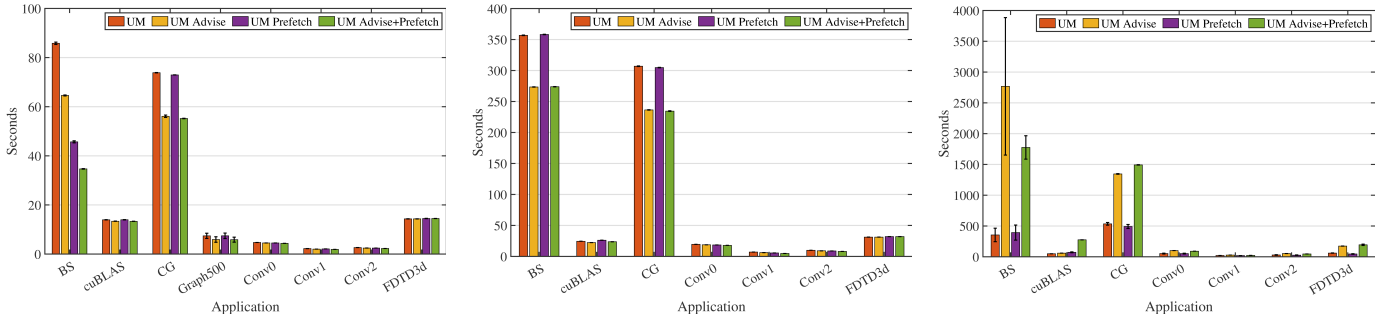
Oversubscription of GPU memory is a key new feature of UM. It resembles the paging of unused memory pages to secondary storage to free up memory in classical virtual memory management. Similarly to the CPU memory subscription case, excessive use can lead to system slowdown and can severely impact performance. Our results show that all applications execute correctly, even when running out of GPU memory. However, techniques that improved performance for in-memory do not necessarily perform well when GPU memory is oversubscribed. On the contrary, the use of these techniques without careful optimization can lead to severe performance degradation.

We present the execution time of our applications in Fig. 6. Since the case does not exist with original versions with explicit allocation, a comparison is not possible. Instead, the minimal UM version is used as a baseline. By using advise, specific applications can achieve up to over 20% improvement on Intel platforms. Our P9 platform, on the other hand, shows a negative impact when advises are used. To better understand data movement, we perform tracing with the BS and CG on Intel-Pascal, and with BS and FDTD3d on P9-Volta.

For the Black-Scholes application, the use of advise results in performance improvement on Intel-Pascal. Fig. 7a shows the breakdown of time spent on page-fault related events of BS between host and device while Fig. 8a shows the detailed tracing on Intel Pascal. One significant difference between default UM and UM advise is that a lot less time is spent on transferring data back to the host. The reduction in data movement can contribute to the improvement in performance. One possible reason for the reduction in data transfer from device to host is that instead of migrating data from GPU to host memory to make space, read-only data can simply be discarded as a copy already exists on host memory. On the other hand, on P9-Volta, significantly more time is spent on stalls. This can be seen in Fig. 7c, where the total time is a few times higher than when no advise is used. Fig. 8c examines the data movement traces and clearly shows an intense data movement in both directions. This implies that the read-mostly advise has an interestingly negative effect on P9-Volta when data size exceeds device memory. A naive prefetch on Intel-Pascal provides performance improvement; however, it has little to no effect on P9-Volta.

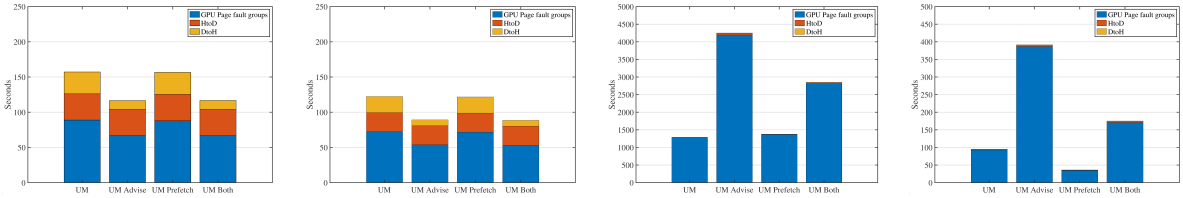
CG on the Intel-Pascal platform benefits from using advise. The time breakdown for page faults and data movement is shown in Fig. 7b. As in the case of the Black-Scholes application, less time is spent on transferring data back to the host than in the case of basic UM. However, we note that a similar amount of data is sent from host to device in the two cases. This can also be seen in the detailed tracing in Fig. 8b, where less device to host transfer is made.

FDTD3d is a finite difference solver, and it reads and writes to two arrays in an interleaving manner. Both arrays are being initialized using the same data. One of the arrays is being set to prefer GPU memory and will be accessed by the CPU. No advise is set on the other array. Since both arrays will be written to during execution, no read-mostly advise is set for them. However, read-mostly is set for a small array that contains coefficients. Fig. 7d shows the breakdown



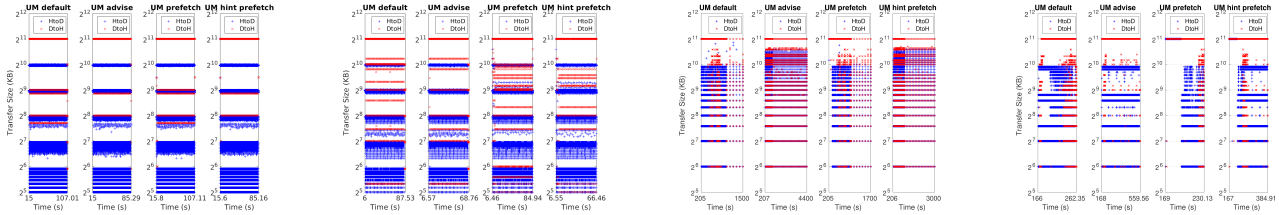
(a) Intel-Pascal (b) Intel-Volta (c) Power9-Volta.

Fig. 6: GPU kernel execution time of applications where data do not fit in GPU memory.



(a) BS on Intel-Pascal (b) CG on Intel-Pascal (c) BS on P9-Volta (d) FDTD3d on P9-Volta

Fig. 7: Breakdown of total time spent handling page faults and data movement when input size exceeds GPU memory.



(a) BS Intel-Pascal (b) CG Intel-Pascal (c) BS P9-Volta (d) FDTD3d P9-Volta

Fig. 8: UM data transfer traces when input size exceeds GPU memory.

of time in handling data movement and page faults on P9-Volta. Similarly to the Black-Scholes application, the usage of advise results in much higher spent on stalling. Execution time also increased significantly by approximately 3×. When prefetching, only one of those two data arrays is prefetched as they are originally identical. Interestingly, less data is seen transferring in both directions when prefetch is used. Fig. 8d shows the detailed tracing of the application. Smaller data transfers at the beginning become a bulk transfer. This is also reflected in the execution time, which is reduced from 60.9s to 45.3s as well as a reduction in time spent stalling. One possible reason is the size of the array being prefetched. Since only one array, which represents 50% of the total problem size, is prefetched, the entire array can reside entirely on GPU memory without needed to evict previously prefetched data.

V. Related Work

The separate memory system between host and GPU has long been a programming challenge for developers. With UM, the runtime can transparently handle data movement between CPU and GPU. Earlier works [11], [13] have investigated the impact of UM in applications while [13] investigated the programming model support for UM in OpenMP through an extended LLVM compiler. These studies lack the support of advanced memory features, which only become available recently. Recent efforts in the operating system, such as Heterogeneous Memory Management (HMM) in the Linux kernel [4], [8], [20], provides mechanisms to mirror CPU page table on GPU and integrate device memory pages in the system page table by adding a new type of struct page.

CPU to GPU interconnect is another factor that impacts the performance of data movement directly. Extensive efforts have reported evaluation on modern GPU systems [6], [9]. For instance, [16] developed a microbenchmark tool to evaluate the raw bandwidth performance with UM. While their works focus on interconnect performance and provide optimization insights, our work focuses on the impact of advanced memory features in optimizing the locality of pages.

Some of the recent works that apply advanced features of UM are Deep-Learning frameworks. One example is OC-DNN [2], an extended Caffe framework that uses UM to

support the training of out-of-core batch sizes. They use memory advises to trigger data eviction and prefetch to trigger migration. They find these techniques useful in optimizing training performance. However, incorrect use can lead to performance degradation.

The memory oversubscription in GPU memory requires efficient page eviction to make space for newly requested pages. [3] proposed two pre-eviction policies using a tree-based neighborhood prefetching technique to select candidate pages. [10] introduced an ETC framework for eager page pre-eviction and memory throttling in memory trashing. However, these optimization techniques target future GPU designs that require hardware modifications.

VI. Conclusion

In this work, we investigated the impact of UM *memory advises*, *prefetch*, and GPU memory oversubscription, on CUDA application performance. We found that the performance of *memory advises* mostly depends on the system in use and whether the GPU memory is oversubscribed. The use of *memory advises* results in a performance improvement only when the GPU memory is oversubscribed on the Intel-Volta/Pascal-PCI-E systems. The use of *memory advises* on Power9-Volta-NVLink based system, leads to a performance improvement when applications run in-memory while it results in a considerable performance degradation with GPU memory oversubscription. CUDA Unified *prefetch* provides a performance improvement only on the Intel-Volta/Pascal-PCI-E based systems while it does not show a performance improvement on the Power9-Volta-NVLink based system.

In this work, we have set *memory advises* for each memory object following best-practice guidelines from Nvidia. However, a future study on how to select optimal advise placement would help programmers derive different combinations of advises in different applications. In general, we found both *memory advises* and *prefetch* to be simple and effective. Overall, we showed that UM is a promising technology that can be used effectively when programming applications for GPU systems.

Acknowledgment

Funding for the work is received from the European Commission H2020 program, Grant Agreement No. 801039 (EPiGRAM-HS). Experiments were performed on resources provided by the Swedish National Infrastructure for Computing (SNIC) at HPC2N and Lassen supercomputer at LLNL. Part of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 LLNL-PROC-788778. This research was also supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

References

[1] Linux programmer's manual. http://man7.org/linux/man-pages/man3/posix_madvise.3.html, 2019.

[2] Ammar Ahmad Awan, Ching-Hsiang Chu, Hari Subramoni, Xiaoyi Lu, and Dhabaleswar K Panda. OC-DNN: Exploiting Advanced Unified Memory Capabilities in CUDA 9 and Volta GPUs for Out-of-Core DNN Training. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*, pages 143–152. IEEE, 2018.

[3] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. Interplay Between Hardware Prefetcher and Page Eviction Policy in CPU-GPU Unified Virtual Memory. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 224–235. ACM, 2019.

[4] Jerome Glisse. Redhat heterogeneous memory management. https://linuxplumbersconf.org/event/2/contributions/70/attachments/14/6/hmm-lpc18.pdf, 2018.

[5] Richard D Hornung and Jeffrey A Keasler. The raja portability layer: overview and status. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2014.

[6] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. *arXiv preprint arXiv:1804.06826*, 2018.

[7] Ian Karlin, Tom Scogland, Arpith C Jacob, Samuel F Antao, Gheorghe-Teodor Bercea, Carlo Bertolli, Bronis R de Supinski, Erik W Draeger, Alexandre E Eichenberger, Jim Glosli, et al. Early experiences porting three applications to openmp 4.5. In *International Workshop on OpenMP*, pages 281–292. Springer, 2016.

[8] The kernel development community. The linux kernel 4.18.0. https://www.kernel.org/doc/html/v4.18/vm/hmm.html, 2019.

[9] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Xu Liu, Nathan Tallent, and Kevin Barker. Tartan: Evaluating Modern GPU Interconnect via a Multi-GPU Benchmark Suite. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 191–202. IEEE, 2018.

[10] Chen Li, Rachata Ausavarungnirun, Christopher J. Rossbach, Youtao Zhang, Onur Mutlu, Yang Guo, and Jun Yang. A framework for memory oversubscription management in graphics processing units. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 49–63, New York, NY, USA, 2019. ACM.

[11] Wenqiang Li, Guanghao Jin, Xuewen Cui, and Simon See. An Evaluation of Unified Memory Technology on Nvidia GPUs. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 1092–1098. IEEE, 2015.

[12] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter. NVIDIA Tensor Core Programmability, Performance Precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 522–531, May 2018.

[13] Alok Mishra, Lingda Li, Martin Kong, Hal Finkel, and Barbara Chapman. Benchmarking and evaluating unified memory for OpenMP GPU offloading. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*, page 6. ACM, 2017.

[14] NVIDIA. P100 white paper. *NVIDIA Corporation*, 2016.

[15] NVIDIA. CUDA C Programming Guide. *NVIDIA Corporation*, 2019.

[16] Carl Pearson, Abdul Dakkak, Sarah Hashash, Cheng Li, I-Hsin Chung, Jinjun Xiong, and Wen-Mei Hwu. Evaluating Characteristics of CUDA Communication Primitives on High-Bandwidth Interconnects. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ICPE '19, pages 209–218. ACM, 2019.

[17] The TOP500 project. Top500 lists. https://www.top500.org/lists/2019/06/, 2019.

[18] Nikolay Sakharnykh. Maximizing unified memory performance in cuda. https://devblogs.nvidia.com/maximizing-unified-memory-performance-cuda/, 2017.

[19] Nikolay Sakharnykh. Unified memory on pascal and volta. http://on-demand.gputechconf.com/gtc/2017/presentation/s7285-nikolay-sakharnykh-unified-memory-on-pascal-and-volta.pdf, 2017.

[20] Nikolay Sakharnykh. Unified memory on pascal and volta. In *GPU Technology Conference (GTC)*, 2017.

[21] Nikolay Sakharnykh. Everything you need to know about unified memory. *NVIDIA GTC*, 2018.

Explicit Data Layout Management for Autotuning Exploration on Complex Memory Topologies

Swann Perarnau, Brice Videau, Nicolas Denoyelle, Florence Monna, Kamil Iskra, Pete Beckman
Argonne National Laboratory
{swann, ndenoyelle, bvideau, fmonna}@anl.gov, {iskra, beckman}@mcs.anl.gov

***Abstract*—The memory topology of high-performance computing platforms is becoming more complex. Future exascale platforms in particular are expected to feature multiple types of memory technologies, and multiple accelerator devices per compute node.**

In this paper, we discuss the use of explicit management of the layout of data in memory across memory nodes and devices for performance exploration purposes. Indeed, many classic optimization techniques rely on reshaping or tiling input data in specific ways to achieve peak efficiency on a given architecture.

With autotuning of a linear algebra code as the end goal, we present `AML`: a framework to treat three memory management abstractions as first-class citizens: data layout in memory, tiling of data for parallelism, and data movement across memory types. By providing access to these abstractions as part of the performance exploration design space, our framework eases the design and validation of complex, efficient algorithms for heterogeneous platforms.

Using the Intel Knights Landing architecture in one of its most NUMA configurations as a proxy platform, we showcase our framework by exploring tiling and prefetching schemes for a DGEMM algorithm.

***Index Terms*—Deep memory, high-bandwidth memory, explicit memory management**

I. Introduction

As we approach exascale, high-performance computing (HPC) platforms are increasingly featuring complex memory topologies. Intel's Knights Landing [1] (KNL) processor was an early indicator of this trend, with a configuration that can result in a single socket being split into 8 NUMA domains: 4 quadrants for the on-chip network and 2 types of memory (high-bandwidth MCDRAM and regular DRAM) per quadrant. Exascale-class systems are also expected to feature a topology including multiple accelerator devices with their own memory banks along with their host multicore processor. Regardless of the specific technology, these systems will exhibit **deep memory**: multiple levels of cache-coherent, byte-addressable memory.

How these complex topologies should be managed by HPC applications is still an open research question. Vendors have spent considerable effort on hiding most of this complexity behind automatic mechanisms that expose a single coherent virtual address space with each compute element pulling data as close as possible when needed. This is a case for the *cache mode* of the KNL, which uses the high-bandwidth memory as a last-level direct-mapped cache, as well as for the *unified memory* feature on Nvidia GPUs, which allows the GPU to pull pages of data allocated on the CPU side as needed.

We believe that these automatic cache-like management schemes are leaving performance on the table and, more importantly, prevent the internal structure of applications from treating the placement, movement, and shape of data as key performance concerns. In this paper, we revisit typical memory optimization strategies in the autotuning of dense linear algebra kernels as first-class abstractions that can be part of an application design space for heterogeneous architectures. These abstractions are implemented in **AML**: a library providing *building blocks* for the creation of explicit, application-aware memory management policies. Our library allows application and runtime developers to quickly design memory management schemes adapted to complex memory hierarchies, while still being agnostic to the specific hardware topology exhibited by the architecture. As a memory management framework, AML provides facilities to build custom data layouts, adapt memory placement according to those layouts, and handle coarse-grained memory movement between memory layers, as required for prefetching mechanisms.

To motivate exposing those abstractions to application developers, we present an extensive performance exploration study combining autotuning and careful data movement orchestration to achieve efficiency on large input sizes for a DGEMM algorithm on a KNL system configured as a large NUMA topology. We demonstrate that our library enables us to implement efficient tiling schemes and dynamic memory movements to prefetch inputs into the more efficient memory.

This paper is organized as follows. Section II presents our motivation: the reproduction, using autotuning, of an optimization strategy for DGEMM with large inputs. Using this complex memory management optimization as a starting point, we present in Section III three abstractions necessary for building a complex data orchestration policy on a heterogeneous topology and discuss how they can be provided to application performance specialists as building blocks inside the same framework. We explore the full performance of our data orchestration scheme in Section IV. Section V discusses related work, and we conclude in Section VI with a summary and some observations about future platform requirements.

II. Motivation

Our motivating problem is the fine-grained optimization of a DGEMM kernel ($C = A * B + C$) for large matrix sizes, utilizing a state-of-the-art strategy [2]. This strategy is a bottom-up, architecture-aware approach for maximizing

performance: for each level of the memory hierarchy of the target architecture, use subkernels and reorganize data to achieve the best compute intensity (ratio of memory loads to compute instructions). Such a strategy involves finely tuned reorganization of data across cache levels in a non portable way. For productivity and portability reasons, we would like to generalize and simplify the memory management of this kind of approach.

The DGEMM kernel proposed in [2] is tailored to target the Knights Landing architecture, which is notoriously difficult to tune for. In order to reach peak performance, every level of the memory hierarchy needs to be accounted for. The KNL node used in this study has 64 cores with 32 vector registers 512 bits wide (8 double-precision floats), 32 KiB of L1 cache and 1 MiB of L2 cache shared between each pair of cores. The node also has 16 GiB of MCDRAM memory, and although it can be used as a cache, it will exposed as a NUMA memory node in this study. Our strategy thus involves 4 levels of kernel optimization.

Inner Kernel: The inner kernel is optimized for register-level data access patterns. Register reuse is achieved by careful vectorization and limited use of load and store instructions. Specifically, the inner kernel works on small tiles of A, B, and C with sizes $[kb, mr]$, $[kb, nr]$, and $[mr, nr]$, respectively. We denote these tiles $\widehat{A}$, $\widehat{B}$, and $\widehat{C}$ (note that $\widehat{A}$ is transposed). Tile sizes are chosen so that nr is a multiple of the vector length (8 in our case), while mr is small enough that the whole tile $\widehat{C}$ and a row of $\widehat{B}$ fit inside the registers simultaneously. Thus, for this architecture, the possible values of $[mr, nr]$ are $(31, 8)$, $(15, 16)$ and $(7, 32)$. The last parameter, kb, should be large enough to amortize data transfer overheads on $\widehat{C}$ but also small enough for $\widehat{A}$ to fit inside half of the L1 cache.

Intermediary Kernel: The intermediary kernel works on sets of tiles of A, B, and C with sizes of $[kb, mb]$, $[kb, n]$, and $[mb, n]$, respectively. We denote these tilesets $\widetilde{A}$, $\widetilde{B}$, and $\widetilde{C}$. $\widetilde{A}$ is stored as mb/mr (or $nblocka$) consecutive tiles of size $[kb, mr]$ ($\widehat{A}$), $\widetilde{C}$ is stored as $(mb/mr) * (n/nr)$ (or $nblocka * nblockn$) consecutive tiles of size $[mr, nr]$ ($\widehat{C}$), and $\widetilde{B}$ is stored as n/nr (or $nblockn$) tiles of size $[kb, nr]$ ($\widehat{B}$). These sizes are chosen to take advantage of L2 caches, and in particular mb is chosen so that $\widetilde{A}$ fits inside half of the L2 cache available to a core.

Outer Kernel: The outer kernel updates a block C of size $[m, n]$, using a block A of size $[k, m]$ and a block B of size $[k, n]$. The layouts used here are as follows: C is stored as m/mb (or $nblockm$) blocks of $\widetilde{C}$, A is stored as $(k/kb) * (m/mb)$ (or $nblockk * nblockm$) blocks of $\widetilde{A}$, and B is stored as (k/kb) (or $nblockk$) blocks of $\widetilde{B}$. This is where our approach differs from [2]. In the original algorithm, the transposition of $\widehat{A}$ is performed in the intermediary kernel, while $\widehat{B}$ is transposed in the outer kernel. We extracted those transformations to take place before the outer kernel. This approach results in another level of blocking in the algorithm: the outer kernel is operating on blocks of A, B, and C with a shape of $[nblockk, nblockm, nblocka, kb, mr]$, $[nblockk, kb, nblockn, nr]$, and $[nblockm, nblockn, nblocka, mr, nr]$, respectively. We denote them $\dot{A}$, $\dot{B}$, and $\dot{C}$. The sizes m, n, and k are chosen so that the blocks are approximately square and large enough to amortize the transform cost. The pseudocode for this entire algorithm is given in Listing 1.

```
void inner_kernel(
  double ah[KB][MR],
  double bh[KB][NR],
  double ch[MR][NR]) {
  /* ch += ah * bh */
}

void intermediary_kernel(
  int nblockn,
  double at[NBLOCKA][KB][MR],
  double bt[nblockn][KB][NR],
  double ct[nblockn][NBLOCKA][MR][NR]) {
#pragma omp parallel for num_threads(NUM_INNER_TH)
  for (int jr = 0; jr < nblockn; jr++) {
    for (int ir = 0; ir < NBLOCKA; ir++) {
      inner_kernel( &at[ir][0][0],
                    &bt[jr][0][0],
                    &ct[jr][ir][0][0]);
    }
  }
}

void outer_kernel(
  int nblockm, int nblockn, int nblockk,
  double ad[nblockk][nblockm][NBLOCKA][KB][MR],
  double bd[nblockk][nblockn][KB][NR],
  double cd[nblockm][nblockn][NBLOCKA][MR][NR]) {
  for (int p = 0; p < nblockk; p++) {
#pragma omp parallel for num_threads(NUM_OUTER_TH)
    for (int i = 0; i < nblockm; i++) {
      intermediary_kernel(
        nblockn,
        &ad[p][i][0][0][0][0]
        &bd[p][0][0][0],
        &cd[i][0][0][0][0]);
    }
  }
}
```

Listing 1. Pseudocode for the DGEMM outer kernel.

Top Kernel: The top kernel is responsible for orchestrating the transformation of the input matrices and the launch of the outer kernel. Matrix $\bar{C}$ is of size $[M, N]$, matrix $\bar{A}$ of size $[M, K]$, and matrix $\bar{B}$ of size $[K, N]$. The matrices are arranged in blocks of size $[m, n]$ (C), $[m, k]$ (A), and $[k, n]$ (B), respectively. Those blocks are transferred from main memory to MCDRAM and transformed on the fly into blocks $\dot{C}$, $\dot{A}$, and $\dot{B}$, respectively. Since MCDRAM can hold several of those blocks simultaneously, computation and transfer can overlap, and some blocks can be reused, depending on the order of evaluation.

Challenge: This multilayered algorithm has one major drawback: its memory management is not an explicit parameter of the algorithm. Indeed, to port this algorithm to a different architecture, one must rediscover the appropriate shapes and sizes of the data for each topology level. To automate this work, memory management must be made explicit and composable enough that an application performance specialist

can automate the search for ideal parameters, using autotuning for example. Moreover, the layouts are complex enough that facilities to track and reason about these geometries are required.

III. Abstractions for Efficient In-Memory Data Orchestration

Our motivating problem points towards the need for a principled approach to three memory abstractions: layout, tiling, and movement across a topology. In order to improve on existing algorithms and adapt our work to future platforms, these three abstractions need to become available as first-class constructs that we can use for implementation and for further experimentation.

Such goal is also in line with the PADAL whitepaper [3], which identified that as we approach exascale, high-performance computing platforms will move toward cheap and massively parallel compute power while data movement will dominate energy and performance costs. To facilitate the development of applications on those platforms, the authors called for the community to establish locality-preserving abstractions. Data layout, tiling, and topology were identified as critical components of such an effort.

We present here how these three abstractions can be made available inside a single framework, as building blocks for further performance optimization studies. Our framework should have the following goals:

- *Composable*: application developers and performance experts should be able to pick and choose which building blocks to use depending on their specific needs.
- *Flexible*: one should be able to customize, replace, or change the configuration of each building block as much as possible.
- *Declarative*: to the extent possible, each building block should provide the means for users to describe how it is used by the application, without having to change the existing programming model.
- *Hardware-oblivious*: given the right initialization parameters, the resulting memory management policies should work on any hardware configuration.

Given these goals, we detail the scope and intent for each abstraction and highlight how they interact with one another.

Data Layouts: This abstraction is in charge of representing how the bytes of a data structure are organized in a linear virtual address space. In this paper, we focus on layouts typically encountered when dealing with dense linear algebra algorithms: multidimensional arrays of a single element type, with optional strides. This abstraction provides methods to find a single element and, more important, slicing and reshaping. We identify slicing as the act of returning a subset of the layout; reshaping provides the user with a view of the layout using different dimensions (without changing the underlying data organization).

Tiling Data: Tiling, or blocking, is a common optimization strategy to improve the data locality of an algorithm. It can be summarized as the action of grouping data elements to be placed or worked on together. Based on our motivating problem, tiling is the abstraction in charge of providing an indexation and generation mechanism over slices of a layout. Indeed, for a blocked DGEMM algorithm it is critical to be able to reason about the layout of entire input matrices as tiles for which coordinates are available so that the tiles can be iterated on correctly.

Table I
Summary of Data Management Abstractions

Abstraction	Intent	Methods
Layout	data organization in memory	`deref`, `reshape`
Tiling	generate/index slices	`index`, `slice`
Movement	orchestrate movement	`copy`, `transform`

Moving Data: The core abstraction behind the movement of data inside a topology can be identified by two main functions: the means to perform the movement itself and the type of movement being performed. Indeed, depending on the architecture and the specific programming model, moving data between memories can be done by a regular `memcpy`, by moving physical pages of data around, by calling into device-specific APIs (`cudaMemcpy`), by queuing copy operations (`clEnqueueCopyBuffer` in OpenCL), or even by using parallel runtimes (OpenMP 5.0). On the other hand, the data movement operation itself might include more than just copying and moving data around, for example, transposing matrices for better efficiency or packing/filtering the data to improve locality.

These abstractions, summarized in Table I, were implemented in **AML**, a lightweight framework written in C99 using pthreads for its asynchronous movement facilities.

IV. Performance Exploration of DGEMM on a Large Heterogeneous Topology

We now present how our memory management framework can be used to generalize the DGEMM optimization presented earlier, on Intel's Knights Landing architecture.

A. Experimental Setup

All the experiments presented in this section run on an Intel Knights Landing processor model 7210, comprising 16 GiB of high-bandwidth, on-package MCDRAM and 192 GiB of DDR. This platform has 64 cores available, running at 1.3 GHz, and hyperthreading is deactivated. Unless otherwise specified, the node is configured at boot time to run in Flat/Quad mode, meaning that the MCDRAM is exposed to the system as a single NUMA node and that the distributed cache directory is configured to improve latency on the cache misses.

The node is running CentOS 7.5, kernel version 3.10. The frequency governor has been set to `performance`. The benchmarks are compiled with icc 17.0.1; architecture-specific optimizations (`-xHost`) are active. All experiments are run using 60 cores for computation (with OpenMP) and 4 cores for data movement. The inner kernel is generated ahead of time by using the BOAST autotuning framework [4].

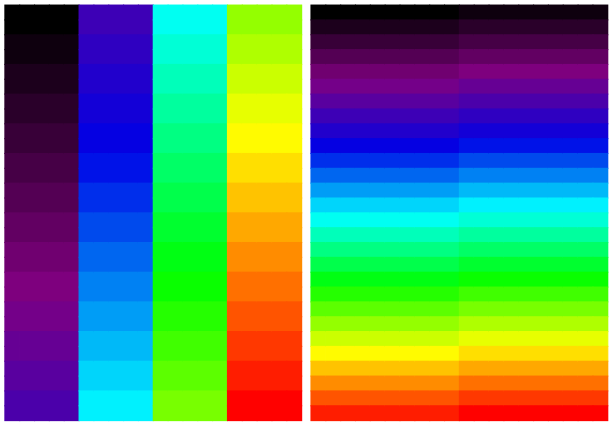


Figure 1. Example of a $\dot{A}$ before and after transposition. Elements of each $\widehat{A}$ have the same color.

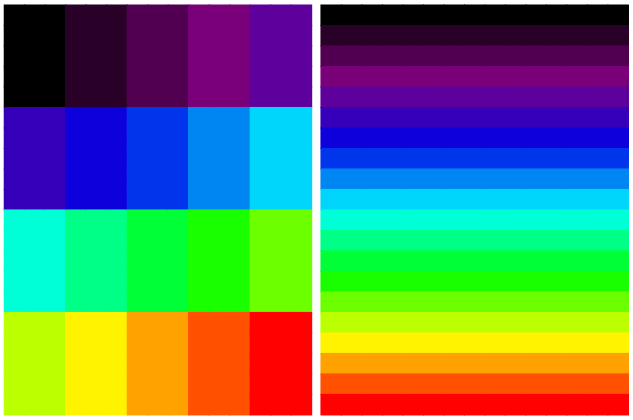


Figure 2. Example of a $\dot{B}$ before and after transposition. Elements of each $\widehat{B}$ have the same color.

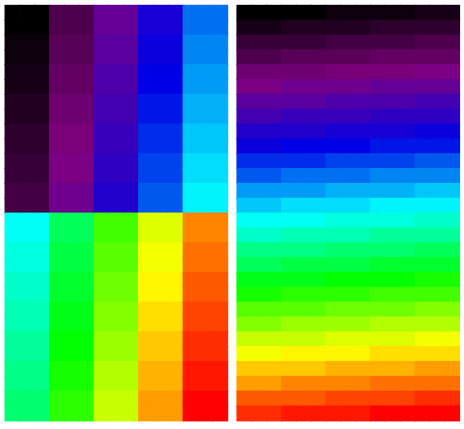


Figure 3. Example of a $\dot{C}$ before and after transposition. Elements of each $\widehat{C}$ have the same color.

```
/*
A dims: {NB, M, NB, K};
B dims: {NB, K, NB, N};
C dims: {NB, M, NB, N};

Tiles:
A dims:         {M, K};
A reshaped:     {NBLOCKM, NBLOCKA, MR, NBLOCKK, KB};
A transposed:   {NBLOCKK, NBLOCKM, NBLOCKA, KB, MR};

B dims:         {K, N};
B reshaped:     {NBLOCKK, KB, NBLOCKN, NR};
B transposed:   {NBLOCKK, NBLOCKN, KB, NR};

C dims:         {M, N};
C reshaped:     {NBLOCKM, NBLOCKA, MR, NBLOCKN, NR};
C transposed:   {NBLOCKM, NBLOCKN, NBLOCKA, MR, NR};

Tiles are prefetched from DDR to MCDRAM and
reshaped and transposed during transfer.
*/
void top_kernel( const double *A, const double *B,
                 double *C ) {

  prefetch(firstTileOf_A);
  prefetch(firstTileOf_B);
  prefecth(firstTileOf_C);
  for (int i = 0; i < block_number; i++) {
    for (int j = 0; j < block_number; j++) {
      wait(tile of C);
      prefetch(nextTileOf_C)
      for (int k = 0; k < block_number; k++) {
        wait(tileOf_A);
        prefetch(nextTileOf_A);
        wait(tileOf_B);
        prefetch(nextTileOf_B);
        outer_kernel(NBLOCKM, NBLOCKN, NBLOCKK,
          tileOf_A, tileOf_B, tileOf_C);
      }
      wait(flushPreviousTileOf_C);
      flush(tileOf_C);
    }
  }
  wait(flushLastTileOf_C);
}
```

Listing 2. Pseudocode for the DGEMM top kernel.

Inline Transformation Top Kernel: The DGEMM algorithm presented in Section II is adapted to use our framework to handle its tiling as well as the data movement and simultaneous transformations. Data movement is executed ahead of time when possible, using a scheme similar to double buffering. That is, for each matrix, the next block of the outer kernel is prefetched. Result tiles (those of matrix C) are copied back into the source data in DRAM, too. To perform this data movement, dedicated threads each perform active polling on a workqueue protected by a spinlock (one lock per queue per thread). Each thread is in charge of one of the 4 operations in this algorithm: prefetch and transform $\dot{A}$, $\dot{B}$, $\dot{C}$, and flush previous $\dot{C}$. Figures 1, 2, and 3 showcase examples of these transformations between the top and outer kernel, for $MR = 2$, $NR = 3$, $KB = 5$, $NBLOCKA = 7$, $NBLOCKM = 2$, $NBLOCKN = 5$, $NBLOCKK = 4$. Listing 2 provides the pseudocode for this kernel, while detailing the layouts and data movements.

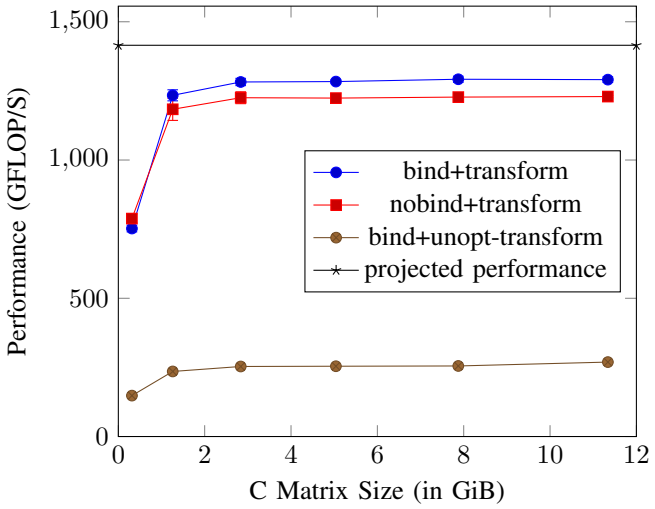


Figure 4. DGEMM performance depending on transform implementation.

Performance Exploration: For our performance exploration we compare several versions of our kernel, depending on two changes:

- Transform performance: whether the implementation uses a generic and unoptimized algorithm to handle each transformation or a custom code, optimized for the specific block sizes involved here,
- Thread binding: whether the threads performing the data transformation share cores with the regular worker threads or use dedicated resources.

Figure 4 presents the results of these experiments, for varying sizes of the result matrix. Experiments are repeated 10 times, and standard error is displayed. We included in the figure the performance of one tile of the outer kernel, as a reference for maximum reachable performance for this strategy. Note that this performance is in line with the reported data in the original study.

These results highlight the impact of the method of transform and the placement of data management threads on overall performance of this algorithm. This is precisely the kind of experiments that are made easier by providing higher-level abstractions about memory management in a complex algorithm. We hope to explore further these design points for this algorithm and others in the future.

V. Related Work

Deep memory architectures have become widely available only in the past couple of years, and studies focusing on them are rare. Furthermore, since vendors recommend using them as another level of hardware-managed cache, few works make the case for explicit management of these memory types. Among existing ones, two major trends can be identified: studies arguing either for *data placement* or for *data migration*.

Data placement [3] addresses the issue of distributing data among all available memory types only once, usually at allocation time. Several efforts in this direction aim at simplifying the APIs available for placement, similar to work on general NUMA architectures: memkind [5], the Simplified Interface for Complex Memory [6], and Hexe [7]. These libraries provide applications with intent-based allocation policies, letting users specify *bandwidth-bound* data or *latency-sensitive* data, for example. While placement is critical for the efficient use of deep memory architectures, these mechanisms lack the means to move data around to accommodate workloads that cannot fit in the right layer, resulting in missing optimization opportunities. Nevertheless, our framework also provides placement features as the basis for its data movement facilities.

Data migration addresses the issue of moving data dynamically across memory types during the execution of the application. Our preliminary work [8] on this approach showcased that performance of a simple stencil benchmark could be improved by migration, using a scheme similar to out-of-core algorithms, when the compute density of the application kernel is high enough to provide compute/migration overlapping. Further work [9] studied performance models for such strategies, including heuristics to migrate data as part of the execution of a workflow with task dependencies. The prefetching strategy used in this paper matches some of these heuristics. Another study [10] discussed a runtime method to schedule tasks with data dependencies on a deep memory platform. Unfortunately, the scheduling algorithm is limited to scheduling a task only after all its input data has been moved to faster memory.

VI. Conclusion

We presented in this paper how additional memory management abstractions can be used to simplify and extend complex optimization strategies for deep memory and heterogeneous platforms. While we used Intel's Knights Landing architecture as a basis for this study, we expect that exascale topologies will exhibit the same kind of complexity and will require the same kind of optimization strategies. In particular, heterogeneous platforms are ideal for such strategies since a separate compute element can be used for transform operations.

Further tuning of the various abstractions can also be performed, in particular autotuning of the transform operators. As we move closer to exascale, we will also continue to improve the abstractions offered by our framework for future architectures. These improvements will include support for heterogeneous platforms (CPU-GPU with unified memory), as well as better abstractions to handle the distribution of data layouts over multiple NUMA nodes or the use of helper cores to perform data movement, which might be necessary for the Fujitsu A64FX Post-K computer architecture.

The AML library, documentation, and links to the benchmarks are available online at https://argo-aml.readthedocs.io/en/latest/.

Acknowledgments

Argonne National Laboratory's work was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computer Research, under Contract DE-AC02-06CH11357. This research was supported by the Exascale

Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

REFERENCES

[1] A. Sodani, "Knights Landing (KNL): 2nd generation Intel® Xeon Phi processor," in *27th IEEE Hot Chips Symposium (HCS)*, 2015.

[2] R. Lim, Y. Lee, R. Kim, and J. Choi, "An implementation of matrix–matrix multiplication on the Intel KNL processor with AVX-512," *Cluster Computing*, June 2018. [Online]. Available: https://doi.org/10.1007/s10586-018-2810-y

[3] D. Unat, J. Shalf, T. Hoefler, T. Schulthess, A. Dubey, and others (Eds.), "Programming abstractions for data locality," Tech. Rep., 04 2014.

[4] B. Videau, K. Pouget, L. Genovese, T. Deutsch, D. Komatitsch, F. Desprez, and J.-F. Méhaut, "BOAST: A metaprogramming framework to produce portable and efficient computing kernels for HPC applications," *International Journal of High Performance Computing Applications*, vol. 32, no. 1, pp. 28–44, Jan. 2018. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01620778

[5] Intel Corporation, "Memkind: A user extensible heap manager," https://memkind.github.io/, 2018.

[6] "Simplified interface to complex memory," https://github.com/lanl/SICM, 2017.

[7] L. Oden and P. Balaji, "Hexe: A toolkit for heterogeneous memory management," in *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2017.

[8] S. Perarnau, J. A. Zounmevo, B. Gerofi, K. Iskra, and P. Beckman, "Exploring data migration for future deep-memory many-core systems," in *IEEE International Conference on Cluster Computing (CLUSTER)*, 2016.

[9] A. Benoit, S. Perarnau, L. Pottier, and Y. Robert, "A performance model to execute workflows on high-bandwidth-memory architectures," in *Proceedings of the 47th International Conference on Parallel Processing, (ICPP)*, 2018.

[10] K. Chandrasekar, X. Ni, and L. V. Kalé, "A memory heterogeneity-aware runtime system for bandwidth-sensitive HPC applications," in *IEEE Int. Parallel and Distributed Processing Symposium Workshops, Orlando, FL, USA*, 2017, pp. 1293–1300. [Online]. Available: https://doi.org/10.1109/IPDPSW.2017.168

[11] A. Haidar, S. Tomov, K. Arturov, M. Guney, S. Story, and J. Dongarra, "LU, QR, and Cholesky factorizations: Programming model, performance analysis and optimization techniques for the Intel Knights Landing Xeon Phi," in *IEEE High Performance Extreme Computing Conference (HPEC)*, 2016.

[12] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst, "LIBXSMM: Accelerating small matrix multiplications by runtime code generation," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016.

[13] *Intel Math Kernel Library. Reference Manual*, 2018. [Online]. Available: https://software.intel.com/en-us/articles/mkl-reference-manual

[14] K. Kim, T. B. Costa, M. Deveci, A. M. Bradley, S. D. Hammond, M. E. Guney, S. Knepper, S. Story, and S. Rajamanickam, "Designing vector-friendly compact BLAS and LAPACK kernels," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017.

[15] L. Dagum and R. Menon, "OpenMP: An industry-standard API for shared-memory programming," *IEEE Comput. Sci. Eng.*, 1998.

[16] I. Z. Reguly, G. R. Mudalige, and M. B. Giles, "Beyond 16GB: Out-of-core stencil computations," in *Proceedings of the Workshop on Memory Centric Programming for HPC*, 2017.

[17] L. Alvarez, M. Casas, J. Labarta, E. Ayguade, M. Valero, and M. Moreto, "Runtime-guided management of stacked DRAM memories in task parallel programs," in *Proceedings of the 2018 International Conference on Supercomputing (ICS)*, Beijing, China, 2018. [Online]. Available: http://ics2018.ict.ac.cn/essay/ICS18-Paper130.pdf

[18] L. Alvarez, M. Moreto, M. Casas, E. Castillo, X. Martorell, J. Labarta, E. Ayguade, and M. Valero, "Runtime-guided management of scratchpad memories in multicore architectures," in *Proceedings of the International Conference on Parallel Architecture and Compilation (PACT)*, 2015.

[19] C. Rosales, J. Cazes, K. Milfeld, A. Gómez-Iglesias, L. Koesterke, L. Huang, and J. Vienne, "A comparative study of application performance and scalability on the Intel Knights Landing processor," in *ISC Workshops*, 2016.

[20] D. Doerfler, J. Deslippe, S. Williams, L. Oliker, B. Cook, T. Kurth, M. Lobet, T. Malas, J.-L. Vay, and H. Vincenti, "Applying the roofline performance model to the Intel Xeon Phi Knights Landing processor," in *International Conference on High Performance Computing*, 2016.

[21] C. Pohl, "Stream processing on high-bandwidth memory," in *Grundlagen von Datenbanken*, 2018.

[22] N. Butcher, S. L. Olivier, J. Berry, S. D. Hammond, and P. M. Kogge, "Optimizing for KNL usage modes when data doesn't fit in MCDRAM," in *International Conference on Parallel Processing*, 2018. [Online]. Available: http://par.nsf.gov/biblio/10064736

[23] (2018) OpenBLAS: An optimized BLAS library. [Online]. Available: https://www.openblas.net/

[24] J. Dongarra, S. Hammarling, N. J. Higham, S. D. Relton, P. Valero-Lara, and M. Zounon, "The design and performance of batched BLAS on modern high-performance computing systems," *Procedia Computer Science*, vol. 108, pp. 495–504, 2017, International Conference on Computational Science (ICCS). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050917307056

[25] R. Asai, "Clustering modes in Knights Landing processors: Developer's guide," Colfax International, Tech. Rep., 05 2016.

[26] A. Vladimirov and R. Asai, "MCDRAM as high-bandwith memory (HBM) in Knights Landing processors: Developer's guide," Colfax International, Tech. Rep., 05 2016.

[27] A. J. Pena and P. Balaji, "Toward the efficient use of multiple explicitly managed memory subsystems," in *IEEE Int. Conf. on Cluster Computing (CLUSTER)*, 2014, pp. 123–131.

[28] H. Servat, A. J. Peña, G. Llort, E. Mercadal, H. Hoppe, and J. Labarta, "Automating the application data placement in hybrid memory systems," in *IEEE International Conference on Cluster Computing, (CLUSTER)*, 2017, pp. 126–136. [Online]. Available: https://doi.org/10.1109/CLUSTER.2017.50

[29] G. Voskuilen, A. F. Rodrigues, and S. D. Hammond, "Analyzing allocation behavior for multi-level memory," in *Proceedings of the Second International Symposium on Memory Systems, (MEMSYS)*, 2016, pp. 204–207. [Online]. Available: http://doi.acm.org/10.1145/2989081.2989116

[30] NVIDIA, "CUDA: Unified memory programming," http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd, 2018.

[31] R. Landaverde, T. Zhang, A. K. Coskun, and M. Herbordt, "An investigation of unified memory access performance in CUDA," in *IEEE High Performance Extreme Computing Conference (HPEC)*, 2014.

[32] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst, "Data-aware task scheduling on multi-accelerator based platforms," in *IEEE Int. Conf. on Parallel and Distributed Systems*, Dec 2010, pp. 291–298.

Machine Learning Guided Optimal Use of GPU Unified Memory

Hailu Xu
Florida International University
Miami, FL, USA
hxu017@fiu.edu

Murali Emani
Argonne National Laboratory
Lemont, IL, USA
memani@anl.gov

Pei-Hung Lin
Lawrence Livermore National Laboratory
Livermore, CA, USA
lin32@llnl.gov

Liting Hu
Florida International University
Miami, FL, USA
lhu@cs.fiu.edu

Chunhua Liao
Lawrence Livermore National Laboratory
Livermore, CA, USA
liao6@llnl.gov

ABSTRACT

NVIDIA's unified memory (UM) creates a pool of managed memory on top of physically separated CPU and GPU memories. UM automatically migrates page-level data on-demand so programmers can quickly write CUDA codes on heterogeneous machines without tedious and error-prone manual memory management. To improve performance, NVIDIA allows advanced programmers to pass additional memory use hints to its UM driver. However, it is extremely difficult for programmers to decide when and how to efficiently use unified memory, given the complex interactions between applications and hardware. In this paper, we present a machine learning-based approach to choosing between discrete memory and unified memory, with additional consideration of different memory hints. Our approach utilizes profiler-generated metrics of CUDA programs to train a model offline, which is later used to guide optimal use of UM for multiple applications at runtime. We evaluate our approach on NVIDIA Volta GPU with a set of benchmarks. Results show that the proposed model achieves 96% prediction accuracy in correctly identifying the optimal memory advice choice.

CCS CONCEPTS

• **Computer systems organization → Heterogeneous (hybrid) systems**; • **Computing methodologies → Machine learning**.

KEYWORDS

Unified memory, GPU, Data allocation, Machine learning

ACM Reference Format:
Hailu Xu, Murali Emani, Pei-Hung Lin, Liting Hu, and Chunhua Liao. 2019. Machine Learning Guided Optimal Use of GPU Unified Memory. In *MCHPC '19: Workshop on Memory Centric High Performance Computing, November 18, 2019, Denver, Colorado, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/1122445.1122456

MCHPC '19, November 18, 2019, Denver, Colorado, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Graphic Processing Units (GPUs) have been the fundamental hardware components for supporting high performance computing, artificial intelligence, and data analysis in datacenters and supercomputers. As of June 2019, 133 of Top 500 supercomputers are GPU-accelerated, and 128 systems debuting on the TOP500 are accelerated with NVIDIA GPUs [16]. Efficient memory management across CPUs and GPUs has been a challenging problem, while it is critical to performance and energy efficiency. Before CUDA 6.0, data shared by CPUs and GPUs is allocated in discrete memories, which require explicit memory copy calls to transfer data between CPUs and GPUs. Since CUDA 6.0, NVIDIA has introduced unified Memory (UM) with a single unified programmable memory place within a heterogeneous CPU-GPU architecture consisting of separated physical memory spaces. UM relieves programmers from manual management of data migration between CPUs and GPUs such as inserting memory copy calls and deep copying pointers. It tremendously improves productivity and also enables oversubscribing GPU memory.

NVIDIA continuously improves UM throughout different generations of GPUs. The latest UM implementation has accumulated a rich set of features including GPU page fault, on-demand migration, over-subscription of GPU memory, concurrent access and atomics, access counters, and so on. Moreover, NVIDIA provides the `cudaMemAdvice`[1] API to advise the UM driver about the usage pattern of memory objects (e.g. dynamically allocated arrays). Different hints (such as `ReadMostly`, `PreferredLocation`, `AccessedBy`) can be specified in this API by programmers to improve the performance of UM. However, it is extremely challenging for programmers to decide when and how to efficiently use UM for various kinds of applications. For a given memory object, there is a wide range of choices including managing it with the traditional discrete memory API, the unified memory API without advice, and the unified memory API combined with various memory hints.

In this paper, we present a novel approach to choosing between discrete memory and unified memory on GPUs, with additional consideration of different memory usage hints. Our approach consists of two phases: an offline learning phase and an online inference phase. 1) The offline learning phase involves building a classifier via

[1]Nvidia CUDA Runtime API (May 2019) https://docs.nvidia.com/cuda/cuda-runtime-api/index.html

supervised learning. It first collects the runtime GPU kernel features from selected benchmarks and labels the best advice based upon the performance caused by various GPU memory usage choices. After that, it constructs a classifier that can predict the best advice for new applications. 2) The online inference relates to determining the proper advice at runtime for a running CUDA program. By combining offline learning and online inference, our method can effectively and accurately obtain optimal use of GPU memory for different kinds of CUDA applications and presents fine-grain control over managed memory allocations.

This paper makes the following contributions:

- We study the hybrid use of both discrete and unified memory APIs on GPUs, with additional consideration for selecting different memory advice choices.
- A machine learning-based approach is proposed to guide optimal use of GPU unified memory.
- We design code transformation to enable runtime adaptation of CUDA programs leveraging online inference decisions.
- We incorporate kernel features at runtime to provide fine-grain control over GPU memory.
- Our experiments show that our approach is effective to predict the optimal memory advice choices for the selected benchmarks.

The remainder of this paper is organized as follows: Section 2 presents the background of GPU memory and our motivation. Section 3 describes the details of our design and methodology. Section 4 presents the experimental results of evaluation in GPU. We discuss the related works in Section 5 and conclude this work in Section 6.

2 BACKGROUND AND MOTIVATION

2.1 Choices for Using GPU Memory

Programmers often encounter multiple choices to manage their data on GPU memory. NVIDIA's CUDA traditionally exposes GPU device memory as a discrete memory space from CPU memory space. Programmers are responsible for using a set of memory API functions to explicitly manage the entire life cycle of data objects stored in GPU memory, including allocation, de-allocation, data copying, etc. Since CUDA 6.0, NVIDIA has introduced unified Memory (UM) with a new set of API functions. The idea of UM is to present developers a single memory space unifying both CPU and GPU memories. CUDA uses a unified memory driver to automatically migrate data between CPU and GPU memories at runtime. As a result, UM significantly improves the productivity of GPU programming. Both traditional memory APIs and unified memory APIs can be used together within a single CUDA program.

To enable better performance of UM, CUDA allows developers to give the UM driver additional advice on managing a given GPU memory range via an API function named `cudaMemAdvise(const void *, size_t, enum cudaMemoryAdvise, int)`. The first two parameters of this function accept a *pointer* to a memory range with a specified size. The memory range should be allocated via `cudaMallocManaged` or declared via `__managed__variables`. The third parameter sets the advice for the memory range. The last parameter indicates the associated device's id, which can indicate either a CPU or GPU device. The details and differences of these four kinds of advice are presented as follows:

- `Default`: This represents the default on-demand page migration to accessing processor, using the first-touch policy.
- `cudaMemAdviseSetReadMostly`: This advice is used for the data which is mostly going to be read from and only occasionally written to. The UM driver may create read-only copies of the data in a processor's memory when that processor accesses it. If this region encounters any write requests, then only the write occurred page will be valid and other copies will be invalid.
- `cudaMemAdviseSetPreferredLocation`: Once a target device is specified, this device memory can be set as the preferred location for the allocated data. The host memory can also be specified as the preferred location. Setting the preferred location does not cause data to migrate to that location immediately. The policy only guides what will happen when a fault occurs on the specified memory region: if data is already in the preferred location, the faulting processor will try to directly establish a mapping to the region without causing page migration. Otherwise, the data will be migrated to the processor accessing it if the data is not in the preferred location or if a direct mapping cannot be established.
- `cudaMemAdviseSetAccessedBy`: This advice implies that the data will be accessed by a specified CPU or GPU device. It has no impact on the data location and will not cause data migration. It only causes the data to be always mapped in the specified processor's page tables, when applicable. The mapping will be accordingly updated if the data is migrated somehow. This advice is useful to indicate that avoiding faults is important for some data, especially when the data is accessed by a GPU within a system containing multiple GPUs with peer-to-peer access enabled.

The effect of `cudaMemAdvise` can be reverted with the following options: `UnsetReadMostly`, `UnsetPreferredLocation`, and `UnsetAccessedBy`.

2.2 Impact of Different Usage of GPU Memory

Various applications have diverse data access patterns throughout their executions. Different choices of memory APIs and their parameter values often result in a wide variation in performance. To explore the impact of various memory usage choices, we modify several benchmarks from Rodinia [3] to use different memory allocations and advice choices and subsequently examine their execution times. We focus on large dynamically allocated data objects since they usually have major impact on execution time.

Table 1 lists a subset of various code variants for the `gaussian` benchmark in Rodinia. There are two matrices `a`, `m` and one array `b` which are the major data objects in `gaussian`. We apply different memory usage choices to these objects and get multiple combinations. Code variant 1 is the baseline version using the default discrete memory for the three data objects. Variant 2 to 7 use unified memory for matrix `a` and different memory advise of `AccessedBy`, `ReadMostly`, and `PreferredLocation`. We can further specify CPU or GPU as the device for `AccessedBy` and `PreferredLocation`.

We evaluate the performance of all the code variants and present results in Figure 2. The input data is a 1024×1024 matrix and the measurement includes all the memory transferring between

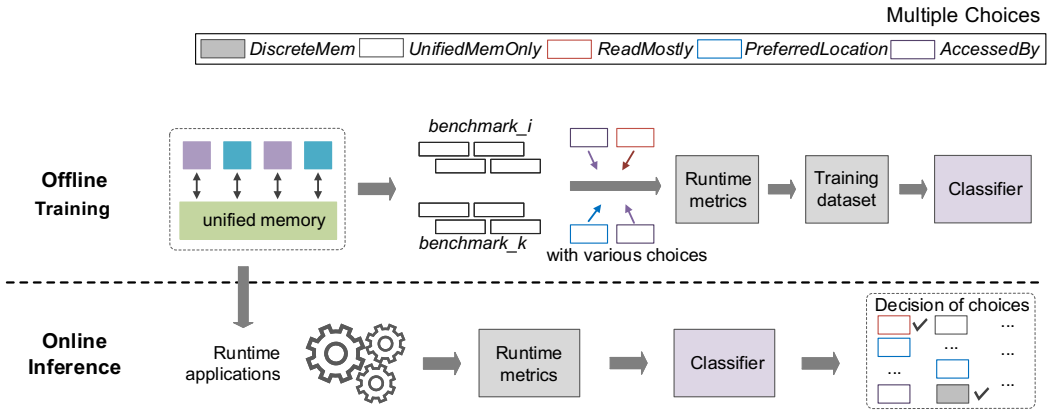


Figure 1: The workflow of the proposed approach in guiding the GPU unified memory advice.

Variants	Description
1	baseline using discrete memory for all objects
2	modified to use unified memory for all
3	set array a with the `ReadMostly` advice
4	set array a with the `PreferredLocation` on GPU
5	set array a with the `AccessedBy` for GPU
6	set array a with the `PreferredLocation` on CPU
7	set array a with the `AccessedBy` for CPU

Table 1: Code variants in the `gaussian` benchmark

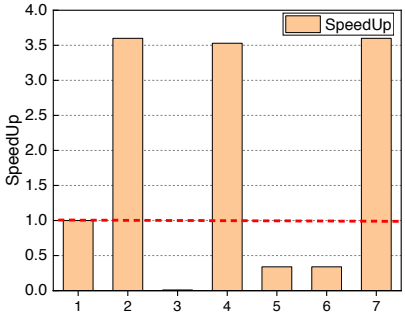


Figure 2: Speedup of different code variants in `gaussian`

the CPU memory and the GPU device memory. It is shown that the code achieves a speedup of 3.5× when matrix a is given the `PreferredLocation` to GPU (variant 4). A 200× performance degradation is observed when the `ReadMostly` (variant 3) advice is wrongfully given to matrix a. This experiment demonstrates the significant performance impact of choosing the right memory usage of GPU memory.

3 DESIGN

CUDA allows programmers to use either discrete memory or unified memory APIs, potentially combined with different kinds of advice, to manage memory objects in one application or benchmark. Applications and benchmarks can be deployed with various combinations of these choices at different granularity, such as program-level, kernel-level, or object-level. The coarse-grain memory usage optimization is easy to implement but may not deliver the best performance gain. On the other hand, fine-grain memory usage optimization involves carefully deciding a choice for each object of each kernel, even with the consideration of the kernel's calling context. This is challenging to implement but may result the best performance improvements.

We limit the scope of this work to be finding optimal memory usage choices at the object level, i.e., different memory usage choices are used for different data objects for a given kernel function under all calling context. For example, if a program has two kernel functions and both refer to two data objects during the execution, we can assign the default unified memory choice for the first data object for the first kernel function, we then assign the UM combined with the `ReadMostly` advice for the other data object. We can re-assign different choices for the data objects for the next kernel.

3.1 Approach

We are developing a machine learning framework to automatically decide the optimal choice of GPU memory usage for CUDA applications. The framework has a two-phase workflow: offline learning and online inference. As shown in Figure 1, the offline learning phase uses code variants using different GPU memory usage choices, to collect a set of runtime profiling metrics via Nsight CUDA Profiler[2]. The best performing versions are identified and labeled. We then use the collected data as a training data set to construct a classifier model via supervised learning. We explore different kinds of machine learning classifiers such as Random Forest, Random Tree, LogitBoost and select the one that yields highest accuracy and F1 measures. The online inference phase uses Nsight to collect runtime metrics of running applications and passes the input feature vector to the learned model to guide the runtime GPU

[2]Nvidia Compute Command Line Interface https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html

memory usage choices for various applications. We elaborate the two phases in the following subsections.

3.2 Offline Learning

In this offline learning phase, we design training configurations that help to capture diverse memory usage variants. Running these experiments will yield the raw training data to train the classifiers.

Training Benchmarks. We manually prepare several variants of selected benchmarks and execute them to find the best performing variant, which is then labelled for supporting training later. Rodinia benchmark [3] is selected to implement different memory usage choices for selected arrays or data structures. Figure 3 presents an example using unified memory and different cudaMemAdvise() settings for two arrays (`a` and `b`) in `gaussian` benchmark. `xplacer_malloc()` is a wrapper function we introduce to switch between discrete or unified memory version of CUDA memory allocation.

```
...
//Modified to utilize unified memory in xplacer_malloc()
a = (float *) xplacer_malloc(Size*Size* sizeof(float), Managed);
b = (float *) xplacer_malloc(Size* sizeof(float), Managed);
...
//Assign unified memory advice for specific data object
#if advOptionA == 1
cudaMemAdvise(a, Size*Size* sizeof(float), cudaMemAdviseSetReadMostly, 0);
#elif advOptionA == 2
cudaMemAdvise(a, Size*Size* sizeof(float),
        cudaMemAdviseSetPreferredLocation, 0);
...
#endif
#if advOptionB == 1
cudaMemAdvise(b, Size* sizeof(float), cudaMemAdviseSetAccessedBy, 0);
#elif advOptionB == 2
cudaMemAdvise(b, Size* sizeof(float), cudaMemAdviseSetPreferredLocation, 0);
#endif
...
ForwardSub(); //Run the kernel function
```

Figure 3: Code showing `gaussian` benchmark using unified memory with memory advice.

Feature Engineering. We utilize the `Nsight Compute` command line profiler to fetch detailed runtime performance metrics of the benchmarks. We implement the data collection on machines using Tesla V100 GPUs. `Nsight Compute` provides metrics organized within different sections. Each section focuses on a specific part of the kernel analysis. The default profiling phase contains 8 sections, including GPU Speed Of Light, Compute Workload Analysis, Memory Workload Analysis, Scheduler Statistics, Warp State Statistics, Instruction Statistics, Launch Statistics and Occupancy. We collect a total of 49 non-zero-valued metrics that correspond to these sections. We then utilize feature correlation and information gain techniques to remove the redundant features. The remaining 9 useful features are listed in Table 2.

Model Training. We evaluate multiple classical machine learning classification algorithms with the collected data. These models include classifiers such as Random Forest, Random Tree and Decision Tree. To guarantee the robustness of our model, we use 10-fold cross validation to verify the model's performance and also ensure

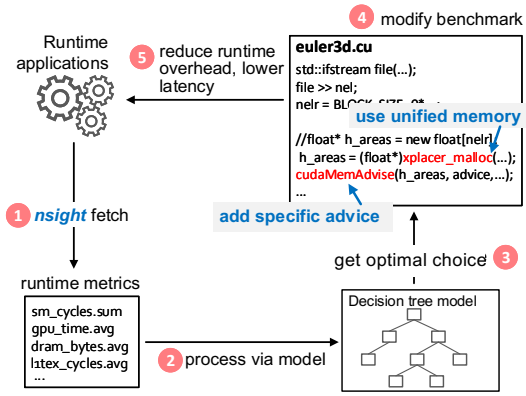


Figure 4: Workflow of the online inference.

No.	Feature Name
1	Elapsed Cycles
2	Duration
3	SM Active Cycles
4	Memory Throughput
5	Max Bandwidth
6	Avg. Execute Instructions Per Scheduler
7	Grid Size
8	Number of Threads
9	Achieved Active Warps Per SM

Table 2: List of selected features in the model.

the model performs evaluation on unseen data. We rely on model's prediction accuracy as the metric of evaluation since the GPU memory choice determined by this model has direct correlation to the program execution time.

3.3 Online Inference

The online inference consists of five major steps, as shown in Figure 4. First, we fetch the runtime metrics from the running applications with the Nsight profiler. Next, a feature vector is composed after normalizing these metrics, which is then passed as input to the offline trained model. The model will then output its predicted memory advice for each of the kernel instance. Once the new predicted choice is given, as shown in the fourth step, we then modify the original application code to implement the optimal choice of memory usage. Modifications to the source code can be automated in the future by a source-to-source tool if available or by a library support to switch among the memory advises. Finally, the optimized code is run with the corresponding memory advice.

4 EVALUATION

4.1 Experiment Settings

We evaluate our approach with multiple benchmarks running on the Lassen supercomputer at Livermore Computing [8]. Each compute node of Lassen has two IBM Power9 CPUs and four Tesla V100 GPUs.

We selected four benchmarks from Rodinia [3] for our evaluation. They are *Computational Fluid Dynamics Simulation* (CFD), *Breadth-first Search* (BFS), *Gaussian Elimination* (Gaussian), and *HotSpot* as shown in Table 3. All the variants are generated by two options of flags (cudaMemAttachGlobal or cudaMemAttachHost) given to data allocation by cudaMallocManaged API, and six memory advise options (no advise, `ReadMostly`, `PreferredLocation` for GPU, `PreferredLocation` for CPU, `AccessedBy` GPU, and `AccessedBy` CPU) for each kernel in the benchmark. For example, we specify memory advises to three arrays in CFD benchmark. The overall number of variants become 432 (2×6×6×6). There are six arrays used in one GPU kernel for the BFS benchmark with a total of 93312 (2×6^6) variants. We reduce the variant number down to only 84 (2×6×6) by only specifying advise to one array in a variant. Note that we use large number of variants to extract the different runtime metrics. When using in the training and prediction, we category these variants by program-level advice based on the most common advice among them with minimal execution time. We use the default input data provided by the Rodinia benchmark suite and generate additional input data sets for HotSpot and Gaussian following the instructions given by the Rodinia suite.

	Kernels	Arrays	Variants	Input data set
CFD	4	3	(2×6×6×6)	3
BFS	2	6	(2×6×6)	3
Gaussian	2	3	(2×6×6×6)	67
HotSpot	1	2	(2×6×6)	8

Table 3: Benchmarks for experiments

4.2 Preliminary Results

We collect total 2,753 instances for training data. After normalization and reformatting, the data is made into one single dataset. We split them into ten subsets, train the evaluated models with many algorithms based on nine of ten, and the final subset is used for examining the predictions of advice from the trained models.

We evaluate the collected data with various classifiers and display the F-measure scores in Figure 5 compared against ground truth which are the memory advice that yield best performance. Note that F-measure score illustrates the harmonic mean of the fraction of correctly predicted advice in all predicted results and the fraction of correctly identified advice in the original results.

The measured values are across all the evaluated benchmarks with various input data set sizes. It can be observed that when implementing the model with a Random Forest classifier, it achieves the best performance with F-measure up to 96.3%. These results establish that our approach can effectively predict optimal choices for the benchmarks. The model is generic and portable across different applications and input data set sizes and thus demonstrates the potential use in guiding the optimal memory choices.

Fig. 6 shows performance comparisons between the execution time from the original benchmarks which is the baseline for this evaluation and from the codes with the predicted memory advice. All the selected benchmarks with the predicted memory advice achieve equivalent or better performance compared to the original benchmarks. The model can thus effectively assist to achieve better performance for the selected benchmarks.

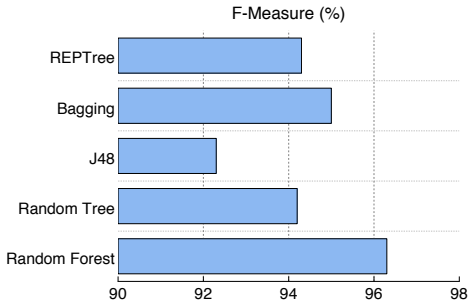


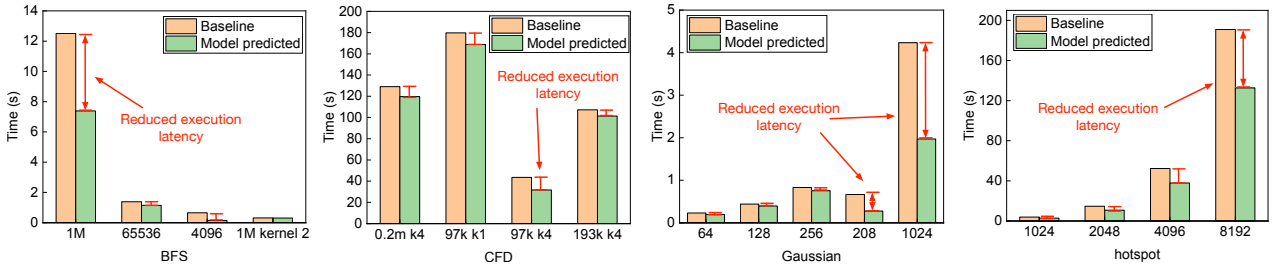
Figure 5: Evaluation of the model prediction accuracies of different classifiers.

5 RELATED WORK

Numerous studies have explored optimization strategies to place data within the various types of memories and caches of GPUs, without considering unified memory. For example, PORPLE [4, 5] is a portable approach using a lightweight performance model to guide run-time selection of optimal data placement policies. Huang and Li [6] have analyzed correlations among different data placements and used a sample data placement to predict performance for other data placements. Jang et al. [7] have presented several rules based on data access patterns to guide the memory selection for a Tesla GPU. Yang et al. [18] proposed compiler-based approach to generate kernel variants for exploiting memory characteristics. More recently, Stoltzfus et al. [15] designed a machine learning approach for guiding data placement using offline profiling and online inference on Volta GPUs. Bari et al. [2] studied the impact of data placement on newer generations of GPUs such as Volta.

Many applications had been implemented with the unified memory in the high performance computing areas to reduce the complexities of memory management [12]. An investigation of early implementation of unified memory [9] showed that applications did not perform well in most cases due to high overhead caused by CUDA 6. Sakharnykh [14] presented a comprehensive overview of unified memory on three generations of GPUs (Kepler, Pascal and Volta), with a few application studies using advanced UM features. Awan et al. [1] exploited advanced unified memory features in CUDA 9 and Volta GPUs for out-of-core DNN training. They observed minor performance degradation in OC-Caffe with the help of memory prefetching and advise operations.

Unified memory is also studied under the context of OpenACC and OpenMP. OpenARC [10] is an OpeACC compiler with extensions to support unified memory. They found that unified memory is beneficial if only a small random portion of data is accessed. Wolfe et al. [17] studied how the data model is supported in several OpenACC implementations. They mentioned some implementations were able to use unified memory. Mishra et al [13] evaluated unified memory for OpenMP GPU offloading. They reported that the UM performance was competitive for benchmarks with little data reuse while it incurred significant overhead for large amount data reuse with memory over-subscription. Li et al. [11] proposed a compiler-runtime collaborative approach to optimize GPU data under unified memory within an OpenMP implementation. Static and runtime analysis are used to collect data access properties to



(a) Execution time of BFS. **(b)** Execution time of CFD. **(c)** Execution time of gaussian. **(d)** Execution time of hotspot.

Figure 6: The comparison of execution times between baseline benchmarks and model predicted performances.

guide if data should be placed on CPU or GPU memory, and how to transfer the data (explicitly through traditional memory copy operations vs. implicitly through UM) if mapped to GPU.

6 CONCLUSION & FUTURE WORK

In this paper, we present a novel machine learning-based approach which can guide the optimal use of unified memory of GPUs for various applications at runtime. It consists of two phases: offline learning and online inference. After collecting and filtering the offline metrics from multiple benchmarks, we train a machine learning model based on remaining useful metrics. We then use the trained model to guide the online execution of applications, by predicting the optimal memory choices for each kernel based on its runtime metrics. The experimental results show that given a set of CUDA benchmarks, the proposed approach is able to accurately determine what kind of memory choices are optimal: either in discrete memory or unified memory space (and combined with various memory advice hints). It alleviates the burden on application developers by automating the complex decision making process which otherwise would require extensive, time-consuming experiments.

In the future, we will extend this work to evaluate the advice choices at a finer granularity considering calling context. Second, using collaborative compiler and runtime support, we will employ runtime code generation and/or adaptation techniques to automatically generate codes using suggested optimal memory choices. Third, we will evaluate the overhead for collecting training data and investigate how to reduce the overhead. Last but not least, the model will be applied to more hardware platforms.

ACKNOWLEDGMENTS

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 and supported by LLNL-LDRD 18-ERD-006. This research was funded in part by the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. LLNL-CONF-793704.

REFERENCES

[1] Ammar Ahmad Awan, Ching-Hsiang Chu, Hari Subramoni, Xiaoyi Lu, and Dhabaleswar K Panda. 2018. OC-DNN: Exploiting Advanced Unified Memory Capabilities in CUDA 9 and Volta GPUs for Out-of-Core DNN Training. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*. IEEE, 143–152.

[2] M Bari, Larisa Stoltzfus, P Lin, Chunhua Liao, Murali Emani, and Barbara Chapman. 2018. *Is data placement optimization still relevant on newer gpus?* Technical Report. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States).

[3] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 44–54.

[4] Guoyang Chen, Xipeng Shen, Bo Wu, and Dong Li. 2017. Optimizing data placement on GPU memory: A portable approach. *IEEE Trans. Comput.* 66, 3 (2017), 473–487.

[5] Guoyang Chen, Bo Wu, Dong Li, and Xipeng Shen. 2014. PORPLE: An extensible optimizer for portable data placement on GPU. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 88–100.

[6] Yingchao Huang and Dong Li. 2017. Performance modeling for optimal data placement on GPU with heterogeneous memory systems. In *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*. IEEE, 166–177.

[7] Byunghyun Jang, Dana Schaa, Perhaad Mistry, and David Kaeli. 2010. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *IEEE Transactions on Parallel & Distributed Systems* 1 (2010), 105–118.

[8] Lawrence Livermore National Laboratory. 2019. Lassen supercomputer. (2019). https://computing.llnl.gov/computers/lassen

[9] Raphael Landaverde, Tiansheng Zhang, Ayse K Coskun, and Martin Herbordt. 2014. An investigation of unified memory access performance in cuda. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–6.

[10] Seyong Lee and Jeffrey S Vetter. 2014. OpenARC: extensible OpenACC compiler framework for directive-based accelerator programming study. In *Proceedings of the First Workshop on Accelerator Programming using Directives*. IEEE Press, 1–11.

[11] Lingda Li, Hal Finkel, Martin Kong, and Barbara Chapman. 2018. Manage OpenMP GPU Data Environment Under Unified Address Space. In *International Workshop on OpenMP*. Springer, 69–81.

[12] Wenqiang Li, Guanghao Jin, Xuewen Cui, and Simon See. 2015. An evaluation of unified memory technology on nvidia gpus. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 1092–1098.

[13] Alok Mishra, Lingda Li, Martin Kong, Hal Finkel, and Barbara Chapman. 2017. Benchmarking and evaluating unified memory for OpenMP GPU offloading. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*. ACM, 6.

[14] Nikolay Sakharnykh. 2018. Everything You Need to Know About Unified Memory. (2018). http://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf

[15] Larisa Stoltzfus, Murali Emani, Pei-Hung Lin, and Chunhua Liao. 2018. Data Placement Optimization in GPU Memory Hierarchy using Predictive Modeling. In *Proceedings of the Workshop on Memory Centric High Performance Computing*. ACM, 45–49.

[16] Tiffany Trader. 2019. Top500 Purely Petaflops; US Maintains Performance Lead. (June 2019). https://www.hpcwire.com/2019/06/17/us-maintains-performance-lead-petaflops-top500-list/

[17] Michael Wolfe, Seyong Lee, Jungwon Kim, Xiaonan Tian, Rengan Xu, Sunita Chandrasekaran, and Barbara Chapman. 2017. Implementing the OpenACC data model. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 662–672.

[18] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. 2010. A GPGPU Compiler for Memory Optimization and Parallelism Management. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, New York, NY, USA, 86–97. https://doi.org/10.1145/1806596.1806606

SUMMARY OF THE EXPERIMENTS REPORTED

ARTIFACT AVAILABILITY

Software Artifact Availability:

List of URLs and/or DOIs where artifacts are available:

- Evaluation artifact: https://gitlab.com/AndrewXu22/optimal_unified_memory.git
- Original Rodinia Benchmark: http://lava.cs.virginia.edu/Rodinia/download.htm

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: Nvidia GPU Tesla V-100

Operating systems and versions: Debian GNU/Linux 16.04

Compilers and versions: NVCC

Applications and versions: Rodinia Benchmark 3.1

Libraries and versions: CUDA 10.1 with Nsight command line tool

Key algorithms: Random Forest, J48

Input datasets and versions: provided in evaluation artifact

Paper Modifications:

ARTIFACT EVALUATION

The workflow can be performed with the following commands:

```
git clone https://gitlab.com/AndrewXu22/optimal_unified_memory.git
cd optimal_unified_memory
./script/driver.sh
```

UMap : Enabling Application-driven Optimizations for Page Management

Ivy B. Peng*, Marty McFadden*, Eric Green*, Keita Iwabuchi*
Kai Wu†,Dong Li†,Roger Pearce*, Maya B. Gokhale*
*Lawrence Livermore National Laboratory, Livermore, CA, USA
†University of California, Merced, CA, USA
*{peng8, mcfadden8, green77, iwabuchi1, pearce7, gokhale2}@llnl.gov, †{kwu42, dli35}@ucmerced.edu

***Abstract*—Leadership supercomputers feature a diversity of storage, from node-local persistent memory and NVMe SSDs to network-interconnected flash memory and HDD. Memory mapping files on different tiers of storage provides a uniform interface in applications. However, system-wide services like mmap are optimized for generality and lack flexibility for enabling application-specific optimizations. In this work, we present *UMap* to enable user-space page management that can be easily adapted to access patterns in applications and storage characteristics. *UMap* uses the userfaultfd mechanism to handle page faults in multi-threaded applications efficiently. By providing a data object abstraction layer, *UMap* is extensible to support various backing stores. The design of *UMap* supports dynamic load balancing and I/O decoupling for scalable performance. *UMap* also uses application hints to improve the selection of caching, prefetching, and eviction policies. We evaluate *UMap* in five benchmarks and real applications on two systems. Our results show that leveraging application knowledge for page management could substantially improve performance. On average, *UMap* achieved 1.25 to 2.5 times improvement using the adapted configurations compared to the system service.**

***Index Terms*—memory mapping, memmap, page fault, user-space paging, userfaultfd, page management**

I. Introduction

Recently, leadership supercomputers provide enormous storage resources to cope with expanding data sets in applications. The storage resources come in a hybrid format for balanced cost and performance [9], [11], [12]. Fast and small storage, which is implemented using advanced technologies like persistent memory and NVMe SSDs, often co-locate with computing units inside compute node. Storage with massive capacity, on the other hand, uses cost-effective technologies like HDD and is interconnected to compute nodes through the network. In between, burst buffers use fast memory technologies and are accessible through the network. Memory mapping provides a uniform interface to access files on different types of storage as if to dynamically allocated memory. For instance, out-of-core data analytic workloads often need to process large datasets that exceed the memory capacity of a compute node [17]. Using memory mapping to access these datasets shift the burden of paging, prefetching, and caching data between storage and memory to the operating systems.

Currently, operating systems provide the *mmap* system call to map files or devices into memory. This system service performs well in loading dynamic libraries and could also support out-of-core execution. However, as a system-level service, it has to be tuned for performance reliability and consistency over a broad range of workloads. Therefore, it may reduce opportunities in optimizing performance based on application characteristics. Moreover, backing stores on different storage exhibit distinctive performance characteristics. Consequently, configurations tuned for one type of storage will need to be adjusted when mapping on another type of storage. In this work, we provide *UMap* to enable application-specific optimizations for page management in memory mapping various backing stores. *UMap* is highly configurable to adapt user-space paging to suit application needs. It facilitates application control on caching, prefetching, and eviction policies with minimal porting efforts from the programmer. As a user-level solution, *UMap* confines changes within an application without impacting other applications sharing the platform, which is unachievable in system-level approaches.

We prioritize four design choices for *UMap* based on surveying realistic use cases. First, we choose to implement *UMap* as a user-level library so that it can maintain compatibility with the fast-moving Linux kernel without the need to track and modify for frequent kernel updates. Also, we employ the recent userfaultfd [7] mechanism, other than the signal handling + callback function approach to reduce overhead and performance variance in multi-threaded applications. Third, we target an adaptive solution that sustains performance even at high concurrency for data-intensive applications, which often employ a large number of threads for hiding data access latency. Our design pays particular consideration on load imbalance among service threads to improve the utilization of shared resources even when data accesses to pages are skewed. *UMap* dynamically balances workloads among all service threads to eliminate bottleneck on serving hot pages. Finally, for flexible and portable tuning on different computing systems, *UMap* provides both API and environmental controls to enable configurable page sizes, eviction strategy, application-specific prefetching, and detailed diagnosis information to the programmer.

We evaluate the effectiveness of *UMap* in five use cases, including two data-intensive benchmarks, i.e., a synthetic sort benchmark and a breadth-first search (BFS) kernel, and three real applications, i.e., Lrzip [8], N-Store database [2], and an asteroid detection application that processes massive data

sets from telescopes. We conduct out-of-core experiments on two systems with node-local SSD and network-interconnected HDD storage. Our results show that *UMap* can enable flexible user-space page management in data-intensive applications. On the AMD testbed with local NVMe SSD, applications achieved 1.25 to 2.5 times improvement compared to the standard system service. On the Intel testbed with network-interconnected HDD, *UMap* brings the performance of the asteroid detection application close to that uses local SSD for 500 GB data sets. In summary, our main contributions are as follows:

- We propose an open-source library[1], called *UMap* that leverages lightweight userfaultfd mechanism to enable application-driven page management.
- We describe the design of *UMap* for achieving scalable performance in multi-threaded data-intensive applications.
- We demonstrate five use cases of *UMap* and show that enabling configurable page size is essential for performance tuning in data-intensive applications.
- *UMap* improves the performance of tested applications by 1.25 to 2.5 times compared to the standard mmap system service.

II. Background and Motivation

In this section, we introduce memory mapping, prospective benefits from user-space page management, and the enabling mechanism *userfaultfd*.

A. Memory Mapping

Memory mapping links images and files in persistent storage to the virtual address space of a process. The operating system employs demand paging to bring only accessed virtual pages into physical memory because virtual memory can be much larger than physical memory. An access to memory-mapped regions triggers a page fault if no page table entry (PTE) is present for the accessed page. When such a page fault is raised, the operating system resolves it by copying in the physical data page from storage to the in-memory page cache.

Common strategies for optimizing memory mapping in the operating systems include page cache, read-ahead, and madvise hints. The page cache is used to keep frequently used pages in memory while less important pages may need to be evicted from memory to make room for newly requested pages. Least Recently Used (LRU) policy is commonly used for selecting pages to be evicted. The operating system may proactively flush dirty pages, i.e., modified pages in the page cache, into storage when the ratio of dirty page exceeds a threshold value [19]. Read-ahead preloads pages into physical memory to avoid the overhead associated with page fault handling, TLB misses and user-to-kernel mode transition. Finally, the madvise interface takes hints to allow the operating system to make informed decisions for managing pages.

[1]UMAP v2.0.0 https://github.com/LLNL/umap.

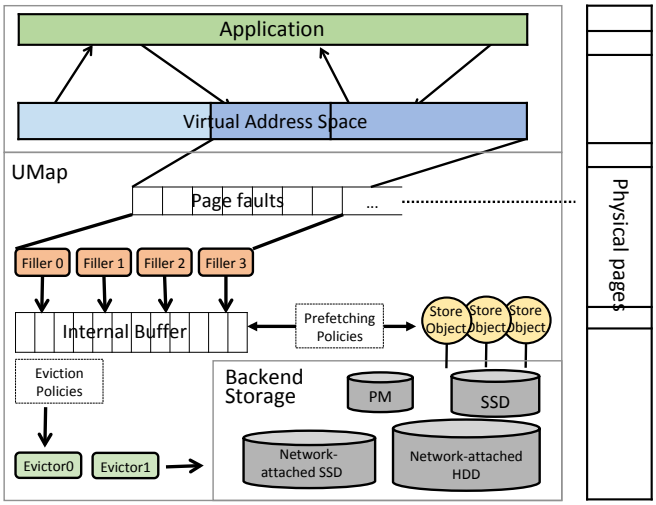


Fig. 1: The *UMap* architecture.

B. User-space Page Management

User-space page management uses application threads to resolve page faults and manage virtual memory in the background as defined by the application. The userfaultfd is a lightweight mechanism to enable user-space paging compared to the traditional *SIGSEGV* signal and callback function [7]. Applications register address ranges to be manged in user-space, and specify the type of events, e.g., page faults and events in un-cooperative mode, to be tracked. Page faults in the address ranges are delivered asynchronously so that the faulting process is blocked instead of idling, allowing other processes to be scheduled to proceed.

The fault-handling thread in the application can atomically resolve page faults with the *UFFDIO_COPY* ioctl, which ensures the faulting process is (optionally) waken up only after the requested page has been fully copied into physical memory [7]. The fault-handling threads may utilize application-specific knowledge to optimize this procedure, providing the flexibility that is unachievable in kernel mode. For instance, the application could select arbitrary page sizes, read-ahead window size, or provides specific pages for prefetching or evicting. All these optimizations remain inside one application and will not impact other applications sharing the same system. User-space paging is not only limited to backing store on file systems. In contrast to kernel mode, the fault-handling thread has the liberty to fetch data from a variety of backing stores, such a memory server, databases, and even another process.

III. Design

In this section, we describe the design of *UMap* . We first provide an overview of the architecture and then focus on four optimizations for achieving high performance in user-space.

A. Overview

UMap provides an interface for applications to register multiple virtual address ranges, called *UMap regions* that

bypass the kernel service and instead, be managed in user-space. Figure 1 presents the *UMap* architecture. Dark blue regions in the virtual address space are *UMap* regions. Each region has a backing store, where the data is physically located. *UMap* provides an abstraction layer in the store object (yellow circles) for accessing different types of storage. When an application accesses a *UMap* region, if the accessed page is not present in the physical memory, page faults are triggered. These page faults queue up in a FIFO buffer and multiple *UMap* fillers cooperatively resolve these faults. If the requested pages are not fetched in yet, *UMap* fillers will invoke the access functions defined in the store object to read data from the underlying storage. If the buffer is fully occupied, some pages need to be evicted following a user-defined strategy. In the background, a group of *UMap evictors* keep monitoring the ratio of dirty pages in the buffer. Once the ratio of dirty pages reaches a (configurable) high watermark, *UMap* evictors will coordinately write data to the storage.

B. I/O Decoupling

Our design decouples the I/O operation from the fault-handing threads to achieve high concurrency in long latency tasks. I/O operations that move data between storage and memory have a much longer latency than memory accesses. For instance, latency to the state-of-art persistent memory (PM) is about 100 - 500 ns [13], latency to NVMe-based SSD is in the range of ≈ 20 μs [3] while accesses to HDD would require several milliseconds. In contrast, memory accesses typically takes 20-100 ns. To improve the I/O performance, *UMap* employ a configurable number of threads for moving data between storage and memory to exploit the bandwidth supported by the hardware.

The dedicated two groups of I/O threads is referred to as *fillers* and *evictors*, as illustrated in the orange and blue boxes in Figure 1. Fillers split the workload of copying pages to memory while evictors concurrently write data to storage. A separate group of manager threads, typically with low concurrency, keeps polling for notification of tracked events from the operating system. By decoupling the tasks into three groups of workers, *UMap* has the flexibility to adapt the concurrency in each group to reflect their different workload. In contrast, a coupled design results in a long blocking operation that has limited flexibility to optimize.

C. Dynamic Load Balancing

UMap employs a dynamic load balancing strategy to improve resource utilization. We find that memory-mapped regions could have hot and cold segments. Hot segments require a higher level of concurrency for frequent data movement and more physical memory for buffering data than cold segments. For instance, social networks are considered as a type of scale-free network whose degree distribution follows a power law. Memory segment that stores high-degree vertices would naturally result in more accesses than the regions that store low-degree vertices. We design *UMap* to avoid load imbalance even in such skewed data access patterns by dynamically distributing workloads from all memory regions among *UMap* fillers.

UMap employs a dynamic scheduling strategy similar to "work stealing" approach in task-based programming models [15]. *UMap* uses a single *UMap buffer* object to manage the metadata of in-memory pages for all regions. When *UMap* receives the notification of a fault event from the operating system, it appends the workload for resolving this fault into a dynamically growing queue. A group of workers split the pending workload to load pages from the backing store collectively. Consequently, when hot memory segments generate more workloads than others, they will be assigned with more working threads. Orthogonal to the data fetching task is the data flushing task that writes dirty pages back to the persistent stores. When the number of dirty pages reaches a high watermark, the workload is appended to a separate queue and then split by a different group of workers. Figure 1 illustrates the shared (internal) buffer and the work distribution among workers. The dynamic load balancing design prepare *UMap* to cope with applications with diverse access patterns.

D. Extensible Back Store

UMap provides a data object abstraction layer to support different types of backing stores. Currently, applications running on leadership supercomputers have multiple choices of storage, including local SSD, network-interconnected SSD, and HDD. In the future, architectures that provide disaggregated memory and storage resources are likely to emerge. Based on this observation, our design ensures that *UMap* is extensible for current and future architectures.

UMap facilitates applications to associate their own backing store for each memory region. The application has specific control over which storage layer to access to resolve a page fault. In this way, an application is presented with a uniform interface as the virtual memory address space while *UMap* in the backend handles data movement to/from various types of storage.

E. User-controlled Page Flushing

We design *UMap* to enable user-space control on page flushing to a persistent store. There are two motivations. First, the system service may write dirty pages to storage whenever the operating system deems appropriate. Unpredictable behavior may occur if a memory range requires strong consistency such as atomicity among multiple pages. Second, frequent page flushing is known to cause increased performance variation and degradation. For instance, RHEL trigger page flushing when more than 10% pages are dirty [19]. With user control, the application could avoid aggressive page flushing by setting a high threshold or even postponed page flushing to a later stage. *UMap* monitors the ratio of dirty pages to compare with a user-defined high watermark to trigger page flushing as well as a low watermark that suspends page flushing.

F. Application-Specific Optimization

UMap maintains a set of parameters for programmers with application knowledge to configure page management. One

of the most performance-critical parameters is the internal page size of a memory region, denoted as *UMap* page. *UMap* supports an arbitrary page size for each memory region while the system service only supports fixed page sizes. *UMap* page defines the finest granularity in data movement between memory and backing store. For the same memory region, choosing a large *UMap* page could reduce the overhead of metadata, but may also move more than accessed data into memory. By tuning the page size, an application could identify an optimal configuration that balances the overhead and data usage. Also, an application can control the page buffer size, which can alleviate OOM situations in unconstrained mmap.

UMap also supports a flexible prefetching policy that can fetch pages even in irregular patterns. The operating systems usually recognize page accesses as either sequential or random, to increase or decrease the readahead window size, respectively. Real-world applications, however, exhibit complex access patterns, and the general prefetching mechanism becomes insufficient. In contrast, *UMap* could prefetch a set of arbitrary pages into memory, as informed by the application. Moreover, an application can control the start of prefetching to avoid premature data migration that interferences with pages in use. This flexibility, together with knowledge from application algorithm or offline profiling, eases application performance tuning.

IV. Implementation

UMap is implemented in $C++$ and uses the userfaultfd system call [1]. *UMap* enables application controls on page management through both API and environmental variables. The fault-handling thread resolves the page fault by calling the application-supplied function (if provided), or performing direct I/O to the backing store by invoking the defined access functions. *UMap* uses the `UFFDIO_COPY` ioctl [7] to ensure atomic copy to the allocated memory page before waking up the blocked process.

A. API

UMap provides similar interfaces as `mmap` to ease porting existing applications. An application can register/unregister multiple memory regions to be managed by *UMap* through the `umap` and `uunmap` interface. One additional flexibility provided by *UMap* is the multi-file backed region. Given a set of files, each with individual offsets and size, *UMap* maps them into a contiguous memory region. While applications can rely on *UMap* runtime for managing pages, *UMap* also provides a plugin architecture that allows application to register callback functions. A set of configuration interfaces with naming convention `umapcfg_set_xx`, allow the application to control paging explicitly: (1) the maximum size of physical memory used for buffering pages; (2) the level of concurrency for processing I/O operations in each group of workers; (3) the threshold value for starting or suspending writing dirty pages to back stores. Listing 1 illustrates a simple application that uses paging and prefetching services in *UMap* .

Listing 1: UMap API

```
1
2  int fd = open(fname, O_RDWR);
3  void* base_addr = umap(NULL, totalbytes,
       PROT_READ|PROT_WRITE, UMAP_PRIVATE, fd, 0);
4
5  //Select two non-contiguous pages to prefetch
6  std::vector<umap_prefetch_item> pfi;
7  umap_prefetch_item p0 = { .page_base_addr = &base[5 *
       psize] };
8  pfi.push_back(p0);
9  umap_prefetch_item p1 = { .page_base_addr = &base[15 *
       psize] };
10 pfi.push_back(p1);
11 umap_prefetch(num_prefetch_pages, &pfi[0]);
12
13 computation();
14
15 //release resources
16 uunmap(base_addr, totalbytes);
```

B. Environmental Controls

UMap uses a set of environment variables to control: the number of fillers and evictors; the buffer size; the buffer draining policy; and the read-ahead window size. We highlight the key environment variables that *UMap* tracks to dictate its runtime behavior:

- UMAP_PAGESIZE sets the internal page size for memory regions
- UMAP_PAGE_FILLERS sets the number of workers to perform read operations from the backing store. Default: the number of hardware threads.
- UMAP_PAGE_EVICTORS sets the number of evictors that will perform evictions of pages. Eviction includes writing to the backing store if the page is dirty and informing the operating system that the page is no longer needed. Default: the number of hardware threads.
- UMAP_EVICT_HIGH_WATER_THRESHOLD sets the threshold in *UMap* buffer to trigger the evicting procedure. Default: 90%
- UMAP_EVICT_LOW_WATER_THRESHOLD sets the threshold in *UMap* buffer to suspend evicting procedure. Default: 70%
- UMAP_BUFSIZE sets the size of physical memory to be used for buffering *UMap* pages. Default: (80% of available memory)
- UMAP_READ_AHEAD sets the number of pages to read-ahead when resolving a demand paging. Default: 0
- UMAP_MAX_FAULT_EVENTS: sets the maximum number of page fault events that will be read from the kernel in a single call. Default: the number of hardware threads.

C. Limitations

The current implementation uses the write protection support from the kernel to track dirty pages in the physical memory. For pages in write-protected memory ranges, a writes will trigger a fault that sends a UFFD message to handling threads. Currently, the write protection support in userfaultfd is only available in the experimental Linux kernel [2].

[2]Linux Patch https://git.kernel.org/pub/scm/linux/kernel/git/andrea/aa.git.

TABLE I: The AMD Testbed Specifications

Platform	Penguin® Altus® XE2112 (Base Board: MZ91-FS0-ZB)
Processor	AMD EPYC 7401
CPU	24 cores (48 hardware threads) × 2 sockets
Speed	1.2 GHz
Caches	64KB 8-way L1d and 32KB 4-way L1i, 512KB 8-way private L2, 8MB 8-way shared L3 per three cores
Memory	16 GB DDR4 RDIMM × 8 channels (2400 MT/s) × 2 sockets
Storage	≈ 3 TB NVMe (type: HGST SN200)

TABLE II: The Intel Testbed Specifications

Platform	S2600WTTR (Base Board: S2600WTTR)
Processor	Intel Xeon E5-2670 v3 (Haswell)
CPU	12 cores (24 hardware threads) × 2 sockets
Speed	2.3 GHz (Turbo 3.1 GHz)
Caches	32KB 8-way L1d and 32KB 8-way L1i, 256KB 8-way private L2, 30MB 20-way shared L3
Memory	2 16 GB DDR4 RDIMM × 4 channels (1866 MT/s) × 2 sockets
Storage	≈ 1.5 TB NVMe SSD(type: HGST SN200)

V. Experimental Setup

In this section, we describe the experimental setup for the evaluation. We summarize the configuration parameters of two testbeds in Table I and II. The AMD testbed includes three identical machines (Altus, Bertha, Pmemio) that feature two AMD EPYC 7401 (24 cores /48 hardware threads) processors. The testbed has a total of 256 GB DDR4 DRAM and 16 memory channels that operate at 2400 MT/s. Each machine has a total of 4.65 TiB disk capacity, including 1.8 GB SATA Micron 5200 Series SATA SSD. The platform runs Fedora 29 with Linux kernel 5.1.0-rc4-uffd-wp-207866-gcc66ef4-dirty (experimental version) . We compiled all applications using GCC 8.3.1 compiler with support for OpenMP. We use the local SSD on the AMD testbed to evaluate the impact of *UMap* page sizes in all applications. The second testbed, the Intel testbed is on a cluster called *flash*. Its storage includes a remote HDD through Lustre parallel distributed file system. It also features 1.5 TB local SSD. We test the asteroid detection application on this testbed to compare the performance of the backing store on Lustre with the local SSD. The platform runs the Red Hat Enterprise Linux 7.6 kernel. We compiled all applications using GCC 8.1.0 compiler.

VI. Evaluation

In this section, we evaluate the performance of *UMap* in data-intensive benchmarks and applications. In particular, we study the performance benefit of enabling flexible page sizes at application level.

A. Out-of-core Sort

Our first evaluation uses an in-house sorting benchmark, called umapsort. Umapsort is a multi-threaded program that performs quicksort on values stored in a file. Thus, umapsort is a read-write workload. For the evaluation, we use a single 500GiB data set of a sequence of ascending 64-bit words. We configured the benchmark to memory map data sets either using the mmap system call or *UMap* API. Then, the program sorts the values in the memory region into descending order. The application was configured to run with

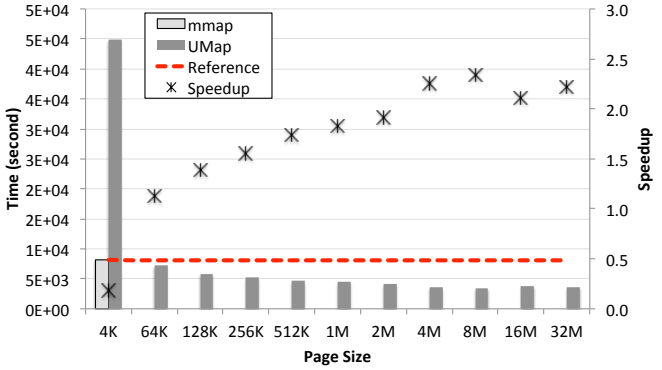


Fig. 2: The performance of *UMap* for sorting 500 GiB data on NVMe-SSD on the AMD testbed, as normalized to that of mmap. *UMap* starts outperforming mmap when the page size is larger than 64KB. At the page size of 8 MB, *UMap* achievs 2.5 times improvement compared to mmap.

96 OpenMP threads on the AMD testbed with 256GiB of physical memory. The data set is stored on the local NVMe-SSD device configured with its default boot-time values. We report the experimental results in Figure 2.

We used different numbers of fillers and evictors to identify the optimal concurrency for this benchmark. In most tested cases, using 48 fillers and 24 evictors brings the best performance. We then fixed the number of fillers and evictors to test the impact of different page sizes. For the mmap tests, we use its default setting and the standard 4KiB page size. For *UMap* tests, we change the page size to identify the optimal configuration. At the smallest page size, *UMap* shows much higher overhead than mmap. We find that increasing page sizes in *UMap* steadily improves the performance. At 64KiB page, *UMap* starts outperforming mmap. By adjusting *UMap* page size to 8MiB, the *UMap* version achieves 2.5 times speedup compared to the mmap version. One reason for the improved performance at larger page sizes is that the reduction in page faults, which reduces the time spent in servicing page faults and also aggregate smaller data transfers into bulky transfers to exploit bandwidth. As the change is localized to the application process, there is no need to modify any OS page size or file system prefetch settings.

B. Graph Application

We implemented a conventional level-synchronous BFS algorithm. Our BFS program takes a graph with compressed sparse row (CSR) data format and stores only the CSR graph in the storage device. We used a separated program to generate a CSR graph to make a read-only benchmark and dropped page cache before running the benchmark to achieve consistent results. As for dataset, we used an R-MAT graph generator with the edge falling probabilities used in the Graph500.

Figure 3 shows Umap's BFS performance normalizing to mmap's best performance case where readahed is off. We varied Umap page size from 4 KB to 4 MB and used the default values for its other environmental variables. Umap

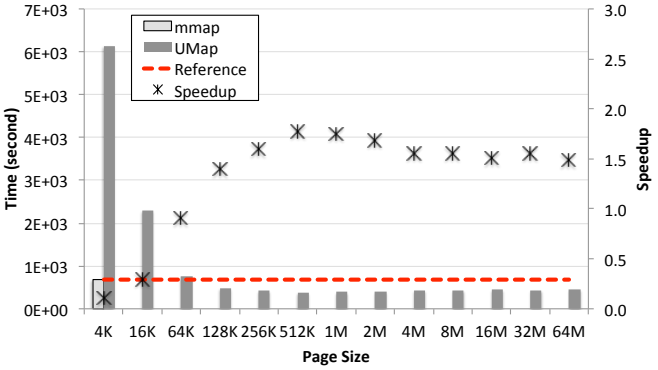


Fig. 3: The relative performance of *UMap* as compared to that of mmap in BFS on an R-MAT scale 31 CSR graph (529 GB) data on NVMe on the AMD testbed.

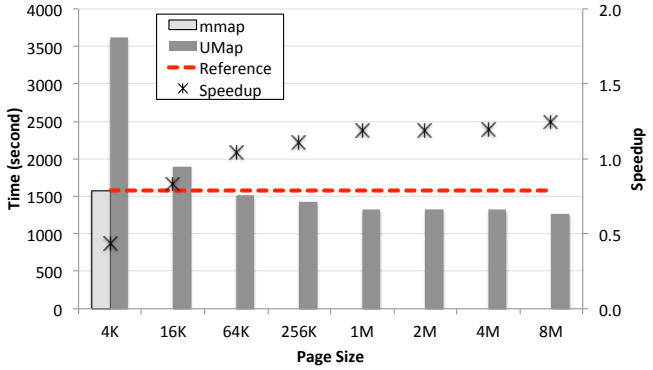


Fig. 4: The relative performance of *UMap* as compared to that of mmap version for LRZIP 64 GB random data on NVMe on the AMD testbed.

showed its best performance and overperformed mmap by 1.8X with 512 KB page size whereas mmap slowed down as increased the page size. We clearly confirmed the benefit of Umap's variable page size feature in terms of not only providing user level control but also better performance.

C. File Compression

Long Range ZIP (lrzip) is a program that implements a full-file compression algorithm [8]. Compression algorithms detect redundancies in input files to reduce size. Lrzip uses a modified RZIP algorithm to achieve an effectively unlimited compression window size. The original mmap version of lrzip uses a large buffer, e.g., one-third of system memory, to mmap a window that 'slides' through the input file. When matches are found, lrzip may use a secondary 64k mmap region to page in any matching regions outside the main window. The *UMap* version removes these sliding buffers and replaces them with a single *UMap* region spanning the entire input file. *UMap* runtime automatically manages the amount of file data paged in memory during execution.

Our experiments run lrzip in pre-processing mode to compare the performance of mmap with *UMap* in RZIP algorithm. We constrain the available memory to the program to ensure out-of-core execution, i.e., 16 GB memory and a 64 GB input data. The *UMap* version sets the environmental variable to limit *UMap* buffer for caching pages in memory. The mmap version requires a command-line option to override the system memory on the testbed. In Figure 4 , lrzip shows low sensitivity to the change in page size. This insensitivity is likely due to the mostly sequential access pattern in lrzip, which only has occasional data reuse of earlier portions of the input file, i.e., when duplicated hash values are found. Once the page size exceeds 1MB, the *UMap* version stabilizes performance at about 1.25 times that of the mmap system call.

D. Asteroid Detection Application

In this case study, we use *UMap* for an on-going study that searches for transient objects, such as asteroids, in intermittent time-series telescope data. We uses UMap to create a 3D cube of virtual address space, where each page is directly mapped to pixel data in a series of image files. UMap has the extensibility to integrate an application-specific FITS handler for resolving page fault to a particular file, which would require extensive porting efforts to achieve in mmap.

The application creates millions or even billions of vectors and then virtually 'traces' them through the image cube to calculate the median pixel value along each vector. The starting point of each vector has a uniform random distribution in the data and their slope follows a given linear function. The backing store contains thousands of FITS format image files. Page faults are resolved to the FITS files containing the requested data, where the pixel data is subsequently read and decoded before copied into the faulting page. Note that a page fault may require access data in multiple files.

The evaluation uses a synthetic data set derived from 537 random images taken from an astronomical survey performed on 12/232018 by the Dark Energy camera in Chile. These files were resized via bicubic resampling to four times their original dimension in each axis in order to emulate the characteristics of real-world datasets. Each file is approximately 977MB with dimensions of 16,000 by 16,000 pixels after this operation. The entire dataset is approximately 512GB. For the Lustre tests, transparent Lustre compression and de-duplication reduces this size to 223GB.

The experiments process a single pass of 32 million vectors with a *UMap* buffer size of 64GB. We demonstrate two types of backing stores in this application. The first uses the local SSD on the AMD testbed. The second uses a backing store mapped to remote disks through a Lustre parallel file system on the Intel testbed. Figure 5 and 6 present the results. Our results show that the application has low sensitivity to page sizes because data reuse among the vectors. A slight performance degradation at large page sizes because larger pages bring more unused data. The execution time initially decreases to the optimal minimum at 1MiB page and then, slightly increases as larger amounts of unused data begins to contend for buffer space.

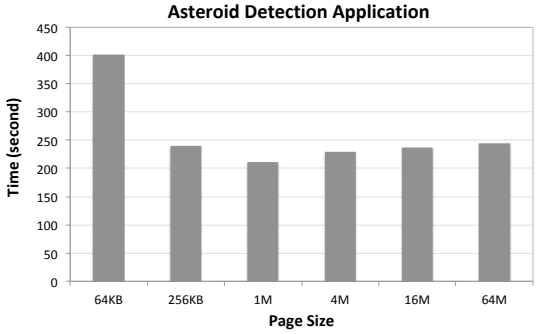


Fig. 5: Execution time of the asteroid application on local SSD at various *UMap* page sizes at 256GB input.

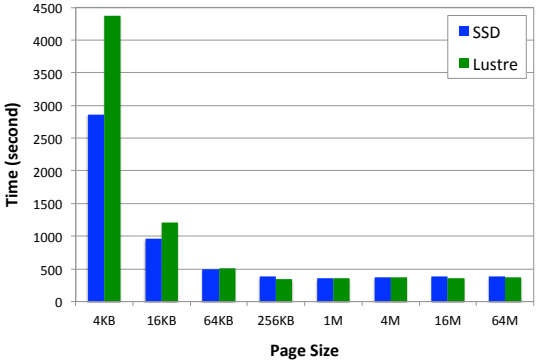


Fig. 6: Compare performance of the asteroid application on local SSD and Lustre using 512GB input.

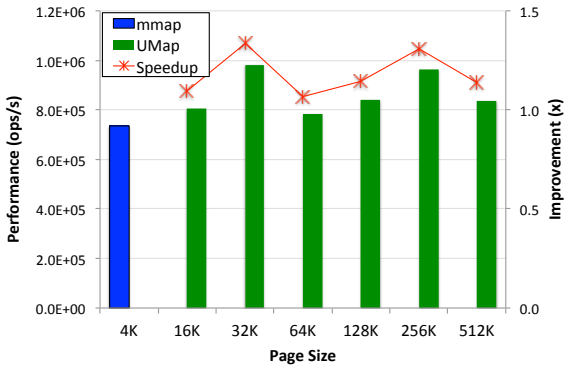


Fig. 7: Compare database throughput using mmap and *UMap* . *UMap* achieves up to 34% improvement at 32KB page.

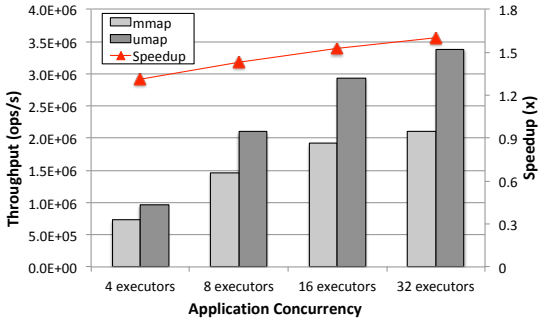


Fig. 8: A scaling test in N-Store using increased number of executors in the database shows that UMap sustains performance scaling at increased application concurrency.

E. Database Workload

This use case demonstrates that *UMap* can be easily plugged into existing database applications to improve user-space control over memory mapping. We ported N-Store [2], an efficient NVM database, to use *UMap* API by changing approximately ten lines of code. N-Store uses persistent memory like SSD as the memory pool for data. Our experiments use a 384 GB persistent memory pool on the local NVMe-SSD on the AMD testbed. N-Store supports multiple executors to execute transactions to the database concurrently. In our evaluation, we sweep 4-32 executors to understand the scalability of *UMap* on variable concurrency. Our workload uses the popular YCSB [4] benchmark with eight million transactions and five million keys. The measurement is repeated ten times, and we report throughput from N-Store as the metric for performance.

We tested different numbers of fillers and evictors to select the concurrency to be 48 fillers and 24 evictors for this benchmark. Then, with a fixed number of fillers and evictors, we test the impact of different page sizes. Figure 7 reports the throughput of *UMap* version at different page sizes and the original mmap version at the default 4KiB page. We find that increasing page sizes in *UMap* does show a trend of increased performance as other applications. The highest throughput is achieved at 32KiB page size, which is about 34% improvement of the mmap version. This page size is smaller than the optimal page sizes in other applications because the access pattern in the benchmark has low locality and mostly random.

Figure 8 report the throughput of the database at an increased application concurrency, i.e., the number of executors increases. The scaling test results demonstrate the advantage of *UMap* in addressing application requirements that change dynamically. When the number of executors increases from four to 32, the gap between the *UMap* version and the mmap version increases (in the gray bars). In particular, the speedup by *UMap* increases from 1.3x to 1.6x steadily (the red line). This result highlights the importance of a scalable design in *UMap* for handling various application workloads.

VII. Discussion

There are several future directions for *UMap* to support emerging architectures.

Multi-tiered Storage has tiered access latency and bandwidth. Currently, *UMap* is extensible for new layers by defining new data objects. In the future work, we will automate data migration between data objects and adapt to application characteristics to improve storage utilization.

Disaggregated Memory architecture has large-capacity memory servers connected to compute node through high-performance network to provide memory on demand. *UMap* can be used to port applications on such architecture by

providing a backing store that defines access functions likely using RDMA for moving to/from memory server.

Byte-addressable NVM requires strong consistency for system software like file systems and DAX-aware mmap lacks such support [20]. The *UMap* buffer could provide applications with explicit control on when to persist changes cached in volatile memory.

VIII. Related Works

Previous works have identified limitations in system services for data-intensive applications that perform out-of-core execution for large data sets [5], [18]. [16] analyzes the overhead in the path through Linux virtual memory subsystem for handling memory-mapped I/O. They conclude that kernel-based paging will prevent applications to exploit fast storage. Our approach aims to provide flexibility to adapt memory mapping to application characteristics and back store features.

DI-MMAP [17] provides a loadable kernel module that combines with a runtime to optimize page eviction and TLB performance. This approach requires updates to remain compatible with the fast-moving kernel. CO-PAGER [10] also provides a user-space paging service by combining a kernel module with a user-space component. CO-PAGER bypasses complex I/O subsystem in the kernel to reduce the overhead of accessing NVM. Our approach stays in user-space completely, and require no modification in the kernel or updates due to kernel updates. Moreover, our design can support a variety of back stores. For instance, remote memory paging that fetches data from a memory server or compute node [6], [14] could be easily integrated into *UMap* by providing a new store object.

IX. Conclusions

In this work, we provide a user-space page management library, called *UMap* , to flexibly adapt memory mapping to application characteristics and storage features. *UMap* employs the lightweight userfaultfd mechanism to enable applications to control critical parameters that impact the performance of memory mapping large data sets while confining the customizations within the application without impacting other applications on the same system. We evaluate *UMap* in five applications using large data sets on both local SSD and remote HDD. By adapting the page size in each application, *UMap* achieved 1.25 to 2.5 times improvement compared to the system service mmap. In summary, *UMap* can be easily plugged into data-intensive applications to enable application-specific optimization.

Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-PROC-788145). This research was also supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

References

[1] Andrea Arcangeli. Userland page faults and beyond. https://schd.ws/hosted_files/lcccna2016/c4/userfaultfd.pdf, 2019.

[2] Joy Arulraj, Andrew Pavlo, and Subramanya R Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 707–722. ACM, 2015.

[3] Danny Cobb and Amber Huffman. NVMe overview. In *Intel Developer Forum*. Intel, 2012.

[4] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[5] Michael Cox and David Ellsworth. Application-controlled demand paging for out-of-core visualization. In *Proceedings. Visualization'97 (Cat. No. 97CB36155)*, pages 235–244. IEEE, 1997.

[6] Sandhya Dwarkadas, Nikolaos Hardavellas, Leonidas Kontothanassis, Rishiyur Nikhil, and Robert Stets. Cashmere-vlm: Remote memory paging for software distributed shared memory. In *Proceedings 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing. IPPS/SPDP 1999*, pages 153–159. IEEE, 1999.

[7] Linux kernel. Userfaultfd. https://www.kernel.org/doc/Documentation/-vm/userfaultfd.txt, 2019.

[8] Con Kolivas. Lrzip – long range zip. https://github.com/ckolivas/lrzip, 2019.

[9] Anthony Kougkas, Hariharan Devarajan, and Xian-He Sun. Hermes: a heterogeneous-aware multi-tiered distributed I/O buffering system. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pages 219–230. ACM, 2018.

[10] Feng Li, Daniel G Waddington, and Fengguang Song. Userland co-pager: boosting data-intensive applications with non-volatile memory, userspace paging. In *Proceedings of the 3rd International Conference on High Performance Compilation, Computing and Communications*, pages 78–83. ACM, 2019.

[11] Sai Narasimhamurthy, Nikita Danilov, Sining Wu, Ganesan Umanesan, Stefano Markidis, Sergio Rivas-Gomez, Ivy Bo Peng, Erwin Laure, Dirk Pleiter, and Shaun De Witt. Sage: percipient storage for exascale data centric computing. *Parallel Computing*, 83:22–33, 2019.

[12] I. B. Peng and J. S. Vetter. Siena: Exploring the design space of heterogeneous memory systems. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 427–440, Nov 2018.

[13] Ivy B. Peng, Maya B. Gokhale, and Eric W. Green. System evaluation of the Intel Optane byte-addressable NVM. In *Proceedings of the International Symposium on Memory Systems*. ACM, 2019.

[14] Sergio Rivas-Gomez, Roberto Gioiosa, Ivy Bo Peng, Gokcen Kestor, Sai Narasimhamurthy, Erwin Laure, and Stefano Markidis. MPI windows on storage for HPC applications. *Parallel Computing*, 77:38–56, 2018.

[15] Arch Robison, Michael Voss, and Alexey Kukanov. Optimization via reflection on work stealing in tbb. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8. IEEE, 2008.

[16] Nae Young Song, Yongseok Son, Hyuck Han, and Heon Young Yeom. Efficient memory-mapped I/O on fast storage device. *ACM Transactions on Storage (TOS)*, 12(4):19, 2016.

[17] Brian Van Essen, Henry Hsieh, Sasha Ames, Roger Pearce, and Maya Gokhale. DI-MMAP–a scalable memory-map runtime for out-of-core data-intensive applications. *Cluster Computing*, 2013.

[18] Brian Van Essen, Roger Pearce, Sasha Ames, and Maya Gokhale. On the role of nvram in data-intensive architectures: an evaluation. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 703–714. IEEE, 2012.

[19] Rik van Riel and Peter W. Morreale. Sysctl in kernel version 2.6.29. https://www.kernel.org/doc/Documentation/sysctl/vm.txt, 2008.

[20] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, 2016.

Extending OpenMP `map` Clause to Bridge Storage and Device Memory

Kewei Yan
University of North Carolina at Charlotte
Charlotte, North Carolina, USA
kyan2@uncc.edu

Anjia Wang
University of North Carolina at Charlotte
Charlotte, North Carolina, USA
awang15@uncc.edu

Xinyao Yi
University of North Carolina at Charlotte
Charlotte, North Carolina, USA
xyi2@uncc.edu

Yonghong Yan
University of North Carolina at Charlotte
Charlotte, North Carolina, USA
yyan7@uncc.edu

***Abstract*—Heterogeneous architectures for high performance computing, particularly those systems that have GPU devices attached to the host CPU system, offer accelerated performance for a variety of workloads. To use those systems, applications are commonly developed to offload most computation and data onto an accelerator while utilizing host processors for helper tasks such as I/O and data movement. The approach requires users to program I/O operations for reading and writing data from and to storage, and to and from host memory. Then users are required to program operations for moving data between host memory and device memory. In this paper, we present our extension to the OpenMP `map` clause for programming directly reading and writing data between storage and device memory. The extension includes mechanism for handling metadata such that metadata can be manipulated independently from data itself. This work demonstrates a prototype runtime, and the support for binary and image data format, including jpeg and png, with OpenCV. Experiments on matrix and image processing kernels show that the designed extension can significantly reduce programming efforts for manipulating data and metadata among storage, host memory, and device memory.**

***Index Terms*—OpenMP, `map` clause, accelerator, data movement, storage and memory**

I. Introduction

The past decade has seen dramatically increased complexity of computer systems for high performance computing. We have experienced the increase of parallelism from 10s to 100s and 1000s cores. Memory and storage systems have been changed significantly, including the adoption of 3D-stacked memory [1], and the use of high-density persistent memory that has both storage and memory properties. The widely-used heterogeneous architectures, particularly those that use GPU accelerators for HPC applications, introduce a discrete and separate memory space between host memory and device memory, adding another dimension of memory complexity for node-level parallel programming. On top of decomposing computation and mapping parallelism onto hardware [2], [3], users now often spend a large amount of efforts to develop and tune code for data movements between different memory spaces, and to optimize local and shared data access with regard to the memory hierarchy and storage system.

Parallel programming models for HPC such as OpenMP and OpenACC provide high-level APIs for programming accelerators by annotating sequential or CPU-parallelized code with pragmas. These pragmas, such as the OpenMP's **`target`** directive and **`map`** clause, and OpenACC's similar directives including **`acc`**, **`copyin`** and **`copyout`** can be used to indicate compiler how to generate code for the runtime operations for offloading data and computation between host and accelerators (or devices). Compared with using low-level APIs such as NVIDIA CUDA and OpenCL, such APIs significantly improve the productivity of programming accelerators by reducing large amount of hand-tuning efforts for platform-dependent optimization. A common practice to program application on accelerators is to let the application offload most computation and data onto an accelerator while utilizing host processors for helper tasks such as I/O and data movement. The approach requires users to program I/O operations for reading and writing data from and to storage, and to and from host memory. Then users are required to program operations for moving data between host memory and device memory. While GPU unified memory (with host) and the upcoming GPUDirect Storage [4] technique offers driver-based solutions to enable GPU's direct access to host memory and storage, correctly using those APIs still requires understanding of the storage and memory system in heterogeneous architectures, and substantial amount of programming efforts.

In this paper, we explore high-level APIs for bridging data movement between storage and device memory in accelerator-based heterogeneous systems with goals to reducing programming efforts of handling I/O between storage and host memory, and for moving data between host and device. The technical contribution includes: 1) we develop extension to the OpenMP **`map`** clause for reading and writing data from and to an I/O URI, to and from a device memory. This extension enables direct storage and network access for a program from device without the need to write code for I/O operations; 2) we

develop mechanism for handling metadata such that metadata can be manipulated independently from data itself; 3) we have demonstrated the implementation of the runtime support, and experimented POSIX stream data and image data format with OpenCV using matrix and image processing kernels. Our prototype and experiments show that the designed extension can significantly reduce programming efforts for manipulating data and metadata among storage, host memory and device memory.

In the rest of the paper, Section II provides description about the background and motivation of this work. In Section III, we present our extension to the OpenMP **map** clause to address the need mentioned in the paper. In Section IV, we discuss how the runtime supports those extension in the implementation and evaluation results generated from matrix and image processing kernels. Related work is discussed in Section V and the paper concludes in Section VI.

II. Background and Motivation

High level programming models such as OpenMP and OpenACC provide pragma APIs for users to annotate code and data region to be offloaded to accelerators for computation. In this section, we describe how to use those APIs focusing on the data mapping and data movement clause, and thus motivate our work.

A. OpenMP **map** clause

The OpenMP's **map** clause is used with the OpenMP **device** related directives (e.g. **target**, **target data**) for indicating how data should be copied between host memory and device memory[5]. The syntax of the **map** clause is as follows and Figure 1 demonstrates its usage.

map(*[[map-type-modifier[,] [map-type-modifier[,]...] map-type :] locator-list***)**.

```
1  extern void init(float*, float*, int);
2  void vec_mult(float p[N], float *v1, float *v2,
       int N) {
3    init(v1, v2, N);
4    #pragma omp target map(to: N, v1[0:N], v2[:N])
         map(from: p)
5    #pragma omp parallel for
6    for (int i=0; i<N; i++)
7      p[i] = v1[i] * v2[i];
8  }
```

Fig. 1: Example of using **map** clause for specifying mapping of scalar variable (N), array (P), and array sections (v1[0:N] and v2[:N]). The **target** directive is used to indicate offloading of the associated code region.

An item of *locator-list* in the syntax can be any l-value expression, which declares the data to be mapped. The *map-type-modifier* and the *map-type* specify the effect of the **map** clause. The *map-type* could be **to**, **from**, **tofrom**, **alloc**, **release** and **delete**. *map-types* **to** and **from** indicate one-way transfer, to device or to host. **tofrom** combines **to** and **from** modifiers together. Modifier **alloc** indicates the memory need to be allocated for the data. Modifiers **release** and **delete** indicate the allocated memory need to be released or deleted, respectively. For the similar functionality as the OpenMP's **map** clause, OpenACC has clauses of **copyin**, **copyout**, **copy**, **create**, **delete**, and **present**.

The *map-type-modifier*s could be **always**, **close** and **mapper**(*mapper-identifier*). The **always** option means that regardless of whether the items in the *locator-list* are deleted or added, the data is always mapped according to the *map-type*. For example, if the *map-type* is **to** or **tofrom**, and **always** is used, the value of the original list item will be copied to the device environment, regardless of whether the item was mapped before or not. The **close** *map-type-modifier* is a hint to the runtime to allocate memory close to the target device. The **mapper** *map-type-modifier* indicates the data will be mapped following a predefined pattern. And the *mapper-identifier* indicates the mapper to be used in the current **map** clause.

OpenMP allows mapping user-defined data type, e.g. a struct variable. In general, if a list item in a **map** clause is a variable of struct type, then it is treated as if each struct element in the variable is a list item in the clause. If the list item is an element of a struct, then all other elements in the struct would form a struct sibling list for mapping. Figure 2 gives an example. In this example, *S.a*, *S.b* and *S.p* in the list of the **map** clause have corresponding variables and storage on the device. But *S.buffera*, *S.bufferb* and *S.x* cannot be accessed from device since they are not specified in any **map** clause.

```
1  struct foo {
2    char buffera[1000000], bufferb[1000000];
3    float x, a, b, *p;
4  };
5
6  int main() {
7    struct foo S;
8    S.a = 2.0; S.b = 4.0;
9    S.p = (float *)malloc(sizeof(float)*100);
10   for(int i=0; i<100; i++) S.p[i] = i;
11   #pragma omp target map(alloc:S.p) map(S.p
         [0:100]) map(to:S.a, S.b)
12     for(int j=0; j<100; j++)
13       S.p[j] = S.p[j]*S.a + S.b;
14   return 0;
15 }
```

Fig. 2: Example of using **map** clause for struct mapping

OpenMP also allows for users to declare a **mapper** using the **declare mapper** directive, and use the mapper in the **map** clause to do default data mapping associated with certain data types or customized mapping between host memory and device memory. The syntax is shown below:

declare mapper(*[mapper-identifier:] type var) [clause[[,] clause] ...]*

The optional *mapper-identifier* field specify the mapper name that can be used later within **map** clause via modifier **mapper***(mapper-identifier)*. The *type* field indicate user-defined data type associated with this mapper. *clause* here

are mainly the **map** clauses that are for specifying the mapping rules. Figure 3 shows an example of using **declare mapper** directive. The directive specifies a mapper identified by *my_mapper* that should be applied to the *myvec_t* type. Line 9 shows how we can use the mapper in **map** clause, with the mapping rules of the mapper applied to the *s* variable of *myvec_t* type.

```
1 typedef struct myvec {
2    size_t len;
3    double *data;
4 } myvec_t;
5 #pragma omp declare mapper( my_mapper: myvec_t v
      ) map(v, v.data[0:v.len])
6 ...
7 myvect_t s;
8 ...
9 #pragma omp target map(mapper(my_mapper), to: s)
```

Fig. 3: Example of **declare mapper**

B. Motivation

In the current OpenMP specification, the **map** clause can be used for only mapping and transferring data between memories, mainly between host memory and device memory. For applications that process large amount of data, a typical workflow of using accelerator has been 1) reading data from storage or network to host memory, 2) copying data to device memory, 3) processing data, 4) copying results back to the host memory, and 5) writing data to the storage. Thus with the OpenMP **target** directive and **map** clause, users have to program I/O operations for reading and writing data between storage and host memory. To reduce the programming effort, our extension relaxes the restriction of **map** clause to enable direct access to data at any location.

III. Extension to OpenMP map Clause

Our goal for extending OpenMP **map** clause is to improve the productivity of parallel programming with data processing. We propose to unify major data processing operations with much simpler interface, including loading data from storage, laying out data in memory, moving data between host memory and device memory and handling metadata. In this section, we present our extension and illustrate the extension with examples.

A. Design

The current syntax of the **map** clause in the specification is " **map(***[[map-type-modifier[,] [map-type-modifier[,]...] map-type:] locator-list***)** ". Our extension is to append an optional field for a list item of the *locator-list* to include the source or destination location of the data. The syntax can be described as follows:

list-item [= {[*data-format-driver*:] *data-location*[, *place-modifier*][, **metadata**([*place-modifier*,] *meta-identifier*)]}].

The extension allows for specifying a *data-location* in the storage, network, or a network live stream. It could refer to a local file, network URL, or a mounted storage device. A *data-location* includes a URL, and an optional range field. The range field is used to specify a range of *data-location* where data are to be read from or written to. Thus data can be partially loaded by specifying a linear range of the location at the *data-url*. It works like array section. For instance, *data-url[lower-bound:length]* indicates to read or write data at the location starting from index *lower-bound* for *length* elements continuously. The data range can be applied to different data formats. The unit of data depends on the corresponding data format. For POSIX stream data, it's assumed that each element is a `char`, which takes one byte. In other cases, the data type of elements is determined by the specific *data-format-driver*.

data-format-driver indicates the driver that should be used to read and write data from data source or destination location. Currently, we experimented POSIX stream data access and image access through OpenCV library. *data-format-driver* can be one of the following in the current implementation: **posix**, **jpeg**, **png**, or *user-defined-data-driver*. If it's not specified, the data are handled as POSIX stream data by default. This field is extendable for users who want to define their driver for application-specific data format. We expect to define API conventions for users to create a driver so that it can be integrated with our extension and the runtime implementation.

The **metadata** field and its parameters are used to specify where metadata should be read to or written from. The provider of a *data-format-driver* defines how metadata should be read or written when creating such a driver. For example, POSIX stream data format does not require metadata. For image data format, we use OpenCV library to handle image metadata.

The *place-modifier*, which can be **host** or **hostonly**, indicating whether data or metadata should be read into or written from host as well as the device, or from host only. This field is optional. If it is not used, it indicates data or metadata will only be read to or written from device of the associated **target** directive. The *place-modifier* for data and metadata has slightly different semantics depending on whether the *map-type-modifier* of the **map** clause is **to** or **from**. For **to** map type, if the *place-modifier* is not provided, data or metadata will be read to the device memory only. The **host** *place-modifier* indicates to read data or metadata onto host and device data environment. **hostonly** indicates to read data or metadata to the host only but not device. For the **from** map type, if the *place-modifier* is not provided, data or metadata on the device will be written to the provided *data-location*. The **host** and **hostonly** *place-modifier* have the same semantics indicating that the data or metadata source for writing are located in host environment.

To specify a source or destination location field of a mapped *list-item*, four parameters can be provided. Only one parameter *data-location* is required and the other three are optional. The extension maintains backward compatibility of the **map** clause since if the source or destination field is not present for a *list-item*, it has the same syntax as of the standard **map** clause.

B. Examples

In this subsection, we show three examples that use our extension. They are matrix multiplication, and image processing with or without handling metadata. Each of them has been implemented with annotated pragmas to demonstrate how the proposed OpenMP extension works.

1) POSIX stream data: In this case, matrix multiplication is shown as an example for illustration. Figure 4 shows that matrices `A` and `B`, which are read from files `vectorA.data` and `vectorB.data`, produce matrix `C`. Then matrix `C` is saved to file `vectorC.data`. In this situation, users do not need to write any code for data copy from storage to the memory of either host or device. Moreover, when the result is going to be saved in a certain file, users only need to specify the file destination, as shown in the figure for matrix `C`.

```
1  void mm(float *A, const float *B, float *C, int
       numElements) {
2  #pragma omp target map(to:A[0:numElements]={"
       data/vectorA.data"}, B[0:numElements]={"data
       /vectorB.data"}) map(from:C[0:numElements
       ]={"data/vectorC.data"})
3     int i,j,k;
4     #pragma omp for private(i,j,k)
5     // for loops
6     ...
7  }
```

Fig. 4: Example of using the extended **map** clause for matrix multiplication

2) image data: Smoothing is very common in image processing. It applies a filtering kernel to each pixel of the image and update the values properly. Figure 5 shows that the first **map** clause is used to transfer the image data from the input file `image_in.jpg` to device directly and its metadata is stored in `meta_in` variable for later use. Users do not need to write extra code for I/O operations since they are covered by the extended **map** clause. Similarly, the second **map** clause is used to write the updated image data `imgout` on GPU to the output file `image_out.jpg`.

In this case, only image data need to be mapped to accelerator for computing. The metadata, such as image size, will not be modified. They are copied from input file to output file via an intermediate variable `meta_in`. The *place-modifier* **host** in both **map** clauses indicates that the metadata are stored only on host temporarily.

In the following example, Figure 6 shows an application involving data movement as well as metadata handling. It's used for image resizing and the metadata are modified. In this case, users have to provide modified metadata in the **map** clause to generate correct output file. Because the new metadata have to be ready ahead of the output, to resize an image requires two pragmas. Line 5 shows that metadata of the image are loaded into `meta_in` variable on host only. Line 10 shows the code that generates the new metadata stored in `meta_out` variable. They both are stored in the host environment since we only need CPU to process the metadata. Finally, the new image data computed by device and metadata stored in `meta_out` variable are combined together to produce the output file `image_out.jpg`.

```
1  uchar* imgin, imgout;
2  void* meta_in;
3  #pragma omp target \
4    map (to: imgin = {jpeg : "image_in.jpg",
         metadata(host: meta_in)}) \
5    map (from: imgout = {jpeg : "image_out.jpg",
         metadata(host: meta_in)})
6    for (i=1; i<row-1; i++)
7      for (j=1; j<col-1; j++)
8        imgout[i*col+j] = filter(imgin, i, j);
```

Fig. 5: Example of using the extended **map** clause for smoothing image

```
1  uchar* imgin, imgout;
2  void* meta_in, meta_out;
3  int row = 800, col = 600;
4  #pragma omp target \
5    map (to: imgin={jpeg: "image_in.jpg", metadata
         (host_only: meta_in)})\
6    map (alloc: imgout[800*600])
7    for (i=1; i<row-1; i++)
8      for (j=1; j<col-1; j++)
9        imgout[i*col+j] = zoom(imgin, i, j);
10 meta_out = transform(meta_in, row, col);
11 #pragma omp target data \
12   map (from: imgout = {png: "image_out.png",
         metadata(host: meta_out)})
```

Fig. 6: Example of using the extended **map** clause for resizing image

IV. Prototype Implementation for the Runtime

A. Implementation

In this section we demonstrate how the extended **map** clause can be implemented in the runtime for NVIDIA GPU with CUDA using host memory as bounce buffer. As indicated in our related work, some of existing solutions use host memory as bounce buffer in the library level or driver level, thus our prototype is representative of those work. The implementation of using host-bypass solutions such as NVIDIA GPUDirect storage is our future work when the support for GPUDirect or other similar techniques are available. At the end of this section, we further discuss the potential benefits of the extension using matrix and smoothing kernels.

There are two ways to use CPU memory as bounce buffer. For the first one, host memory is used as bulk bounce buffer for device global memory. Data are read into host bounce buffer from storage according to the type of *data-format-driver*. Then, `cudaMalloc()` is called to allocate a bulk memory for data which is to be offload to GPU. Next step, `cudaMemcpy()` is called for copying data from host bounce buffer to device memory. After finishing computing work, the result is copied back by calling `cudaMemcpy()` as well. In this approach, computation data are read or written in bulk

directly between storage, host memory and device memory. It is hard to achieve pipelined data movement and overlapping data movement and computation in this approach since the amount of data to be moved is determined by the application. The amount of data is also limited by the size of device memory. However, for application that has high reuse data access pattern, this approach enables near data access for all the data since they are copied to the device memory at once.

The second way of using host memory as bounce buffer is through CUDA Unified Memory approach. Unified memory can be allocated using `cudaMallocManaged()` function. Data in CUDA Unified Memory is shared between host and GPU via paging mechanism, thus we call it page bounce buffer. No explicit `cudaMemcpy()` calls are needed. With GPU paging mechanism, when a GPU accesses data that is in unified memory, but not in GPU, GPU initiates page fault signal to the driver on the CPU. The driver has a paging thread that sends the page from host to GPU. With unified memory and paging mechanism, data are copied to GPU on demand, which may increase data access latency compared with bulk data movement. But it allows for overlapping paging and computation.

In the next two subsections, implementations for both POSIX stream data and image data are shown for the two approaches of using host memory as bounce buffer.

1) POSIX stream data: Pseudo codes shown in Figure 7 and Figure 8 are the implementations of matrix multiplication example using the extended **`map`** clause, which is shown in Figure 4. Figure 7 is for using host memory as bulk bounce buffer for GPU global memory. In this case, `fread()` is used to copy the bulk data from storage to host memory. The size of array is given by `N*K`, determined by the application. The data type of the array element is `float`. `cudaMalloc()` and `cudaMemcpy()` are used for memory allocation and data movement between host memory and device memory. Figure 8 is for using host memory as page bounce buffer via GPU Unified Memory. In this version, only `cudaMallocManaged()` is needed to allocate unified memory, and on-demand paging between host memory and device memory are handled by the driver.

2) image data: For the program in Figure 5, one can manually implement in the similar way as we present in Figure 9, which demonstrates a typical runtime implementation to support our extension using host memory as bulk bounce buffer for global memory. Figure 10) shows the implementation using GPU Unified Memory. In this way the CUDA runtime will manage the data transfer between host and device automatically using paging mechanism. However, these two implementations requires users to have sufficient CUDA programming experiences besides OpenMP knowledge.

As for handling metadata when processing images, if metadata are specified, OpenCV functions will be called to load both image data and metadata from storage to the destination given by users. The destination could be host or device.

```
1  fd = fopen("data/vectorA.data", "rb");
2  fread(tA, sizeof(float), N*K, fd);
3  fclose(fd);
4  ...
5  cudaMalloc(&A, sizeof(float)*N*K);
6  cudaMemcpy(A, tA, sizeof(float)*N*K,
       cudaMemcpyHostToDevice);
7  ...
8  cudaMalloc(&C, sizeof(float)*N*M);
9  ...
10 float *h_C = (float *)malloc(sizeof(float)*N*M);
11 ...
12 // MM kernel
13 ...
14 cudaMemcpy(h_C, C, sizeof(float)*N*M,
       cudaMemcpyDeviceToHost);
15 ...
16 FILE *f3;
17 f3 = fopen("data/vectorC.data", "wb");
18 fwrite(h_C, sizeof(float), N*M, f3);
19 fclose(f3);
```

Fig. 7: Implementation of extended **`map`** clause using GPU global memory for matrix multiplication.

```
1  fd = fopen(argv[1], "rb");
2  printf("The size of array1 is: %ld\n", N*M);
3  cudaMallocManaged(&A, sizeof(float)*N*M);
4  fread(A, sizeof(float), N*M, fd);
5  fclose(fd);
6  ...
7  cudaMallocManaged(&C, sizeof(float)*N*M);
8  ...
9  // MM kernel
10 ...
11 FILE *f3;
12 f3 = fopen("data/vectorC.data", "wb");
13 fwrite(C, sizeof(float), N*M, f3);
14 fclose(f3);
```

Fig. 8: Implementation of extended **`map`** clause using GPU unified memory for matrix multiplication.

B. Benefits

The extension of **`map`** clause reduces the programming effort significantly. With the **`map`** extension, users do not need to implement I/O operations and data movements on their own with lengthy code (compare Figure 5 and Figure 9 or 10).

The other benefit is that the extension enables more opportunities of optimization. We choose two use cases to measure detailed execution time cost, which are image smoothing and matrix multiplication. They are implemented with GPU global memory and unified memory, respectively.

The break-down execution time of image smoothing are shown in Tab. I and II. The time for writing output data to storage dominates the total time in both global memory and unified memory cases in this example. One of the potential optimizations is to overlap the I/O operation with kernel execution via `mmap()` function or image processing pipeline. `mmap()` maps the storage to memory and only transfers the data where they are actually required via CPU paging.

The other typical application is matrix multiplication, which

```
1  uchar* imgin_d, imgout_d, imgout_h;
2  uchar* gpu_filter(uchar*);
3  Mat image = cv::imread("image_in.jpg");
4  size_t img_size = input.ncols * input.nrows;
5  cudaMalloc(imgin_d, img_size);
6  cudaMalloc(imgout_d, img_size);
7  malloc(imgout, img_size);
8  // copy data HtoD
9  cudaMemcpy(imgin_d, image.data, img_size,
       cudaMemcpyHostToDevice);
10 // run GPU kernel
11 imgout_d = gpu_filter(imgin_d);
12 // copy data DtoH
13 cudamemcpy(imgout_h, imgout_d, img_size,
       cudaMemcpyDeviceToHost);
14 // write result to a new file
15 image.data = imgout_h;
16 cv::imwrite("image_out.jpg", image);
```

Fig. 9: Implementation of extended **map** clause using OpenCV and global memory. Metadata is loaded to the host only and it is used to write to the output image in this example. For other algorithms, such as enlarging images, metadata may need to be changed to match the output image.

```
1  uchar* imgin, imgout;
2  uchar* gpu_filter(uchar*);
3  Mat image = cv::imread("image_in.jpg");
4  size_t img_size = input.ncols * input.nrows;
5  cudaMallocManaged(imgin, img_size);
6  cudaMallocManaged(imgout, img_size);
7  memcpy(imgin, image.data, img_size);
8  // run GPU kernel
9  imgout = gpu_filter(imgin);
10 // write result to a new file
11 image.data = imgout;
12 cv::imwrite("image_out.jpg", image);
```

Fig. 10: Implementation of extended **map** clause using OpenCV and unified memory. Metadata is loaded to the host only and it is used to write to the output image.

is more compute-intensive than image smoothing. According to Tab. III and IV, GPU kernel consumed the most of total time. Therefore, the data transfer and I/O operations can be divided into multiple smaller pieces and overlapped in the GPU kernel computation.

Our implementation can leverage emerging techniques such as GPUDirect Storage from NVIDIA, which is to be released in the near future. This library skips the system memory thus no bounce buffer is involved. We may apply it to our implementation, for direct access to any storage or NVMe from GPU. Not only the storage, the network could also be considered as a main source for data. We have noticed that GPUDirect RDMA from mellanox, or now part of NVIDIA, is already there. We believe that extending **map** clause for multiple source could be possible.

V. Related Work

Previous work has shown efforts of enhancing connecting GPU to a third party peer device directly, including GPU,

Image Size	Input	Output	HtoD	DtoH	Kernel
512x512	3	274	0.066	0.061	0.204
512x1024	6	629	0.142	0.123	0.526
1024x1024	10	1285	0.338	0.706	0.853
1024x2048	20	2622	0.694	2.197	2.186
2048x2048	35	4833	1.471	5.289	3.545

TABLE I: Breakdown of execution time for image smoothing using global memory (ms)

Image Size	Input	Output	HtoD	DtoH	Page Fault	Kernel
512x512	3	316	0.250	0.174	2.294	2.499
512x1024	6	660	0.303	0.239	3.096	3.176
1024x1024	10	1288	0.381	0.305	2.718	3.491
1024x2048	19	2637	0.813	0.600	5.314	7.241
2048x2048	37	4823	1.381	1.085	8.693	11.785

TABLE II: Breakdown of execution time for image smoothing using unified memory (ms)

storage device and network. GPUDirect RDMA [6] enables GPU-GPU direct connections via PCIe. Some changes are applied to the device drivers to make it work. Traditionally, CPU MMU is used as memory I/O address for mapping resources to user space or kernel address space. However, after changing drivers, NVIDIA drivers enable data mapping or paging from a third party device to GPU. Thus in this situation, system memory is bypassed and communication overheads are eliminated. Based on this feature, some hardware products are created, such as Fusion-io ioMemory flash storage [7]. GPUDirect RDMA could also be extended for Ethernet and Internet communications with Chelsio iWARP RDMA technology [8]. A hardware TCP/IP stack is applied to run in the adapter, completely bypassing the host software stack, thus eliminating any inefficiencies due to software processing.

According to NVIDIA, the upcoming GPUDirect Storage [4] solution requires no more bounce buffer on system memory and PCIe switch handles data communication between NVMe and GPU. Similar to GPUDirect Storage, DRAGON [9] provide a host-based framework to enable user transparent and direct NVMe access by enhancing NVIDIA driver. For NVM-specific operations, DRAGON's solution extends the driver for CUDA unified memory to page data between GPU and NVM. Data in file can be read or write by common load/store instructions like what `mmap()` does. Thus pages will be allocated and data will be paged to GPU.

Besides, GPUfs [10] is created to expose the file system API to GPU programs. It enables I/O read and write from the CPU context and further allows the OS optimize data access and locality across independently-developed GPU modules. Mustafas [11] presents GPUDrive to remedy performance degradation of GPU-accelerated data processing caused by file-driven data movement. It is handled by designing the flash array and optimizing the system software stacks associated to GPU computing.

Techniques and programming interfaces for managing heterogeneous memory, such as device memory, host memory,

Size	Input	Output	HtoD	DtoH	Kernel
512x512	2	1	0.087	0.081	8.778
512x1024	5	1	0.235	0.169	17.469
1024x1024	7	3	0.473	1.195	69.515
1024x2048	16	5	1.467	3.179	141.750
2048x2048	25	13	2.012	7.243	566.620

TABLE III: Breakdown of execution time for matrix multiplication using global memory (ms)

Size	Input	Output	HtoD	DtoH	Page Fault	Kernel
512x512	2	2	0.191	0.087	1.615	10.317
512x1024	4	2	0.482	0.177	3.064	20.465
1024x1024	8	3	0.770	0.349	4.612	73.686
1024x2048	14	6	1.817	0.511	9.168	151.79
2048x2048	25	15	3.248	1.363	17.259	581

TABLE IV: Breakdown of execution time for matrix multiplication using unified memory (ms)

high bandwidth memory, and persistent memory have been popular topics recently. Efforts such as pmem.io [12], which focuses on programming persistency on the emerging non-volatile memory, and memkind and hbmalloc library [13] that are designed to manage high-bandwidth memory for applications, are popular. OpenMP 5.0 standard has its own memory management support [14], which is also platform-agnostic. `malloc()` like interfaces such as `omp_alloc()` and **`allocate`** directive are introduced for users to manage memory on different memory spaces, including host memory, device memory, and others. Umpire [15] introduces unified, application-focused API to handle different kind of platforms as well as different demand of memory allocation. Once the system is detected, the corresponding memory resources will be created thus user do not need to care about the programming platform too much and effort for memory allocation would be saved.

Those techniques and efforts demonstrate the needs and feasibility of direct access from accelerator to storage or network devices, and for managing heterogeneous memories in the existing computer systems. The work in this paper aims to address the programmability challenge of using those diversified storage and memory by leveraging and extending high-level OpenMP APIs for users to simplify programming I/O and memories. Our implementation also enables additional optimization opportunities at both compiler level and runtime level, such as pipelining I/O with data movement between memories and computation, and data layout optimization on both storage and memory according to data format and data processing algorithms.

VI. Conclusion

In this paper, we present the extension of the OpenMP **`map`** clause for programming direct data read and write between storage and device memory. The mechanisms for handling metadata are also included in the extension so that metadata can be manipulated independently. The extension significantly reduces programming efforts for manipulating data and metadata among storage, host memory, and device memory. As recent memory and storage technologies indicate converging of storage and memory in the form of storage-class memory (such as NVMe and NVDIMM) or memory-capable storage (such as byte-addressable SSD), our work explores programming interface and system support for enabling those techniques with high programmability to the end-users. For the future work, the extension and implementation will be refined and enhanced. We will explore GPUDirect Storage library for the implementation and performance evaluation, when it becomes available.

References

[1] G. H. Loh, "3d-stacked memory architectures for multi-core processors," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 453–464. [Online]. Available: http://dx.doi.org/10.1109/ISCA.2008.15

[2] L. Carrington, A. Snavely, and N. Wolter, "A performance prediction framework for scientific applications," *Future Gener. Comput. Syst.*, vol. 22, no. 3, pp. 336–346, Feb. 2006. [Online]. Available: http://dx.doi.org/10.1016/j.future.2004.11.019

[3] *Performance Optimization of Numerically Intensive Codes*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2001.

[4] "Gpudirect storage: A direct path between storage and gpu memory," https://devblogs.nvidia.com/gpudirect-storage/.

[5] "Openmp application programming interface, version 5.0 november 2018," https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.

[6] "Developing a linux kernel module using gpudirect rdma," https://docs.nvidia.com/cuda/gpudirect-rdma/index.html.

[7] "Fusion-io products," https://www.westerndigital.com/products/storage-systems#all-flash-hybrid-storage.

[8] "The gpudirect solution overview," https://www.chelsio.com/the-gpudirect-solution-overview/.

[9] P. Markthub, M. E. Belviranli, S. Lee, J. S. Vetter, and S. Matsuoka, "Dragon: breaking gpu memory capacity limits with direct nvm access," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, 2018, p. 32.

[10] M. Silberstein, B. Ford, I. Keidar, and E. Witchel, "Gpufs: integrating a file system with gpus," in *ACM SIGPLAN Notices*, vol. 48, no. 4. ACM, 2013, pp. 485–498.

[11] M. Shihab, K. Taht, and M. Jung, "Gpudrive: Reconsidering storage accesses for gpu acceleration," in *Workshop on Architectures and Systems for Big Data*, 2014.

[12] "pmem.io: Persistent Memory Programming," https://pmem.io.

[13] C. Cantalupo, V. Venkatesan, and J. R. Hammond, "User extensible heap manager for heterogeneous memory platforms and mixed memory policies," 2015.

[14] "Memory management support for openmp 5.0," https://www.openmp.org/wp-content/uploads/openmp-TR5-final.pdf.

[15] "An application-focused api for memory management on numa & gpu architectures," https://github.com/LLNL/Umpire.

Panel: Software and Hardware Support for Programming Heterogeneous Memory

Moderator: Maya B Gokhale (Lawrence Livermore National Laboratory)

Panelist: Mike Lang (LANL), Jeffrey Vetter (ORNL), Vivek Sarkar (Georgia Tech), David Beckinsale (LLNL), Paolo Faraboschi (HPE)

Usability and programmability of complex and heterogeneous memory systems remain significant challenges facing the HPC and data analytics communities. Existing memory systems that include DRAM, SRAM, discrete memory, software unified memory, and distributed memory are difficult to exploit while maintaining portable performance. Approaches include programming language constructs and runtime libraries, OS enhancements, and even hardware mechanisms to enable the competing goals of programmability and portability.

We would like the panel to address challenges and solutions that address the problems of maintaining portability in applications that must navigate the complex memory hierarchy without sacrificing performance and capability.