Beyond 16GB: Out-of-Core Stencil Computations

István Z Reguly (PPCU ITK, Hungary) - reguly.istvan@itk.ppke.hu
Gihan R Mudalige (University of Warwick)
Michael B Giles (University of Oxford)

MCHPC'17: Workshop on Memory Centric Programming for HPC
12/11/2017, Denver
Fast stacked memory

- GPUs come with small but fast on-board or on-chip memory
  - Small: 2-16 GB
  - Fast: 100-800 GB/s
  - PCI-e bottleneck: 16 GB/s
  - IBM + NVLink cards: 40-80 GB/s
- Lately Intel’s CPUs as well: Knights Corner and Knights Landing
  - Small: 6-16 GB
  - Fast: 200-500 GB/s
  - PCI-e bottleneck, or with KNL DDR4 with about 90 GB/s
- Need high amounts of data re-use or very high computational intensity to make the transfer worth it
  - Up to 50x for bandwidth, or 2500 flop for compute
Problem scaling

• What happens if my problem is larger than fast memory?
  • Use fast memory as a cache
    • Only really feasible on Intel’s Knights Landing – works well, graceful degradation, at 48 GB 2-5x slower
    • Managed Memory on Pascal and later GPUs theoretically allows this, but it’s not intended for that use, performance is not great
    • For GPUs, PCI-e is just too much of a bottleneck
  • Data streaming applications
    • Triple buffering – upload, compute, download
    • Need lots of reuse/compute
  • Cache blocking tiling
    • Lots of research targeting CPU caches, stencil and polyhedral compilers
    • Fairly limited in scope, particularly big problem for large-scale applications
      • Lots of data per gridpoint
      • Operations scattered across many compilation units
      • Data-driven execution
Cache Blocking Tiling

- Given a sequence of loops and their access patterns, split their iteration spaces and reorder their execution so data used fits in cache.
Cache Blocking Tiling

- Given a sequence of loops and their access patterns, split their iteration spaces and reorder their execution so data used fits in cache

- For the applications we are interested in, no compiler can do this...
The OPS DSL for Structured Meshes

- **Blocks**
  - A dimensionality, no size
  - Serves to group datasets together
    ```
    ops_block = ops_decl_block(dim, name);
    ```

- **Datasets on blocks**
  - With a given arity, type, size, optionally stride
    ```
    ops_dat = ops_decl_dat(block, arity, size, halo, ..., name);
    ```

- **Stencils**
  - Number of points, with relative coordinate offsets, optionally strides
    ```
    ops_stencil = ops_decl_stencil(dim, npoints, points, name);
    ```

---

Hardware through such high-level frameworks. Research published as a result of this work includes a number of performance analysis studies on standard CFD benchmark applications as well as a full industrial-scale application from the production workload at Rolls-Royce plc. OP2 uses an “active library” approach where a single application code written using the OP2 API can be transformed into different parallel implementations which can then be linked against the appropriate parallel library enabling execution on different back-end hardware platforms. At the same time the generated code from OP2 and the platform-specific back-end libraries are highly optimized utilizing the best low-level features of a target architecture to make an OP2 application achieve near-optimal performance including high computational efficiency and minimized memory traffic.

With the success of OP2, our aim is to apply the same strategy in developing high-level frameworks for another key class of applications, multi-block structured mesh applications, that require significant developer effort to parallelize on modern hardware systems. The resulting research and development is being carried out under the EPSRC-funded OPS project (Oxford Parallel Library for Structured-mesh solvers). Multi-block structured mesh applications can be viewed as an unstructured collection of structured mesh blocks. Thus the starting point of our work was to adopt the high-level framework and code generation strategy from OP2 and apply it for the development and solution of single block structured mesh applications.

OPS is designed to appear as a classical software library with an API. It is currently domain specific to the single block-structured mesh problems and will later be extended to multi-block structured problems. Much of the API and library follows the design of the OP2 high-level library for unstructured mesh applications. However, the structured mesh domain is distinct from the unstructured mesh applications domain due to the implicit connectivity between neighboring mesh elements (such as vertices, cells) in structured meshes/grids. The key idea is that operations involve looping over a “rectangular” multi-dimensional set of grid points using one or more “stencils” to access data. The next section illustrates the OPS API using examples from CloverLeaf.

## 2.1 The OPS API

The CloverLeaf mini-app involves the solution of the compressible Euler equations, which form a system of three partial differential equations. The equations are statements of the conservation of energy, density and momentum and are solved using a finite volume method on a structured staggered grid. The cell centers hold internal energy and density while nodes hold velocities. The solution involves an explicit Lagrangian step using a predictor/corrector method to update the hydrodynamics, followed by an advective remap that uses a second order Van Leer up-winding scheme. The advective remap step returns the grid to its original position. The original application is written in Fortran and operates on a 2D structured mesh. It is of fixed size in both x and y dimensions.

OPS separates the specification of such a problem into four distinct parts: (1) structured blocks, (2) data defined on blocks, (3) stencils defining how data is accessed and (4) operations over blocks. Thus the first aspect of declaring such a single-block structured mesh application with OPS is to define the size of the regular mesh over which the computations will be carried out. In OPS vernacular this is called an `ops_block`.

```c
ops_block = ops_decl_block(dim, name);
```

CloverLeaf works on a number of data arrays (or fields) which are defined on the 2D structured mesh (e.g., density, energy, x and y velocity of the fluid). OPS allows users to declare these using the:

```c
ops_dat = ops_decl_dat(block, arity, size, halo, ..., name);
```

```c
ops_stencil = ops_decl_stencil(dim, npoints, points, name);
```
The OPS DSL for Structured Meshes

• The description of computations follows the Access-Execute abstraction
• Loop over a given block, accessing a number of datasets with given stencils and type of access, executing a kernel function on each one
  • Principal assumption: order of iteration through the grid doesn’t affect the results

void calc(double *a, const double *b) {
    a[OPS_ACC0(0,0)] = b[OPS_ACC1(0,0)] + b[OPS_ACC1(0,1)] +
    b[OPS_ACC1(1,0)];
}

int range[4] = {12,50,12,50};
ops_par_loop(calc, block, 2, range,
ops_arg_dat(a,S2D_0,“double”,OPS_WRITE),
ops_arg_dat(b,S2D_1,“double”,OPS_READ));
Delayed execution

• Loop constructs describe operations and data accesses
• User contract: no side-effects, data only accessed through API calls
• When such a loop construct is called, we don’t have to execute immediately
  • Save all the information necessary for execution later into a data structure
  • When some data is returned to user space, then we have to execute all the queued operations – delayed evaluation
• This gives us an opportunity to analyse a number of loops together
  • Given a “loopchain”
  • Run-time dependency analysis and creation of a tiled execution scheme

• No changes to the user code
Runtime Tiling in OPS

• Given a sequence of loops, datasets accessed and their access patterns, we perform dependency analysis & construct execution plan
  1. First, we determine the union of all iteration ranges, and partition it into N tiles
  2. Looping over the sequence of computational loops in reverse order, we loop over each dimension and each tile
     1. Start index for the current loop, in the current dimension, for the current tile, is either the end index of the previous tile, or the start index of the original index set
     2. End index is calculated based on a read dependency of a loop with a higher index in the tile for any datasets written
     3. End index updated to account for write-after-read and write-after-write dependencies across tiles where the ordering will effectively change
     4. Based on the computed iteration range, the read and write dependencies of datasets are updated, accounting for the stencils used
• This algorithm is directly applicable to Intel Knights Landing
  • MCDRAM can be used as a cache, just like on CPUs

• For GPUs, there are two options
  • Rely on managed memory, and page migration – just like a cache + explicit prefetches
  • Use explicit memory management with async copies, kernel launches, etc.
  • Both require some extra logic
Managing transfers on GPUs

- Tiles shrink as we progress to later loops due to data dependencies
  - But also extend on the other side -> Skewed tiles
- Overlap in data accessed by adjacent tiles
  - Full footprint: all the data accessed by the tile
  - Left edge: data that is also accessed by the previous tile
  - Right edge: data that is also accessed by the next tile
Managing transfers on GPUs

• Triple buffering scheme
  • One for the current tile that is being computed
  • One for uploading the next tile
  • One for downloading the previous tile
  • Async memcopies can be fully overlapped in two directions + with compute using CUDA streams
  • Plus copy of “edge” data from one buffer to the next before execution of the current tile

• Is there enough data re-use to hide all the copies between CPU and GPU with kernel execution?
  • Tall order, given the ~40x bandwidth difference between PCI-e and Pascal’s memory
  • Can we reduce the amount of data to be transferred?
Reducing memory traffic

- We know how datasets are accessed – two trivial optimisations
  - Read-only data is not copied back to the CPU
  - Write-first data is not copied to the GPU
- “Cyclic” optimisation – temporary datasets
  - In many applications, there are datasets that are used as temporaries within a timestep, but do not carry information across them
  - In our OPS applications they are not explicitly marked as temporaries
  - Datasets that are written first in a loopchain are considered temporaries, and neither uploaded or downloaded
- Speculative prefetching
  - In most applications the same loopchains repeat
  - OPS does not know what the next loopchain will look like though
  - When processing the last tile, speculatively upload data needed for tile 0 of the next chain – based on tile 0 of the current loopchain
Stencil codes

• CloverLeaf 2D
  • Hydrodynamics Mini app in the Mantevo suite
  • Structured hydrodynamics solving compressible Euler equations
  • ~6k LoC
  • 25 variables per gridpoint, 30 different stencils
  • 83 different parallel loops, in 15 source files – lot of branching in & between parallel loops
  • Single time iteration: chain of 153 parallel loops

• CloverLeaf 3D
  • 3D version: 30 variables per gridpoint, 46 stencils, 141 parallel loops, chain of 603 in one time iteration

• OpenSBLI
  • Compressible Navier-Stokes solver, with shock-boundary layer interactions
  • 3D Taylor-Green vortex testcase: 29 variables, 9 stencils, 27 parallel loops, chain of 79 per iteration
  • No reductions – can tile across multiple time iterations & increase data reuse
Methodology

• Testing hardware:
  • Xeon Phi x200 7210 (64-core), cache mode, quadrant mode (4 MPI x 32 OpenMP)
  • Tesla P100 GPU 16 GB, PCIe in an x86 machine
  • Tesla P100 GPU 16 GB, NVLink in a Power8+ (Minsky) machine

• Problem scaling
  • CloverLeaf 2D: 8192*X -> X grows, 3D: 300*300*X, OpenSBLI: 300*300*X
  • For 6 GB to 48 GB total memory footprint

• Performance metric
  • Achieved “effective” bandwidth: for each loop, the number of datasets accessed * grid size / time
  • -> Bandwidth as seen by the user
Results on the Knights Landing

• Tiling performance + Flat mode MCDRAM/DDR4
• 314 (450) GB/s bandwidth to MCDRAM, 60 GB/s to DDR4
• Achieved:
  • DDR4: 50 GB/s, MCDRAM: 240 GB/s (4.8x)
  • No tiling & cache mode: steady degradation
  • With tiling: only slight degradation
    2.2x at 48GB
• Hit rates:

CloverLeaf 2D

<table>
<thead>
<tr>
<th>Problem Size (GB)</th>
<th>DDR4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6.3</td>
</tr>
<tr>
<td>0.5</td>
<td>7.8</td>
</tr>
<tr>
<td>1</td>
<td>9.4</td>
</tr>
<tr>
<td>5</td>
<td>12.5</td>
</tr>
<tr>
<td>10</td>
<td>14.1</td>
</tr>
<tr>
<td>15</td>
<td>15.7</td>
</tr>
<tr>
<td>20</td>
<td>17.2</td>
</tr>
<tr>
<td>25</td>
<td>18.8</td>
</tr>
<tr>
<td>30</td>
<td>20.4</td>
</tr>
<tr>
<td>35</td>
<td>21.9</td>
</tr>
<tr>
<td>40</td>
<td>23.5</td>
</tr>
<tr>
<td>45</td>
<td>25.0</td>
</tr>
<tr>
<td>50</td>
<td>26.6</td>
</tr>
<tr>
<td>55</td>
<td>28.2</td>
</tr>
<tr>
<td>60</td>
<td>30.5</td>
</tr>
<tr>
<td>65</td>
<td>34.4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem Size (GB)</th>
<th>flat MCDRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6.3</td>
</tr>
<tr>
<td>0.5</td>
<td>7.8</td>
</tr>
<tr>
<td>1</td>
<td>9.4</td>
</tr>
<tr>
<td>5</td>
<td>12.5</td>
</tr>
<tr>
<td>10</td>
<td>14.1</td>
</tr>
<tr>
<td>15</td>
<td>15.7</td>
</tr>
<tr>
<td>20</td>
<td>17.2</td>
</tr>
<tr>
<td>25</td>
<td>18.8</td>
</tr>
<tr>
<td>30</td>
<td>20.4</td>
</tr>
<tr>
<td>35</td>
<td>21.9</td>
</tr>
<tr>
<td>40</td>
<td>23.5</td>
</tr>
<tr>
<td>45</td>
<td>25.0</td>
</tr>
<tr>
<td>50</td>
<td>26.6</td>
</tr>
<tr>
<td>55</td>
<td>28.2</td>
</tr>
<tr>
<td>60</td>
<td>30.5</td>
</tr>
<tr>
<td>65</td>
<td>34.4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem Size (GB)</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6.3</td>
</tr>
<tr>
<td>0.5</td>
<td>7.8</td>
</tr>
<tr>
<td>1</td>
<td>9.4</td>
</tr>
<tr>
<td>5</td>
<td>12.5</td>
</tr>
<tr>
<td>10</td>
<td>14.1</td>
</tr>
<tr>
<td>15</td>
<td>15.7</td>
</tr>
<tr>
<td>20</td>
<td>17.2</td>
</tr>
<tr>
<td>25</td>
<td>18.8</td>
</tr>
<tr>
<td>30</td>
<td>20.4</td>
</tr>
<tr>
<td>35</td>
<td>21.9</td>
</tr>
<tr>
<td>40</td>
<td>23.5</td>
</tr>
<tr>
<td>45</td>
<td>25.0</td>
</tr>
<tr>
<td>50</td>
<td>26.6</td>
</tr>
<tr>
<td>55</td>
<td>28.2</td>
</tr>
<tr>
<td>60</td>
<td>30.5</td>
</tr>
<tr>
<td>65</td>
<td>34.4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem Size (GB)</th>
<th>Cache Tiled</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6.3</td>
</tr>
<tr>
<td>0.5</td>
<td>7.8</td>
</tr>
<tr>
<td>1</td>
<td>9.4</td>
</tr>
<tr>
<td>5</td>
<td>12.5</td>
</tr>
<tr>
<td>10</td>
<td>14.1</td>
</tr>
<tr>
<td>15</td>
<td>15.7</td>
</tr>
<tr>
<td>20</td>
<td>17.2</td>
</tr>
<tr>
<td>25</td>
<td>18.8</td>
</tr>
<tr>
<td>30</td>
<td>20.4</td>
</tr>
<tr>
<td>35</td>
<td>21.9</td>
</tr>
<tr>
<td>40</td>
<td>23.5</td>
</tr>
<tr>
<td>45</td>
<td>25.0</td>
</tr>
<tr>
<td>50</td>
<td>26.6</td>
</tr>
<tr>
<td>55</td>
<td>28.2</td>
</tr>
<tr>
<td>60</td>
<td>30.5</td>
</tr>
<tr>
<td>65</td>
<td>34.4</td>
</tr>
</tbody>
</table>

Average Bandwidth (GB/s)

Conclusion:
- High bandwidth achieved with tiling.
- Steady degradation without tiling.

11/12/2017
Results on the Knights Landing

**CloverLeaf 3D**

<table>
<thead>
<tr>
<th>DDR4</th>
<th>Flat MCDRAM</th>
<th>Cache</th>
<th>Cache tiled</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**OpenSBLI**

<table>
<thead>
<tr>
<th>DDR4</th>
<th>Flat MCDRAM</th>
<th>Cache</th>
<th>Cache tiled</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

11/12/2017
Results on the P100 GPU

• Best performance achieved with explicit memory management + full optimisations
  • Not enough data re-use for CloverLeaf, OpenSBLI 3 time iterations tiled
Effect of optimisations on P100

- Incrementally applying optimisations
  - No optimisations – 105 & 250 GB/s for PCI-e and NVLink
  - Enabling cyclic optimisations gives a big jump, as less than half of data needs to be moved – 215 & 370 GB/s for PCI-e and NVLink
  - Improves at larger problem sizes – lack of overlap for first tile
  - Enabling speculative prefetching of tile 0 improves performance at small sizes – 230 & 390 GB/s for PCI-e and NVLink
Managed Memory

• When relying on managed memory we allocate everything on the CPU, oversubscribing GPU memory - data is moved to the GPU on-demand, and pages are evicted with LRU when it gets full
• Very promising approach – could use GPU memory as a cache
• Page misses in GPU kernels are extremely expensive – high latency because everything is done in software (GPU driver)
  • Prefetches – API calls that tell the driver to migrate certain pages
  • Hints – such as read-only; can be duplicated and then discarded
• Prefetches take significant numbers of clock cycles
  • They block CPU, and it’s difficult to get copies and compute to overlap
  • When oversubscribing, throughput drops even more
Managed Memory

- Without prefetching, and < 16GB, it works great – above it hits the floor
- With prefetching we have to move more data (less ‘discarding’), plus it’s slower
  - Compounded by oversubscription issue
- Being looked at by NVIDIA engineers – expected to significantly improve soon
Managed Memory

- Latest updates from NVIDIA – thanks to Nikolay Sakharnykh & driver team
Summary

- Out-of-core requires significant data reuse that many applications do not have straight away
  - For stencil computations, tiling algorithms can improve this
  - No tiling compiler/framework before targeting GPUs and out-of-core, and even ones targeting CPUs are not applicable to long chains of loops
- Tiling algorithms deployed at run-time through the OPS eDSL
  - Key technique: delayed evaluation of parallel loops
- Explicit memory management algorithms for GPUs
  - Using a number of optimisations to reduce data movement
- Excellent problem scaling on KNL and P100
- Managed memory hopes