Bit Contiguous Memory Allocation for Processing In Memory

John D. Leidel
Tactical Computing Laboratories
McKinney, Texas 75070
jleidel@tactcomplabs.com

ABSTRACT

Given the recent resurgence of research into processing in or near memory systems, we find an ever increasing need to augment traditional system software tools in order to make efficient use of the PIM hardware abstractions. One such architecture, the Micron In-Memory Intelligence (IMI) DRAM, provides a unique processing capability within the sense amp stride of a traditional 2D DRAM architecture. This accumulator processing circuit has the ability to compute both horizontally and vertically on pitch within the array. This unique processing capability requires a memory allocator that provides physical bit locality in order to ensure numerical consistency. In this work we introduce a new memory allocation methodology that provides bit contiguous allocation mechanisms for horizontal and vertical memory allocations for the Micron IMI DRAM device architecture. Our methodology drastically reduces the complexity by which to find new, unallocated memory blocks by combining a sparse matrix representation of the array with dense continuity vectors that represent the relative probability of finding candidate free blocks. We demonstrate our methodology using a set of pathological and standard benchmark applications in both horizontal and vertical memory modes.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**; • **Computer systems organization** → *Single instruction, multiple data*; *Heterogeneous (hybrid) systems*;

KEYWORDS

memory allocation; processing near memory; vector processor; SIMD

ACM Reference format:
John D. Leidel. 2017. Bit Contiguous Memory Allocation for Processing In Memory. In *Proceedings of MCHPC'17: Workshop on Memory Centric Programming for HPC, Denver, CO, USA, November 12–17, 2017 (MCHPC'17),* 9 pages.
DOI: 10.1145/3145617.3145618

1 INTRODUCTION

Given the recent technological shift in high performance, stacked and embedded memory technologies, we have experienced a shift back to research efforts associated with processing near and processing in memory. These efforts have been generally focused on two hardware methods: stacking logic layers under DRAM device layers or modifying the fundamental DRAM logic circuit.

The Micron In-Memory Intelligence (IMI) device technology [8] is a prime example of the latter alternative DRAM logic circuit. The IMI device is built upon a series of simple, bit-serial computing elements placed on pitch below each sense-amplifier. Each bit-serial computing element is capable of basic logical functions which can be combined with the data movement capabilities of the sense amp to form basic arithmetic operations within the array.

However, unlike traditional DRAM devices, these bit-serial operations rely upon a distinct physical locality between the individual bits of an element. In this manner, bits within an element must be physically co-located within the IMI array in order to maintain numerical consistency in boolean and arithmetic operations. This requirement of physical bit continuity is orthogonal to the standard memory addressing methods found in traditional DRAM memory subsystems. Further, traditional memory allocation schemas, such as those found in GNU's *malloc* API, enforce memory allocation that is interleaved across banks in order to enhance performance. This work describes a bit-contiguous memory allocation methodology built specifically for the Micron IMI device architecture that explicitly allocates memory blocks per the physical locality rules defined therein.

The main contributions of this work are as follows. We introduce a novel main memory allocation methodology for a state of the art processing in memory (PIM) architecture. Given the novelty of the processing architecture's ability to compute values on pitch in the sense amp array, our methodology enforces strict bit continuity in its allocation schema in order to preserve numerical consistency when computing values across bit lines. Finally, we demonstrate that our approach provides a substantially lower time complexity when searching for unallocated, bit-contiguous blocks against the standard *MxN* approach.

The remainder of this work is organized as follows. Section 2 presents previous works in processor near memory and memory allocation methodologies. Section 3 provides an overview of the Micron IMI architecture and a rudimentary explanation of its programming model. Section 4 describes our approach and associated implementation. Section 5 provides an evaluation of our approach and Section 6 provides our final conclusions.

2 PREVIOUS WORK

Recent analysis of large-scale data center workloads has revealed that a large percentage of the total workload is actually spent allocating and managing memory. As result, several research efforts have sought to accelerate memory allocation for large scale data centers using modifications to the core SoC [11]. Further, additional

MCHPC'17, Denver, CO, USA

DOI: 10.1145/3145617.3145618

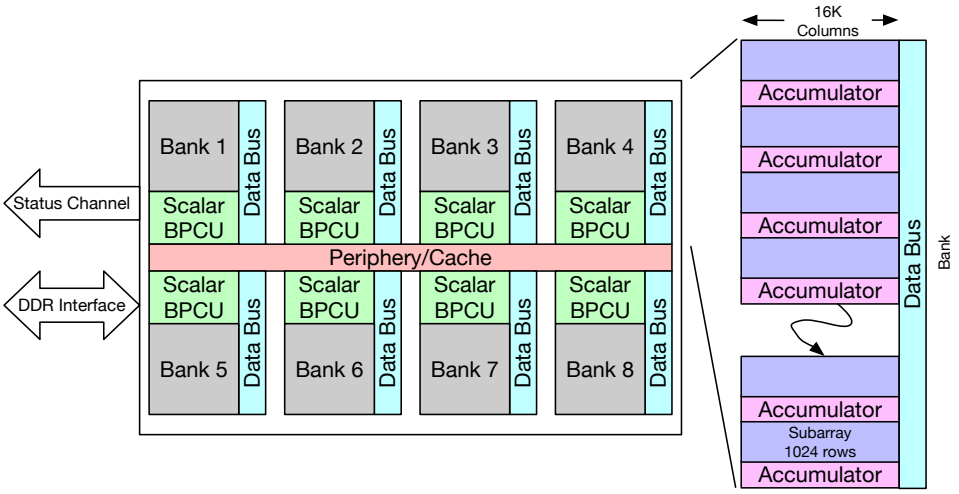

Figure 1: IMI Device Architecture

work has been performed accelerate basic software-driven allocations for common X86 platforms using similar techniques found in hardware transactional allocation systems [14]. These techniques show interesting promise beyond start of the art allocation methods such as DLmalloc [1], JEmalloc [7], Hoard [3] and TBBmalloc [13].

In addition to traditional system architecture memory allocation, quite a number of efforts have focused on allocating memory for special purpose or accelerator architectures. One of the more advantageous uses of exotic memory allocation schemas is for non-volatile devices [21]. Given the orthogonality of the paging, interleaving and device protocols, special purpose allocators for non-volatile devices have proven to both accelerate allocation performance and reduce wear on the internal device storage. Further, several efforts have focused on optimizing allocation schemas for hardware-based transactional memories [10] [4]. These allocators have the unique requirement of initializing additional hardware device state alongside the physical bit storage.

Processing in memory (PIM) architectures have also been the topic of many research efforts. Most recently, the Micron Automata Processor builds upon standard 2D DRAM methods with an extended dataflow execution engine that yields advantageous performance for combinatorial applications [5]. 3D stacked devices have also been the focus of several research efforts to bury computing capabilities in the logic layer [18]. Additionally, many academic efforts, such as the Terasys [9], the Execube [12] and the Logic-in-Memory compute [20] have published interesting near-memory computing paradigms. However, only a small number of projects, such as the Computational RAM [6] and the Berkeley VIRAM [19], have published results based upon a traditional DRAM process.

3 PIM ARCHITECTURE

3.1 Core Architecture

The Micron *In-Memory Intelligence*, or *IMI*, device is a new processing in memory DRAM architecture that combines a classic bit-serial compute architecture built with a planar, 2D memory array and a scalar processing and control mechanism that resides in the device periphery logic [8]. We can see in Figure 1 that the IMI device architecture consists of four unique components. First, the device is interconnected to host compute elements via a 64 bit DDR4 compatible interface such that normal memory I/Os can occur adjacent to the extended processing capabilities. The device also has a sideband status channel for exchanging IMI-specific device feedback with host compute elements.

Next, the overall DRAM architecture is constructed using an 8 Gbit die divided into eight banks. Each bank contains 64 subarrays and is interconnected to the periphery I/O mechanisms using a 2 Kbit wide data bus. Each bank also hosts a scalar bank processing control unit, or *BPCU*. The BPCU is a VLIW architecture that serves as the scalar processing unit in a traditional vector or SIMD architecture.

Finally, each subarray (64 per bank) is constructed using 1,024 rows and 16,384 columns of traditional bit-level DRAM storage attached to a unique sense amp stripe. This sense amp contains an additional accumulator circuit that can perform functions such as *AND*, *NAND*, *NOR*, *OR*, *XOR*, *XNOR*, bit inversion and left/right shifting in as little as three row-cycles. In this manner, these rudimentary functions can be chained together in order to execute standard integer arithmetic, logical and boolean functions directly within the DRAM sense amp.

This unique processing capability is accessed via an equally unique SIMD instruction set architecture. First, the IMI instruction set virtualizes the physical locality of the row and column addresses via a virtual register file [8] that resembles a traditional long vector register file. Additionally, the physical BPCU registers are represented as scalar registers in the virtualized register definitions. These registers are manipulated in conjunction with a scalar and vector instruction set that resembles traditional vector ISA methodologies.

However, the IMI architecture has the unique ability to represent individual vector data elements as *vertical* or *horizontal* storage.

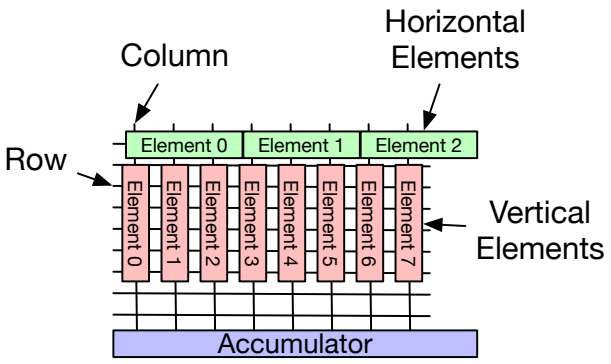

Figure 2: IMI Device Memory Orientation

Vertically stored elements are arranged vertically down an individual column where each subsequent bit is physically placed on the *N'th+1* row. Horizontal elements are placed within a row, where each subsequent bit is placed on the *N'th+1* column. Figure 2 depicts the vertical and horiztonal memory orientation. The maximum vector length (*VL*) in vertical mode is equivalent to the number of columns (16,384 per subarray). The maximum vector length in horizontal mode is the number of columns divided by the size of the element in bits. This duality of physical bit storage is echoed in our SIMD instruction set by explicitly specifying the mode for each instruction as *V* for vertical and *H* for horizontal. Additionally, operations can be performed under mask, true (*T*) or false (*F*). The IMI SIMD instruction mnemonics are denoted as follows.

```
OPERATION.[H|V].TYPE.[T|F]
```

3.2 Programming Model

As mentioned above, the IMI architecture is accessed via a pseudo-traditional SIMD instruction set. The instruction set provides traditional scalar flow control operations as well as a dual set of SIMD operations that exist in both the horizontal dimension and the vertical dimension. This programming model expresses this instruction set as a traditional SIMD or vector programming model. Vectorizable loops are collapsed into a series of vector blocks with prologue, compute and epilogue instructions. Further, loops with vectorizable conditional statements are echoed as operations under mask in the generated instruction stream.

However, traditional vector programming models do not have the implicit ability to represent the duality of the horizontal versus vertical arithmetic execution dimensions. As a result, we add an additional set of keywords, herein referred to as *_vertical* and *_horizontal*, to the C/C++ language that permit users to annotate variable definitions and pointers such that the compiler has inherent knowledge of the target dimension. The underlying implementation of the dimensional keywords is the application of unique address spaces as defined by the C99 specification. As a result, each annotated variable or pointer resides in the desired address space such that the compiler now knows which type of IMI vector instructions, horizontal or vertical, to emit during code generation. We implement the additional keywords, their associated address spaces, any unique optimization passes and the code generation methods in the LLVM compiler infrastructure [15].

Consider the following example. We have a vector loop that performs an *axpy* operation across vectors of length *LEN*. Notice that we utilize the *_vertical* keyword to specify that the *A*, *B* and *R* vectors in the vertical dimension. In this manner, each element of the individual vectors is allocated vertically down a single column in the IMI array. It is not necessary to force the scalar variable *Q* in a target address space as the instruction set inherently handles the vector-scalar-vector compute model.

```
_vertical uint64_t A[LEN];
_vertical uint64_t B[LEN];
_vertical uint64_t R[LEN];
uint64_t Q, i;

for( i=0; i<LEN; i++ ){
   R = Q * A[i] + B[i];
}
```

Further information regarding the IMI core architecture and programming model can be found in the seminal publication [8].

4 BIT CONTIGUOUS MEMORY ALLOCATION

4.1 Requirements

Given the orthogonality of the aforementioned hardware and programming model, the sheer locality required to support efficient operation of the IMI device presents an interesting set of requirements. We summarize these requirements as follows:

Bit Continuity: The IMI processing methodology inherently requires the adjacent bit values within a single element as well as adjacent elements must be co-located within the same physical sub-component. In this manner, traditional carry chain arithmetic and full SIMD operations can be operated within a single IMI bank construct. As a result, the memory allocation schema must depart from traditionally interleaved allocation mechanisms and adopt a rigid, bit-contiguous schema that ensures adjacent bit and element values are physically co-located.

Horizontal Allocation: Programmers naturally visualize the allocation of member data items as horizontally arranged blocks. As a result, we find that users often initially arrange IMI data members as horizontal blocks during the initial stages of porting applications. As a result, we must retain the ability to allocate horizontal data items in the IMI programming environment. This, despite the fact that the maximum horizontal vector length (in elements) is the number of bits in a single row in a single bank modulo the size of the element.

Vertical Allocation: In addition to the aforementioned horizontal allocation capability, we also seek to support the ability to allocate vectors of vertical elements. In this manner, individual elements in a vector are arranged vertically down individual columns where the $N'th+1$ bit is arranged on the $N'th+1$ row. The goal of this allocation and operational model is to support the maximum potential vector length, or the total number of columns in an IMI bank.

Minimize Algorithmic Complexity: One the of the primary initial concerns of allocating memory for the IMI device was the algorithmic complexity of searching the device space for unallocated

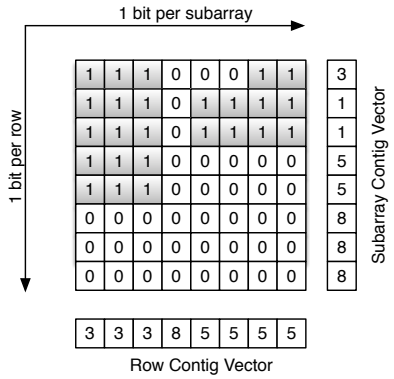

Figure 3: IMI Memory Allocation Representation

blocks. Consider a device arranged using R rows and C columns. Regardless of whether the desired memory blocks are organized as horizontal or vertical vectors, the maximum potential search space for the target block is the total two-dimensional physical device space, or $R \times C$. As a result, for any allocation requesting R' rows and C' columns, there is a nonzero probability that the entire array much be searched with a complexity of $O(R' \times C') = (R \times C)$. Our approach seeks to reduce the maximum potential search space such that $O(R' \times C') < R \times C$.

Maximize Use of the Array Resources: As with many traditional memory allocation methodologies, we seek to maximize the usage of the memory space by minimizing fragmentation. In this manner, we seek to maximize the probability by which we can allocate and co-locate data variables that will subsequently be utilized in the same computation.

Co-Location of Similar Variables: Finally, the target memory allocation methodology for the IMI device architecture must support the ability to co-locate like-minded variables or memory regions. Consider the standard *AXPY* operation, $R = \alpha \times X + Y$, where X and Y are vectors. It is inherently more efficient to co-locate the vectors at allocation time if possible in order to avoid any unnecessary data movement operations. This assumes additional programming model hints and compiler optimizations in order to achieve this co-locality at runtime. However, we feel that the ability to co-locate or derive pseudo-physical locality is an inherent design requirement.

4.2 Algorithmic Approach

Given the aforementioned set of fundamental requirements, we devised a combination of sparse matrix and dense vector techniques by which to represent and subsequently manage the IMI device memory allocations. The approach combines two basic mathematical models in order to efficiently model and map the allocated (or unallocated) blocks in the IMI array.

First, we utilize a sparse matrix in order to represent the allocated or unallocated regions of the IMI array. For this, our matrix stores a single bit for each row in each individual subarray. In this manner, the smallest delineation of memory that can be allocated in our approach is a single row in a single subarray. Zeroes in the matrix represent unallocated rows and one designates rows that have been allocated. Using this, we have the ability to map the horizontal or vertically allocated blocks.

However, the sparse matrix alone does not significantly reduce the complexity required to search and subsequently allocate new blocks of memory in the IMI array. In order to further reduce the complexity required to search for unallocated blocks, our methodology utilizes a set of two dense vectors, or *contig* vectors. The first vector, herein referred to as the *subarray contig vector*, is comprised of unsigned integer elements equal to the number of rows in the IMI device configuration. Each element in the subarray contig vector holds the largest contiguous block of unallocated subarrays in the designated row.

Similarly, we also create a *row contig vector*. The row contig vector is comprised of unsigned integer values, one element per subarray. This vector holds the largest contiguous block of unallocated rows in the target subarray. We depict the full sparse matrix and dense vector representation of the IMI array in Figure 3. Given this, we can calculate that the total space required to store the matrix and the associated vectors for an array with R rows and S subarrays is $((R \times S)bits + (R + S)unsigned\ integers)$.

Given this representation, we utilize a two stage process by which to search and subsequently allocate blocks of memory. When applications request blocks of memory in horizontal or vertical mode, the allocation API converts the request into the required number of rows and subarrays. This is what is utilized for our reduced search space algorithm. The first stage utilizes the subarray and row contig vectors to perform a rudimentary search in order to determine if there is a nonzero probability that there is sufficient space to allocate the desired block. For S' requested subarrays and R' requested rows, we search the subarray and row contig vectors, respectively, in order to find the first intersecting block of unallocated space that is large enough to contain the requested allocation. This intersecting space can be defined as p subarray bits and q row bits. The maximum search complexity of our contig vectors given S subarrays and R rows is $O(S + R)$.

After a successful initial search of the contig vectors, the algorithm moves to the second stage of allocation in a nest fashion. This stage performs an ancillary search of the sparse matrix cross section found in the first stage. This portion of the search is dense. Given a cross section of p subarray bits and q row bits, we perform a search of $O(p \times q)$ of the sparse matrix representation for the first exacting location suitable to hold requested our allocation. Upon a successful search, the sparse matrix bits representing the allocated space are marked as utilized and the contig vectors are each updated based upon the updated continuity of unallocated space.

As a result, for an array containing R rows, C columns and S subarrays and given a memory request that requires p subarrays and q rows, we may define the total search complexity (β) in Equation 1.

$$\beta = O((p \times q) + (S + R)) \tag{1}$$

Finally, we can say that the search complexity of our approach is less than the basic multiplicative search method. In this manner $\beta < O(R \times C)$.

As an example, consider a request from the application to requests two rows across four contiguous subarrays. Also consider

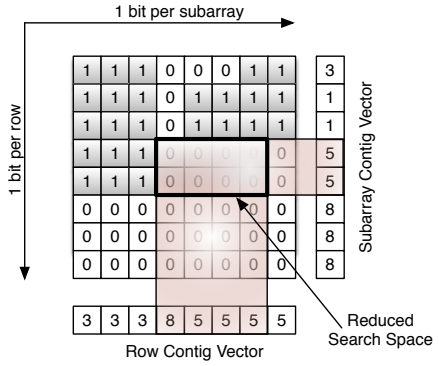

Figure 4: IMI Memory Allocation Example

the initial sparse matrix allocation represented in in Figure 3. We can see that the majority of the allocations have been concentrated on the leftmost and upper most blocks (represented as 1's). We can also see that contig vectors reflect the necessary contiguous blocks of unallocated space. When performing the first stage of the search, we find the first candidate block that holds sufficient contiguous space in both the subarray contig vector and the row contig vector. Note the intersection of this space depicted in Figure 4. We can see that our new, reduced matrix search space is a 4x2 rectangular region. Upon a successful secondary search, these values will be marked as allocated (1) and the contig vectors will be updated to correspond to the new array configuration.

4.3 Implementation

We implement our aforementioned approach in a simple, C library that is designed to mimic a portion of the standard malloc interfaces. The library, herein referred to as the *bmem* library, contains two primary sets of interfaces that represent the initialization/destruction routines and the memory management routines.

One of the interesting idiosyncracies of developing system software for processing near memory architectures is that we often find disparate system configurations with different memory capacities. Given the co-locality of the processing elements within the sense amp stripe, differing capacities of IMI devices also implies differing degrees of compute capacity [8]. As a result, the *bmem* library provides an initialization routine, *bmem_init* that permits users or other system software routines to initialize the allocation environment based upon the row, column, subarray and bank dimensions of the target IMI device or devices.

Once the library has been initialized, applications may request memory blocks using the *bmem* allocation interfaces. The library provides two basic allocation interfaces and a single interface to free allocated blocks. The most prominent of the two allocation interfaces mimics the standard malloc interface with an additional argument. The additional argument receives an enumerated type, *BMEM_DIM*, that specifies the target major dimension of the memory, horizontal or vertical. Finally, there is a single allocation interface that permits users to request memory in an abstract, or *raw*, dimension. This interface requires that the user request the memory block in terms of the number of subarrays and rows, as opposed to the number of bytes. This permits users to optimize the locality of special member variables such as single-bit booleans, non byte-aligned integers and raw floating point variables.

An example of using the *bmem* library implementation is below.

```
// EXAMPLE BMEM LIB USAGE

// variable definitions
_horiztonal void *ptr1;
_vertical void *ptr2;
_horizontal void *ptr3;

// init a device with 1024 rows x 65536 columns
// per subarray and 16 subarrays per
// bank and 8 banks
bmem_init( 1024, 65536, 16,  8 );

// allocate a horizontal block of 4096 bytes
ptr1 = bmem_malloc( 4096, HORI );

// allocate a vertical block of 128 bytes
ptr2 = bmem_malloc( 128, VERT );

// allocate 2 subbarrays with 17 rows
ptr3 = bmem_malloc_raw( 2, 17 );

// free the pointers
bmem_free( ptr1 );
bmem_free( ptr2 );
bmem_free( ptr3 );

// destructor
bmem_dtor();
```

5 EVALUATION

5.1 Benchmarks

In order to validate our work, we utilize a set of two benchmarks to exhibit the performance ramifications of the approach. Given the rather orthogonal nature of the underlying hardware implementation, it was not realistic to compare the actual generated layout of the memory subsystem using traditional allocators such as GNU malloc [17] and JEmalloc [7] against the approach described herein. However, we can compare the basic performance ramifications of our approach versus the GNU malloc implementation.

The first benchmark set, herein referred to as the *Full Array* benchmark, compares the performance of the traditional GNU malloc implementation against our bit contiguous approach when allocating an entire IMI device. The benchmark includes both a *horizontal* and a *vertical* component. Each of the components iteratively allocates blocks of memory until the entire simulated IMI device is saturated. These benchmarks demonstrate two important features of the bit contiguous approach. First, they demonstrate that the approach has the ability to efficiently allocate the entire IMI device with full saturation and zero fragmentation. Second, they demonstrate the relative performance against the same series of allocations using the GNU malloc implementation. For each of the horizontal and vertical versions of the benchmark, we record the total runtime and derive the per-iteration runtime. We summarize

Table 1: Full Array Benchmark Configuration

Dimension	Loop Iters	Bytes Per Iter
Horizontal	1024	1048576
Vertical	128	67108864

Table 2: Malloc-test Benchmark Configurations

Dimension	Threads	Size (Bytes)	Requests Per Thread
Horizontal	1,2,4,8	8	10000
Horizontal	1,2,4,8	64	10000
Horizontal	1,2,4,8	128	10000
Horizontal	1,2,4,8	512	10000
Horizontal	1,2,4,8	1024	10000
Horizontal	1,2,4,8	4096	10000
Vertical	1	8	128
Vertical	2	8	64
Vertical	4	8	32
Vertical	8	8	16
Vertical	1	16	128
Vertical	2	16	64
Vertical	4	16	32
Vertical	8	16	16
Vertical	1	32	128
Vertical	2	32	64
Vertical	4	32	32
Vertical	8	32	16
Vertical	1	64	128
Vertical	2	64	64
Vertical	4	64	32
Vertical	8	64	16
Vertical	1	128	128
Vertical	2	128	64
Vertical	4	128	32
Vertical	8	128	16

the relevant numerical aspects of the Full Array benchmarks in Table 1.

The second benchmark, herein referred to as the *malloc-test*, is comprised of a modified version of the benchmark code utilized in the Lever (et al.) multithreaded Linux malloc benchmark study [17]. Unlike the Full Array benchmark, the multithreaded tests perform an allocation in the specified dimension and a complementary *free* within the loop. As a result, we measure the performance of rapid allocation and deallocations across the series of iterations for each thread. The benchmark permits the user to specify the size of each individual allocation, the target number of iterations and the number of threads in the simulation. The full set of benchmark configurations used in our results is detailed in Table 2. Note that the configurations differ between the horizontal and vertical variants. Given the limited memory available in each column, the vertical benchmark must naturally be different than the complementary horizontal inputs.

The system utilized for each of the aforementioned benchmarks is a Dell CS Cloud server with two sockets populated with six core AMD Opteron 2419 processors with a core frequency of 1.8Ghz. The system was configured with Ubuntu 16.04.03 LTS. Each of the benchmarks was compiled with the default gcc 5.4.0 compiler stack with the standard *-O3* optimizations. The IMI DRAM device used for each of the aforementioned benchmarks is configured using the parameters in Table 3.

Table 3: IMI Device Benchmark Configuration

Rows	Columns	Subarrays	Banks	Size
1024	65536	16	8	1GB

5.2 Results

Given the bit continuity requirements of each allocation for the IMI devices, we can predict that the performance penalty of each allocation will be higher than normal GNU mallocs. This is evident in our Full Array benchmark results. Figure 5 depicts the results of our methodology compared to the standard GNU malloc implementation when allocating a single IMI array's physical storage.

We can see in Figure 5a that the total runtime of the bit contiguous allocator is measurably longer than the standard GNU implementation in both horizontal and vertical mode. We see that the GNU method is 1.48X faster when executing in horizontal mode and 5.86X faster when executing in vertical mode. While this may appear to imply poor results, we can see in previous publications [8] that the relative IMI device performance implications far outweigh the initial costs of the memory allocation as compared to the GNU methodology.

However, we also see that the cost of the horizontal allocation is higher than that of the vertical allocation. The total runtime of the bit contiguous horizontal allocation is 18.43X higher than the full array vertical allocation. Despite the 8X reduction in required iterations to allocate the entire device, the vertical allocator is still 2.66X faster per iteration (Figure 5b).

Unlike the Full Array tests, the malloc-test benchmarks utilized multiple threads in order to test the library's ability to provide scalable and concurrent memory allocations. The first set of tests examined the horizontal allocation scalability using 1 to 8 threads and 8 to 4096 byte allocations. As we can see in Figure 6, the timing for each of the request sizes scales at a rate that is a function of the number of threads and the respective allocation size. We can derive from this data that, on average, the bit memory request methodology requires 0.0002 seconds or approximately 200 microseconds per byte, per thread in order to allocate a block of memory in horizontal mode.

This trend is also echoed in Figure 7 where we plot the average cost per thread per request size. We can see that the scale and curvature of the graph matches the timing results in Figure 6. Across all the memory request sizes, the bit memory library is approximately 71.2% slower than the purely linear scale. Further analysis shows us that the horizontal per-thread timing is directly is closely correlated to the thread scaling and less so to the memory request size scaling. We can see from Table 4 that increasing the thread count from 2 to 8 increases the per-thread timing beyond the linear scale from 39.83% to 92.73%. Conversely, for horizontal requests of 128 bytes or less, we see that on average the per-thread scalability is 56.8%

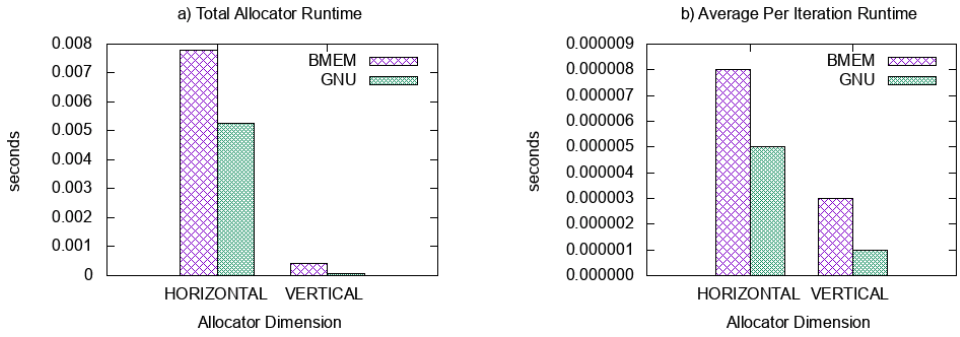

Figure 5: Full Array Saturation Results

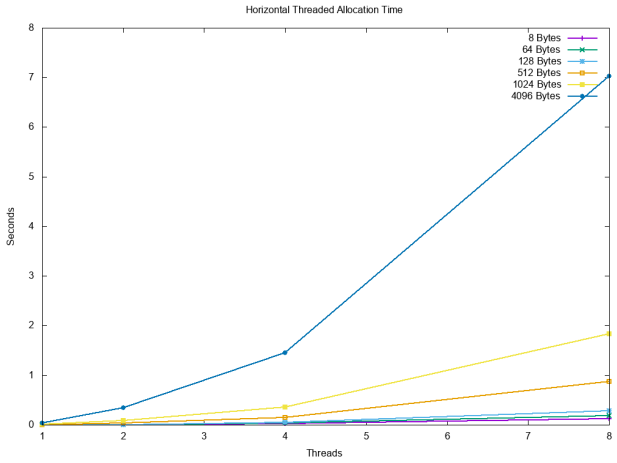

Figure 6: Horizontal Threaded Allocation Timing

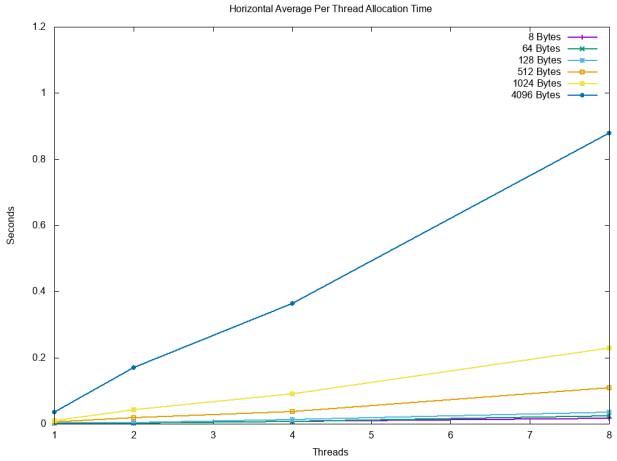

Figure 7: Horizontal Average Per Thread Timing

beyond linear. For requests larger than 128 bytes, the per-thread scalability is 85.62% beyond linear. As a result, we can conclude

Table 4: Horiztonal Sub-Linear Request Scaling

Average Correlator	Percent Above Linear
2 Threads	39.83%
4 Threads	81.11%
8 Threads	92.73%
8 Byte Requests	58.63%
64 Byte Requests	54.82%
128 Byte Requests	57.00%
512 Byte Requests	81.86%
1024 Byte Requests	86.71%
4096 Byte Requests	88.31%

that further work must be performed in order to optimize the per-thread scalability of the internal library implementation. Additional algorithmic work such as an initial, *halo* allocation pass may be required in order to provide better parallel scalability.

Finally, we may also perform a similar analysis of the vertical allocations performed using the malloc-test benchmarks. The results of these tests in vertical mode are quite different than the poor performance of horizontal mode. We see in Figure 8 that the scalability of vertical allocations is rather stable. This is partially due to the method that we utilized to evaluate vertical allocation mode. Vertical allocations are governed by the number of rows and the number of subarrays in the respective IMI device configuration. We can see that the total timing is a function of the size of the allocation, as opposed to the number of threads. On average, we can derive that across all thread counts and all request sizes, that vertical allocations require approximately 0.000005 seconds or 5 microseconds per byte. This is approximately 40X faster than our horizontal tests.

Finally, in Figure 9, we can see that the per-thread timing scale is quite good. As we increase the number of threads, the per-thread cost is reduced. We cannot, however, attribute this to the algorithm's implementation. In a DRAM CAS chain, we are not permitted to address individual columns from user space. The IMI array provides users the ability to *mask* columns as not to write their result of an operation using traditional operations under mask. However, we cannot explicitly allocate an individual column within a subarray. As a result, each vertical allocation's minimum allocation is actually the number of rows required to fulfill the vertical allocation across

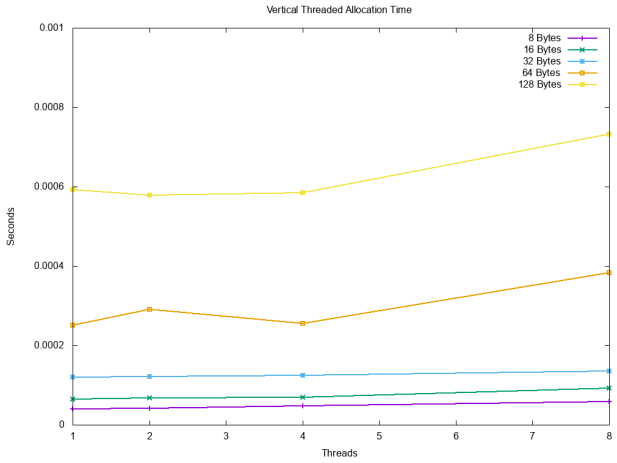

Figure 8: Vertical Threaded Allocation Timing

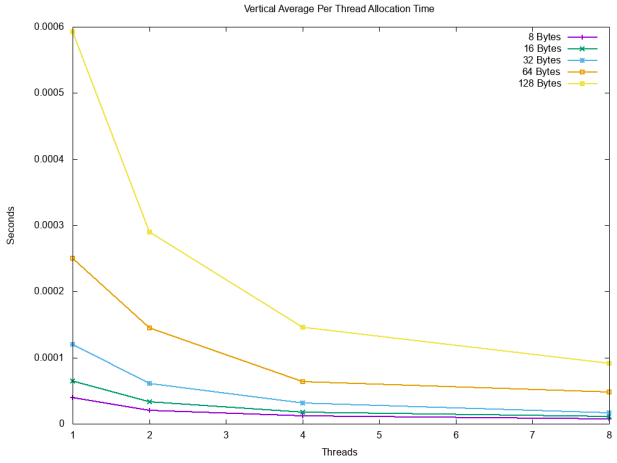

Figure 9: Vertical Average Per Thread Timing

the entire subarray. As such, as we increase the number of threads in our vertical tests, we are forced to decrease the number of iterations per thread in order to avoid exhausting the physical memory space of a single array. The result is vertical allocations appear to be more efficient per thread.

6 CONCLUSION

In this work, we have introduced a novel memory allocation methodology designed to incorporate the idiosyncratic requirements of the Micron IMI processing in memory device architecture. This approach utilizes a combination of a sparse matrix representation in association with two dense, *contig* vectors to significantly reduce the computational complexity of searching the IMI array for free space for new memory allocations. The result being an approach that is significantly more efficient than the base multiplicative search algorithm. Further, this approach permits users and applications to allocate memory both horizontally and vertically in order to optimize the respective data arrangement for the target algorithm.

We utilized a set of two benchmarks in order to evaluate our approach. The first benchmark is designed to deliberately allocate the entire array in both horizontal and vertical modes. The second benchmark is based upon a classic memory allocation benchmark that measures the performance of various memory allocation sizes across an increasing number of threads. We find in these tests that the bit contiguous results are measurably slower than the traditional GNU malloc implementation. However, given the significant performance benefits of the IMI device, such as the 360X speedup on SHA1 hashing over a traditional Intel Xeon platform [8], we can conclude that our allocation mechanisms do not significantly detract from the total system performance. However, we also find that future work can be done in order to improve the scalability of the implementation, especially when allocating memory in horizontal mode.

ACKNOWLEDGMENTS

This work was originally performed under the auspices of the IMI group within Micron Technology Inc's Advanced Memory Systems Group [2]. We acknowledge that the intellectual property [16] associated with this work is currently held by Micron. We would also like to acknowledge Dr. Kevin Wadleigh for his help and advice in developing the methodologies described herein.

REFERENCES

[1] *Some storage management techniques for container classes.* Technical Report.
[2] 2017. Micron Technology, Inc. (2017). http://www.micron.com Accessed: 2017-07-18.
[3] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 2000. Hoard: A Scalable Memory Allocator for Multithreaded Applications. *SIGOPS Oper. Syst. Rev.* 34, 5 (Nov. 2000), 117–128. DOI: https://doi.org/10.1145/384264.379232
[4] Thomas Carle, Dimitra Papagiannopoulou, Tali Moreshet, Andrea Marongiu, Maurice Herlihy, and R. Iris Bahar. 2016. Thrifty-malloc: A HW/SW Codesign for the Dynamic Management of Hardware Transactional Memory in Embedded Multicore Systems. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES '16)*. ACM, New York, NY, USA, Article 20, 10 pages. DOI: https://doi.org/10.1145/2968455.2968513
[5] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes. 2014. An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing. *IEEE Transactions on Parallel and Distributed Systems* 25, 12 (Dec 2014), 3088–3098. DOI: https://doi.org/10.1109/TPDS.2014.8
[6] D. G. Elliott, W. M. Snelgrove, and M. Stumm. 1992. Computational Ram: A Memory-simd Hybrid And Its Application To Dsp. In *1992 Proceedings of the IEEE Custom Integrated Circuits Conference*. 30.6.1–30.6.4. DOI: https://doi.org/10.1109/CICC.1992.591879
[7] Jason Evans. 2006. A Scalable Concurrent malloc(3) Implementation for FreeBSD. (2006).
[8] T. Finkbeiner, G. Hush, T. Larsen, P. Lea, J. Leidel, and T. Manning. 2017. In-Memory Intelligence. *IEEE Micro* 37, 4 (2017), 30–38. DOI: https://doi.org/10.1109/MM.2017.3211117
[9] Maya Gokhale, Bill Holmes, and Ken Iobst. 1995. Processing in Memory: The Terasys Massively Parallel PIM Array. *Computer* 28, 4 (April 1995), 23–31. DOI: https://doi.org/10.1109/2.375174
[10] Richard L. Hudson, Bratin Saha, Ali-Reza Adl-Tabatabai, and Benjamin C. Hertzberg. 2006. McRT-Malloc: A Scalable Transactional Memory Allocator. In *Proceedings of the 5th International Symposium on Memory Management (ISMM '06)*. ACM, New York, NY, USA, 74–83. DOI: https://doi.org/10.1145/1133956.1133967
[11] Svilen Kanev, Sam Likun Xi, Gu-Yeon Wei, and David Brooks. 2017. Mallacc: Accelerating Memory Allocation. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 33–45. DOI: https://doi.org/10.1145/3037697.3037736

[12] Peter M. Kogge. 1994. EXECUBE-A New Architecture for Scaleable MPPs. In *Proceedings of the 1994 International Conference on Parallel Processing - Volume 01 (ICPP '94)*. IEEE Computer Society, Washington, DC, USA, 77–84. DOI: https://doi.org/10.1109/ICPP.1994.108

[13] Alexey Kukanov. 2007. The Foundations for Scalable Multicore Software in Intel Threading Building Blocks. 11 (11 2007).

[14] Bradley C. Kuszmaul. 2015. SuperMalloc: A Super Fast Multithreaded Malloc for 64-bit Machines. *SIGPLAN Not.* 50, 11 (June 2015), 41–55. DOI: https://doi.org/10.1145/2887746.2754178

[15] Chris Arthur Lattner. 2002. *LLVM: An infrastructure for multi-stage optimization.* Ph.D. Dissertation. University of Illinois at Urbana-Champaign.

[16] John D. Leidel and Kevin Wadleigh. 2016. Multidimensional contiguous memory allocation. (07 04 2016). http://appft1.uspto.gov/netacgi/nph-Parser?Sect1=PTO1&Sect2=HITOFF&d=PG01&p=1&u=/netahtml/PTO/srchnum.html&r=1&f=G&l=50&s1=20160098209.PGNR.

[17] Chuck Lever and David Boreham. 2000. Malloc() Performance in a Multithreaded Linux Environment. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '00)*. USENIX Association, Berkeley, CA, USA, 56–56. http://dl.acm.org/citation.cfm?id=1267724.1267780

[18] R. Nair, S. F. Antao, C. Bertolli, P. Bose, J. R. Brunheroto, T. Chen, C. Y. Cher, C. H. A. Costa, J. Doi, C. Evangelinos, B. M. Fleischer, T. W. Fox, D. S. Gallo, L. Grinberg, J. A. Gunnels, A. C. Jacob, P. Jacob, H. M. Jacobson, T. Karkhanis, C. Kim, J. H. Moreno, J. K. O'Brien, M. Ohmacht, Y. Park, D. A. Prener, B. S. Rosenburg, K. D. Ryu, O. Sallenave, M. J. Serrano, P. D. M. Siegl, K. Sugavanam, and Z. Sura. 2015. Active Memory Cube: A processing-in-memory architecture for exascale systems. *IBM Journal of Research and Development* 59, 2/3 (March 2015), 17:1–17:14. DOI: https://doi.org/10.1147/JRD.2015.2409732

[19] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. 1997. A Case for Intelligent RAM. *IEEE Micro* 17, 2 (March 1997), 34–44. DOI: https://doi.org/10.1109/40.592312

[20] Harold S. Stone. 1970. A Logic-in-Memory Computer. *IEEE Trans. Comput.* 19, 1 (Jan. 1970), 73–78. DOI: https://doi.org/10.1109/TC.1970.5008902

[21] Songping Yu, Nong Xiao, Mingzhu Deng, Fang Liu, and Wei Chen. 2017. Redesign the Memory Allocator for Non-Volatile Main Memory. *J. Emerg. Technol. Comput. Syst.* 13, 3, Article 49 (April 2017), 26 pages. DOI: https://doi.org/10.1145/2997651