Collective Memory Transfers for Multi-Core Chips

George Michelogiannakis, Alexander Williams, Samuel Williams, John Shalf
Lawrence Berkeley National Laboratory, 1 Cyclotron Road, Berkeley, CA 94720
Email: {mihelog,awilliams,swwilliams,jshalf}@lbl.gov

ABSTRACT

Future performance improvements for microprocessors have shifted from clock frequency scaling towards increases in on-chip parallelism. Performance improvements for a wide variety of parallel applications require domain decomposition of data arrays from a contiguous arrangement in memory to a tiled layout for on-chip L1 caches and scratchpads. However, DRAM performance suffers under the non-streaming access patterns generated by many independent cores. In this paper, we propose collective memory scheduling (CMS) that uses simple software and inexpensive hardware to identify collective transfers and guarantee that loads and stores arrive in memory address order to the memory controller. CMS actively takes charge of collective transfers and pushes or pulls data to or from the on-chip processors according to memory address order. CMS reduces application execution time by up to 55% (20% average) compared to a state-of-the-art architecture where each processor reads and writes its data independently. CMS also reduces DRAM read power by 2.2× and write power by 50%.

1. INTRODUCTION

In recent years, the primary constraint for microprocessors has shifted from chip area to power consumption, leading to the stall in clock frequencies and the move towards massive parallelism [38, 5]. Emerging data intensive applications for these multicore platforms are projected to have enormous demands on memory bandwidth. However, DRAM bandwidth is not projected to scale proportionally to meet these future demands [43, 41, 33, 10, 17, 2, 30]. Memory bandwidth is already critical in numerous applications such as media applications, which have already been reported to require up to 300GB/s of memory bandwidth to utilize just 48 processors [31]. In addition, processing demands force the Xbox 360 to have 22.4GB/s of GDDR3 bandwidth to satisfy just three processors [43]. The situation will only become worse in the future because computational throughput is projected to improve much faster than memory bandwidth [33]. In fact, projections state that chip pins increase by 10% every year whereas processors double every 18 months [33]. In addition, while memory density nearly doubles every two years, the improvement in cycle time has been hundreds of times less, leading to tens to hundreds of processor cycles per memory access [41]. Energy is also a major concern because given that today's DDR3 technology consumes about 70pJ per bit, a system with only 0.2 bytes per FLOP memory bandwidth requires over 160mW of DRAM power [38]. This problem is already crucial in datacenters, where 25%–40% of total power is attributed to DRAM [42]. Maximizing DRAM performance and energy efficiency is critical, especially for future systems where DRAM's contribution will likely be proportionally larger than today [5].

Numerous crucial applications depend on parallel speedups achieved through bulk-synchronous single program multiple data (SPMD) execution where all compute elements are employed in tandem to speed up a single kernel. In fact, data-parallel applications have been cited as the biggest drivers for multi-core, and applications in future multi-cores are expected to follow data-parallelism even for consumer and mobile applications [7]. Bulk-synchronous kernels typically rely on domain decomposition to expose data parallelism. However, copying data from a contiguous representation in DRAM to the domain-decomposed (tiled) layout in on-chip caches poses significant challenges to modern memories. That is because current chip multiprocessors (CMPs) presume each core operates independently, even for SPMD execution. The result is that the memory is presented with uncoordinated and stochastic requests that exhibit poor locality, which degrades performance and power [42, 45, 32, 31]. Even though a plethora of memory controllers have been proposed, they are typically passive elements which do not control how requests arrive to them. Therefore, their degree of freedom is limited to the entries in their finite-size transaction queues [32, 18, 40, 29, 11].

In this paper, we demonstrate a combined software and hardware approach for coordinating DRAM and on-chip data movement for data-parallel applications named collective memory scheduling (CMS). CMS uses the software layer to identify collective transfer opportunities and collect the data layout information. It then uses that information in the hardware to read and write data arrays in the DRAM in strict memory address order. On the on-chip network side, CMS pushes or pulls data to or from processors according to how the data array should be distributed to the on-chip L1 caches or local stores. Memory access and data distribution are handled by a CMS hardware engine co-located with each memory controller, instead of individual processor prefetch or direct memory access (DMA) engines. Essentially, the CMS engine acts as a memory access accelerator and is inexpensive enough to include even in general-purpose systems. CMS shifts the pattern of data movement from concurrent actors (threads or processes in shared mem-

ICS'14, June 10–13 2014, Munich, Germany.

http://dx.doi.org/10.1145/2597652.2597654.

This work was supported by the Director, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

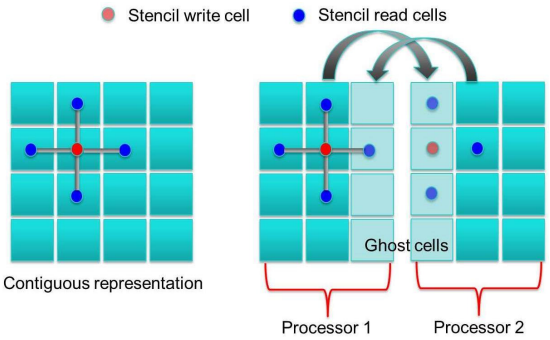

Figure 1: Tiling of a dense array onto a processor grid. Each tile is assigned to a processor. Tiles may include read-only data that replicate neighboring data (shared data). Such an example created by a 2D 5-point stencil is shown.

```
Array = hta(name,
      {[1,3,5], // Tile boundaries before
                //  rows 1 (start), 3 and 5
       [1,3,5]},// Likewise for columns
       [3,3]);  // Map to a 3x3 processor array
```

Figure 2: Example HTA declaration code.

ory) each requesting data from memory directly and independently, to collaborating actors accessing memory as a group by performing collective operations. CMS guarantees in-order memory access even with row- and column-major mappings which are more productive for programmers but normally produce unfavorable memory access patterns [14]. CMS also performs bandwidth filtering in algorithms where processors share data by reading such data just once from memory and then distributing it to all recipients. This can drastically reduce memory reads in data parallel numerical kernels such as matrix multiplication, stencils and FFTs where working sets are highly overlapping.

To make access to CMS easy for programmers, we augment the hierarchically tiled array (HTA) programming abstraction [15] to efficiently and compactly express collective transfers and pass on the required information to the CMS hardware engine. This way, CMS simplifies the application programming interface (API) since the same *collective* function call is performed by all processors, with no need to calculate individual DMA address ranges [36]. Although we use the HTA library to demonstrate the programming interface for CMS, the concept could easily be exploited by numerous other compiler-based programming systems, such as OpenMP and OpenACC.

We use algorithm kernels from the TORCH and PARSEC collections to demonstrate the effectiveness of CMS across a wide variety of applications [3, 19]. CMS supports arbitrary mappings of data to processors and data array dimensions, but we focus on data-parallel applications with regular tiled layouts such as dense and sparse 2D data arrays because of their broad coverage of typical use cases in consumer electronics and scientific/engineering applications [26, 28, 10, 17], and their projected criticality in future multi-cores [7].

By re-establishing a streaming access pattern, CMS reduces the completion time for a collective read and write operation by 39% for dense 2D data arrays, and 60% for sparse 2D data arrays. Consequently, CMS reduces application execution time by up to 55% (20% average), compared to independent DMA or prefetch operations in each processor. CMS also reduces DRAM read power by 2.2× and DRAM write power by 50%. In addition, CMS eliminates network congestion by replacing many independent read and write requests which saturate the network with a handful of control packets. CMS achieves this with a simple hardware engine and with no need for costly and deep transaction queues, which modern memory controllers use to partially recover a streaming access pattern [44, 45, 18, 32, 29].

2. BACKGROUND

2.1 Domain Decomposition

Domain decomposition is commonly used to expose parallelism for SPMD algorithms that operate on data much larger that the available on-chip storage. Figure 1 illustrates decomposition of a dense 2D data array into tiles. Tiles are typically sized to fit L1 caches or local storage in each processor [26]. Therefore, processors typically load their next tile, compute on the tile with minimal communication, and then write it back to memory. There is typically minimal data reuse across tiles assigned to the same processor. Accessing tiles in memory is done with local independent hardware prefetch [21] or cache fill streams for a cache-coherent CMP, a list of outstanding load-store requests for a massively multithreaded architecture like a graphical processor unit (GPU), or via a sequence of DMA requests for a local store architecture like STI Cell [36]. Other algorithms consist of sparse data arrays, where some tiles contain a large number of or exclusively zeroes. Applications with dense process grids leave processors with tiles containing only zeroes idle, whereas applications with sparse process grids do not reserve more processors than non-zero tiles. Data-parallel algorithms constitute critical applications in such diverse areas as image processing, seismic imaging, machine learning, electromagnetics, fluid dynamics, climate modeling, and others [26, 28, 10, 17].

2.2 Memory Access Streams and Efficiency

In current architectures and even in SPMD execution, processors access memory independently causing requests to arrive in nearly random order to memory [45, 42]. Random access patterns make prefetching difficult and degrade DRAM performance and power [44, 32, 42] because they cannot take advantage of pre-activated rows and therefore cause more row activations compared to sequential access patterns [42]. As a result, in many workloads an open row is used only once or twice before being closed due to a conflict [42]. Depending on the access pattern, only 14%–97% of peak memory bandwidth can actually be utilized [31].

Overfetch penalizes both latency and power because opening a new row requires charging bit lines, amplification by sense amplifiers, and then writing bits back to the cells. This further aggravates the memory bandwidth bottleneck in modern architectures, causing a wide variety of applications to be constrained by memory bandwidth [10, 43, 41]. Uncoordinated memory requests can also result in redundant memory accesses, because data shared between processors is fetched independently by each processor using transactions potentially separated by long time intervals. Finally, multiple independent requests congest the network waiting for vacancies in the memory controller's queue.

2.3 Hierarchical Tiled Arrays Representation

HTAs are a polyhedral representation language that compactly expresses tiled data arrays [15]. The declaration of

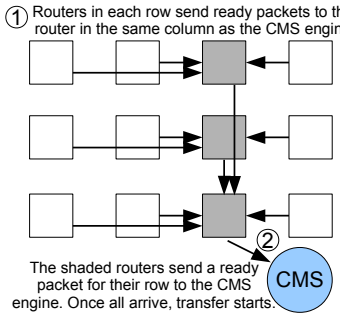

Figure 3: Initiating a synchronous read CMS operation.

HTAs includes how the data array is tiled and how tiles map (are distributed) to processors. The example of Figure 2 divides a 6×6 array into 2×2 tiles and maps those tiles to a 3×3 array of processors, as shown in Figure 8.

3. COLLECTIVE MEMORY TRANSFERS

This Section describes collective read and write operations, different data array layouts, the CMS hardware engine, and finally the programming interface to CMS.

3.1 Collective Read Operations

In read operations, the CMS engine reads memory sequentially and distributes data to the appropriate processors according to the data array's mapping to the processors. Essentially, the CMS engine takes charge of the collective operation and *pushes* data to the processors. Data that is shared between processors, such as ghost zones in stencil computations in the example of Figure 1, is read only once from memory and then distributed to all recipients with multicast or duplicate unicast packets.

The CMS engine initiates the transfer when all processors make the collective read function call for the same data array. This inserts an implicit barrier which can replace existing barrier calls, which are typical in computation loops of data-parallel computations [17]. To coordinate operation start, we employ a simple hierarchical communication pattern, shown in Figure 3. For cases where inserting a barrier would be inefficient, we also implement non-barrier reads where the transfer initiates when only the first processor is ready. This requires non-ready processors to store the next iteration's tile in addition to the current iteration's tile (the tile under computation), and possibly the previous iteration's tile which may be still in the process of being written back to memory. We do not allow processors to be further out of synchronization, to reduce local storage requirements.

3.2 Collective Write Operations

To easily guarantee memory access order, CMS write operations are performed as reads from the standpoint of the CMS engine. In other words, the CMS engine *pulls* data from the processors and writes it to memory. When the processor that holds the first tile line of the data array in memory address order (e.g., processor 1 in the top left of Figure 8) is ready to write its tile, it sends a write ready packet to the CMS engine. The CMS engine then sends read requests to retrieve the whole data array in memory address order. In the mapping of Figure 8, the first read request for elements $(1,1)$ and $(2,1)$ is served by processor 1, $(3,1)$ and $(4,1)$ by processor 2, $(5,1)$ and $(6,1)$ by processor 3, $(1,2)$ and $(2,2)$ by processor 1, and so on. Processors may delay their response until they produce the data.

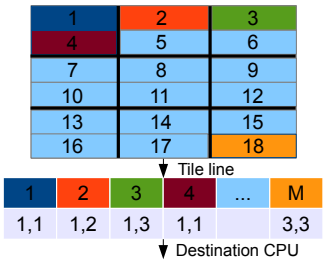

Figure 4: An example data layout array and the resulting mapping of tile lines to a 3×3 processor array.

In order to cover the communication delay and keep the memory constantly busy, there need to be more than one outstanding read requests in flight. Because the network guarantees no ordering, the CMS engine uses a small reorder buffer to enqueue read replies and write data to memory in address order. The size of the reorder buffer represents a tradeoff between eliminating memory idle cycles versus cost and network load. In our 8×8 2D mesh and using dense 2D data arrays, the minimum size which guarantees that the CMS engine constantly has data to write to the memory is six transactions for arrays of 512×512 elements, four transactions for 1024×1024, and three for 2048×2048 arrays.

3.3 Data Array Layout

CMS supports regular and flexible mappings of multi-dimensional data arrays in memory to processors. Regular mappings are for tiled arrays and map each tile to a processor according to a function or a mapping matrix. For example, the dense 2D data array of Figure 8 follows the function $F(x) = x$ because tile i maps to processor i. Regular mappings can also express sparse data arrays. For instance, in the same 3×3 processor array, a 3×3 mapping matrix with tile ID 1 in position $(1,1)$ (indicating that processor $(1,1)$ is assigned tile 1), 2 in position $(2,2)$ and 3 in position $(3,3)$ defines a sparse data array with tiles only in the diagonal. By knowing how the data array is mapped to memory addresses (such as row-major), the CMS engine calculates the address ranges of each tile. Regular mappings also have a parameter to define the shared data in each tile. In Figure 8, this parameter has a value of one.

Flexible mappings support any layout of memory addresses to processors. To enable this, the CMS engine maintains a small dedicated data layout array recording which processor owns each data. An example array is shown in Figure 4. To reduce its size, this array is indexed at a granularity defined by the largest contiguous address range that belongs to a single processor. With row- or column-major mapping and tiles, that is a tile line. The CMS engine may also include an identical additional data layout array to record processors that share each tile line.

The size of the data layout and sharer arrays depends on the size of the data array and the indexing granularity. To support a 2048×2048 2D data array in a 64-processor system with 256B tile lines, the data layout array is 20KB, which is less than 0.1% of the cache size in the Intel Xeon Phi, and similarly a very small fraction of other on-chip caches or local storage. Data arrays with finer indexing granularities may need more storage. This can be mitigated by mapping the data array to memory addresses such as to increase the

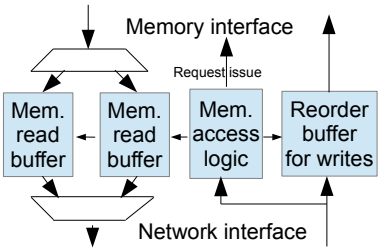

Figure 5: CMS engine outline.

indexing granularity. Instead of adding dedicated storage, the CMS engine can also use part of the L3 cache local to the memory controller the CMS engine is co-located with. If space in the data layout array does not suffice, part of the collective transfer is performed without CMS.

The data layout array is populated by the CMS engine after observing the memory access pattern created when each processor fetches its first tile without coordination. We rely on processors to explicitly mark what read requests are for shared data (data in another processor's tile). The CMS engine also discovers the data layout array indexing granularity during that time by observing the size of read requests. After the initial learning period, all subsequent read and write transfers are performed collectively until the application initiates a new learning period, indicating that it is now accessing a data array with a different layout.

We choose to have the CMS engine discover the layouts of data arrays instead of transferring the data layout information from the processors to the CMS engine in the request packets due to the on-chip data movement that would create. The performance impact from performing the first transfer without coordination is negligible due to the large number of computation iterations typical in data-parallel applications. For most applications, one owner and one reader (sharer) processors suffice. However, if there are more readers than sharer data arrays, the additional readers do not participate in the collective transfers for their read-only (shared) data.

Ready (request) packets contain the starting address of the data array and information about how it maps to processors. For data arrays with regular mappings, ready packets contain the mapping function or matrix as previously discussed, which can easily be contained in a few flits. For flexible mapping arrays, ready packets need only contain an indication of whether a new learning period should initiate.

3.4 Collective Memory Scheduling Engine

We implement the CMS engine atop a typical DMA engine. As illustrated in Figure 5, the CMS engine has a memory interface side and an on-chip network interface side. Read or write requests arrive from the network interface side. At the memory interface side, the CMS engine either sends read requests as fast as the memory controller allows, or it sends write requests whenever it has valid data to write. Once the collective transfer is complete, the CMS engine returns to idle, waiting to accept a new operation.

The memory access logic stores the collective transfer information. In regular mapping mode, the CMS engine stores that information from the collective request packet. Otherwise, the CMS engine uses the logic outlined in Figure 6.

The allowed number of pending memory transactions depends on the size of the CMS engine's buffers. Read operations use two small buffers ("mem. read buffers") for the outstanding DRAM read requests and to permit duplicating

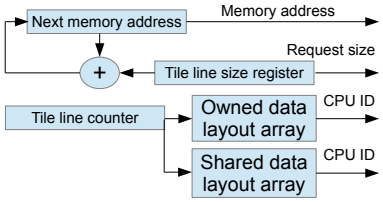

Figure 6: An outline of the memory access logic for data arrays with flexible mapping. Since accessing a tile line in the DRAM requires many cycles, these circuits are pipelined to avoid extending the CMS engine critical path.

```
Array = hta(name, {[1,3,5],  [1,3,5]}, [3,3],
      F(x) = x, // Mapping function
      1); // Tiles share one cell from their edges
```

Figure 7: An example declaration for a regular mapping. Each tile shares cells adjacent to their boundaries (therefore two rows and two columns in total).

shared data packets. No new DRAM read requests may be issued until one of the two buffers is free. The reorder buffer for write operations tags requests to tiles for their tile lines and uses that tag to write the returned data into the correct location in the reorder buffer such that memory address order is preserved. The CMS engine can also include a small queue to store pending collective transfer requests, but only one is active at a time.

We co-locate a CMS engine with each memory controller in order to reduce communication distance. The CMS engine can be integrated into the memory controller instead of remaining a separate entity. In memory controllers with multiple channels, we can extend the CMS engine to perform different parts of the collective transfer in different channels. Alternatively, we can use one CMS engine per channel to perform multiple concurrent collective transfers. With multiple memory controllers, a large collective transfer is divided into smaller ones, each of which is assigned to a CMS engine. Therefore, a chip-wide operation will activate all CMS engines, each performing a portion of the operation.

Because the CMS engine guarantees memory address order, the memory controller need not be more complex than a FIFO scheduler with just enough transaction queue entries for memory pipelining. Moreover, since the CMS engine performs the entire operation instead of individual-processor DMA engines or prefetch units, these is opportunity to simplify memory controllers and local processor DMA engines or prefetch units. Doing so would outweigh the additional complexity of the CMS engine compared to a typical DMA engine because of the complexity of modern memory controllers with large transaction queues and complex scheduling policies [32, 18, 40, 29, 11]. Even without these cost reductions, as we show in Section 4.2.4, CMS engines are inexpensive enough to be included in general-purpose systems at a minimal cost increase. When there are no collective transfer opportunities, CMS engines remain inactive and can power down, similar to any other accelerator.

3.5 Programming Interface

CMS uses the software to identify transfers as collective, transfer the necessary data layout information to the CMS hardware engine, and make the collective transfer capabilities easily accessible to the programmer. Although a variety of APIs and libraries can take advantage of this kind of collective memory operations (such as OpenMP and OpenACC), we choose the HTA syntax [15] as an interface to

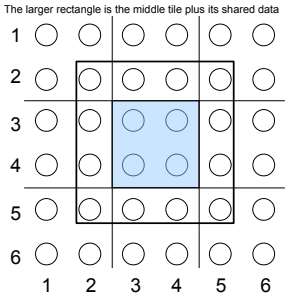

Figure 8: The mapping from our example declaration. Only the shared zones for the shaded (middle) tile are shown. Vertical and horizontal lines are tile boundaries.

```
Loading a HTA with a CMS read
HTA_instance = CMS_read (HTA_instance);

Loading the same HTA with DMA operations for each line of data
Array[row1] = DMA (Starting_address_row1,
                   Ending_address_row1);
                   .
                   .
Array[rowN] = DMA (Starting_address_rowN,
                   Ending_address_rowN);
```

Figure 9: Without CMS, the programmer needs to calculate starting and ending addresses for each tile line in a local-store architecture, including shared data.

formally express tiled data arrays. For regular mappings, CMS adds a parameter to HTAs to define the regular mapping, and another parameter for the number of cells shared between neighboring processors to accommodate computations such as stencils, as discussed in Section 3.3. Figure 7 shows an example. The resulting mapping is illustrated in Figure 8. HTAs have been extended to offer an alternative and more complex but also more powerful syntax to declare shared zones of arbitrary shapes and sizes [15], which we can also use for CMS. For flexible mappings, the library can abstractly determine if a new learning period is required, and thus no added parameters are necessary. Flexible mappings that cannot be expressed with HTAs require library modifications or alternative polyhedral representations.

We provide access to CMS functionality using a library that exposes an API similar to DMA function calls [36]. This leaves the programming style intact and simply requires the programmer to use CMS function calls instead of DMA function calls. A CMS read or write function call requires only the HTA instance as parameters. Since the HTA instance contains all tiling, layout, and addressing information, the library abstractly constructs request packets with the information required by the CMS engine. Therefore, all processors that wish to read or write the same HTA *make the exact same function call.*

The CMS API is considerably simpler than DMA operations in local-store architectures such as STI Cell [36]. In the common case that a processor's tile consists of non-contiguous memory addresses [16, 30], a potentially large number of DMA calls is required, which in turn require deep transaction queues in each DMA engine [22]. As an example, to transfer a tile in a 64-core system from a regularly-mapped 2048×2048 data array without architectural support for strided memory access, each processor requires 256 separate DMA transfers. That is because each processor's tile contains 256 rows each of which are disjoint in memory address order with row- or column-major mappings, which are typically used. The equivalent CMS operation requires only one function call, as shown in Figure 9.

Although our example programming interface is limited to local-store architectures with an explicit API, this is not a requirement. Conventional directives-based approaches such as OpenACC, and Polyhedral representations alternative to HTAs can also use CMS hardware [23, 4]. Alternatively, compilers or run-time systems can analyze memory access patterns and data structure layouts to identify collective transfers. Finally, collective transfers can be performed even using basic language constructs, but this requires the programmer to directly communicate with the CMS engine.

4. EVALUATION

4.1 Methodology

We use DRAMSim2 [34] to model DRAM performance in response to the access patterns generated by our test-cases. We then input these resulting DRAM performance and power models to a heavily-modified version of the Booksim network simulator that includes processors and local stores. This architecture is representative of future many-core chips with local stores, which we use as a proof of concept and later discuss the applicability of CMS to other platforms. All our simulations use dense process grids (all processors participate regardless of the amount of data assigned to them). Initially we simulate writes and barrier read operations of 2D dense and sparse regularly mapped data arrays with no shared data. Variables (cells) in data arrays are 8-byte double precision.

We then present application results for the following important applications: Fluidanimate and streamcluster from the PARSEC benchmark suite [3], seismic wave propagation simulation (RTM) [26], the SOBEL filter used extensively for image processing (SOBEL) [12], LU factorization of a dense matrix (LU), sparse matrix multiply (SpMV), and conjugate gradient (CG) from the TORCH benchmark suite [19], and the Laplacian stencil from [20]. All applications use a regular mapping of tiles to processors, therefore there is no data layout array in the CMS engine. Such regular mapped data arrays comprise the building blocks of applications ranging from image processing in consumer devices to the largest scale high performance computing (HPC) applications such as climate modeling and fluid simulations. SpMV and CG use sparse matrices with approximately 25% non-zeroes, predominantly located in tiles on the diagonal. Similar to past work, only non-zero values are stored in and transferred to and from main memory [6]. LU and streamcluster use dense tiled data arrays, while fluidanimate, RTM, SOBEL and Laplacian use dense tiled data arrays with stencils and therefore use shared data.

For our application results, we model a CMP with an 8×8 array of Intel Xeon Phi co-processors, which are simple in-order x86-based processors and representative of the simple cores projected for future many-core chips [5]. For each application, we calculate the processing time per array variable as well as the shared data size, and simulate ten iterations of each application's execution loop (more iterations produce comparable results). Because thread migration is detrimental to data-parallel applications, we use a static mapping of threads to processors. We assume enough local storage for triple buffering for all tile sizes, which prevents performance degradation due to redundant DRAM or higher-level cache

accesses as tile size increases. In general-purpose systems, the software can perform cache blocking to resize tiles to fit available local storage. We use the typically-used row-major mapping of data arrays to memory (column-major produces comparable results) [16, 30, 14].

Our memory subsystem consists of memory channels with independent controllers, which matches the configuration of many contemporary server processor designs [2, 33]. We place four memory controllers at the corners of our CMP, and a CMS engine co-located with each memory controller. We use static address-based mapping to map tile lines (memory addresses) to memory controllers. Therefore, each processor requests each tile line from the appropriate memory controller and CMS engine.

A 2D mesh on-chip network is used with dimension-order routing (DOR) and four-stage input-buffered routers [9]. Input buffers have 4 virtual channels (VCs), with eight flit slots statically assigned to each. Two VCs are used for request packets, and two VCs for replies. The datapath is 128 bits wide. Data-transferring packets carry one line of a processor's tile, plus one head flit.

For the memory, we simulate a 16MB DDR3 1600MHz memory module from Micron with a 64-bit data path and two ranks with 8 banks each. There is a single memory controller for the two ranks. The most significant bits of the address determine the channel, then the row, column, bank, and finally rank. The memory controller has 32-slot transaction and DRAM command reorder queues, and first ready first come first served (FRFCFS) scheduling [32, 45]. Our FRFCFS scheduler uses an open-row policy which respects row buffer locality by prioritizing transactions to open DRAM rows. This essentially performs limited transaction reordering by address, similar to other modern schedulers [32, 18, 29, 11]. We compare CMS against FRFCFS because FRFCFS maximizes memory throughput compared to a variety of other controllers [40, 32]. FRFCFS does not necessarily minimize application execution time because maximizing memory throughput may be unfair to threads [29]. However, we do not model and therefore hold these adversary effects against FRFCFS. Therefore, our FRFCFS represents an optimized state-of-the-art memory controller. We use 1600MHz for all processors and the network.

4.2 Results

4.2.1 Memory Throughput Degradation

First, we quantify the performance of DRAM in response to an uncoordinated access pattern that results from a SPMD algorithm running on a conventional many-core memory subsystem. In this case, our FRFCFS memory controller tries to maximize performance by reconstructing a linear access pattern and respecting row buffer locality using transaction reordering. However, even a sophisticated controller's reordering capability is inherently limited by the depth of its transaction queue since memory controllers do not control the order requests arrive into their queue. In contrast, CMS guarantees in-order memory access without complex reordering schemes or deep queues.

To set up this experiment, we use DRAMsim2 [34] to simulate a synthetic 16MB in-order trace of loads to represent the coordinated memory accesses created by CMS, and an out-of-order trace to simulate the uncoordinated case where loads or stores are presented to the memory controller in random order. A single load accesses a 64-Byte word, causing an eight-cycle burst due to the 64-bit memory controller datapath. The uncoordinated requests are randomly-ordered in sizes of 128 bytes representing one tile line. These traces therefore *accurately represent* the unpredictable memory access stream that would arrive to a memory controller in the baseline case where processors send requests for each tile line without coordination. Experiments with access traces larger than 16MB produce comparable results.

Our results show that for the uncoordinated access pattern (baseline with FRFCFS), DRAM throughput drops by 25% for loads and 41% for stores. Also, median latency increases by 23% for loads and 64% for stores, maximum latency increases by 2× in both cases, and power increases by 2.2× for loads and 50% for stores. Compared to the DRAM peak bandwidth, reads achieve 80% and writes 75% with CMS compared to 60% and 44% respectively for the uncoordinated case. Even streaming unit-stride traces cannot achieve 100% throughput due to refresh operations.

The uncoordinated case exhibits higher power consumption due to an increase in activate and precharge power (5.2× for loads and 3.4× for stores), caused by a 96% row buffer miss rate in the uncoordinated case compared to 3% with CMS. Past work has found similar results, and not even the best-performing memory transaction scheduler can bridge the gap between random and in-order accesses [32, 42, 44, 41]. For example, the row buffer hit rate drops from 60% for a single processor to 35% in a baseline 16-processor CMP, in a variety of benchmarks [42]. Also, in a 16-core CMP, a row fetched into the row buffer is typically used only once or twice before being closed due to a conflict [42].

4.2.2 Operation Completion Time

With the above DRAM results, we then perform collective (coordinated) and uncoordinated transfers of data arrays. The results are shown in Figure 10 (left). Compared to the uncoordinated case, CMS reduces completion time by an average of 39% for both reads and writes for dense data arrays, and 60% for sparse data arrays. These gains are predominantly due to the lower throughput the DRAM provides with random (uncoordinated) memory access patterns. The slightly lower performance gains for CMS writes for 128×128 and smaller data arrays are due to the propagation delay between processors generating request (ready) packets and accessing memory for the first time, which is 22 cycles for barrier reads and 40 for writes by average in our system. This is easily amortized by the duration of a transfer. Larger transfers make these delays negligible.

Larger data arrays increase the tile line size which favors spatial locality in the uncoordinated case. However, larger tile lines also create more severe contention and degrade on-chip network performance because of the long packets. Furthermore, in the uncoordinated case, sparse data arrays create intense hotspots because the majority of the data are destined to the few processors that are assigned the tiles along the diagonal. This provides the performance advantage for sparse arrays compared to dense arrays and illustrates that CMS load balances the network better because it accesses tile lines in processors in an interleaved manner. Otherwise, hotspots can be created depending on the order requests arrive to the memory controller.

We then repeat the experiments, but we add a uniform random (UR) background traffic pattern with a 10% flit in-

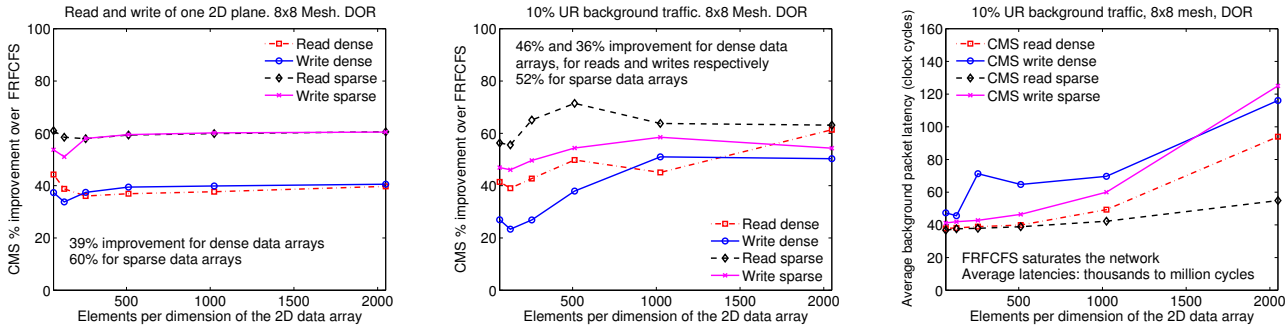

Figure 10: CMS transfer completion time improvement over FRFCFS (uncoordinated) and the impact on background traffic.

jection rate. A 10% injection rate provides non-negligible traffic, but not enough to saturate the network by itself. This traffic is composed of read and write requests and replies similar to DMA traffic, and represents innocent bystander traffic. As shown in Figure 10 (center), the reduction in execution time for CMS in the mesh is 46% for reads and 36% for writes for dense data arrays, and 52% for sparse arrays for both reads and writes. The reduction in speedup for collective writes compared to collective reads for dense arrays illustrates the increased traffic of collective writes compared to reads. While background traffic degrades performance for CMS, it is more adversary to the uncoordinated case (FRFCFS) because that creates hundreds or thousands of request packets that traverse the network most of which are queued in the network for a long time because they cannot be absorbed by the memory controller, thus contenting with the background traffic. Figure 10 (right) illustrates the impact to the background traffic.

Repeating these experiments in a 144-processor system yields comparable results. Finally, having only one or two memory controllers instead of four slightly favors CMS because the baseline case produces more severe network hotspots and stresses the memory more.

4.2.3 Impact on Application Execution Time

We show application results in Figure 11. The gains depend on the ratio of the time spent computing for each tile versus completing a read and then a write operation. This is the application's byte per FLOP ratio. CMS provides a minimal (0%–2%) reduction in execution time for the compute-bound applications in our system (RTM, LU and CG), but CMS still reduces DRAM power. In contrast, memory bandwidth-bound applications directly benefit from CMS in execution time. The geometric mean of the execution time reduction for all applications is 8.5% due to the compute-bound applications in our collection, the average is 20%, and the maximum is 55%.

Figure 11 (right) presents execution time as a function of the bytes per FLOP ratio, in order to estimate execution time for other architectures and applications. Dense data array applications take advantage of CMS beyond 0.6 bytes per FLOP, while applications with sparse arrays similar to our applications benefit beyond 0.5 bytes per FLOP.

Sparse data array applications have their compute time dictated by the tiles with the most non-zeroes, which in our applications are predominantly the tiles on the diagonal. Because even tiles on the diagonal contain zeroes, compute time per tile decreases. This decreases compute time which increases the pressure to memory, therefore allowing CMS to provide larger execution time benefits. Applications with fewer non-zeroes than our 25% would further benefit CMS for the same reason, assuming they are still memory-bandwidth bound. Dense data array applications are perfectly load-balanced. In addition, we ignore scheduling or other effects which can delay individual processors, because such effects are potentially adversarial to any parallel application. Moreover, our stencil-based applications have the additional benefit of eliminating redundant memory reads in read CMS operations, since shared data is read only once and submitted to the owner and reader processors, instead of each processor retrieving its shared data separately. For example, with a 256×256 dense data array and 5-point stencils, there are 12% fewer memory reads with CMS.

Table 1: RTL synthesis results.

	DMA	CMS
ASIC		
Combinational area (μm^2)	743	16231
Non-combinational area (μm^2)	419	61313
Minimum cycle time (ns)	0.6	0.75
FPGA		
LUTs for logic	245	856
Minimum cycle time (ns)	4.4	5.1

4.2.4 CMS Implementation and Synthesis

We implement a CMS engine and a typical DMA engine in RTL and synthesize them using Synopsys Design Compiler and a 40nm general-purpose technology library. We synthesize the same designs using the Xilinx FPGA design flow for a Virtex-5 FPGA. The CMS and DMA engines are configured for the DDR3 Micron modules with 64 bit datapaths used in our evaluations. For the CMS engine, the two read buffers are 16×128 bits each. The reorder buffer for write operations is sized to hold eight transactions of 16×128 bits each, for a total of 2KB. Eight transactions are enough to keep the memory busy in the write operations of our evaluations, as discussed in Section 3.2. In the ASIC flow, the reorder buffer as well as the small read buffer in the CMS engine are implemented using flip-flop (FF) arrays. The CMS engine does not include a data layout array because the size as well as whether the data layout array is included in a L3 cache are architecture-dependent. Table 1 shows the results.

As shown, cycle time for the CMS engine increases by 25% in the ASIC flow and 16% in the FPGA flow. This is due to the extra complexity of the CMS engine, the write reorder buffer, and the two read buffers. Also due to the buffers, the CMS engine occupies more area. However, to make the CMS engine operate at the same clock frequency as the DMA engine, we can simply pipeline the CMS engine

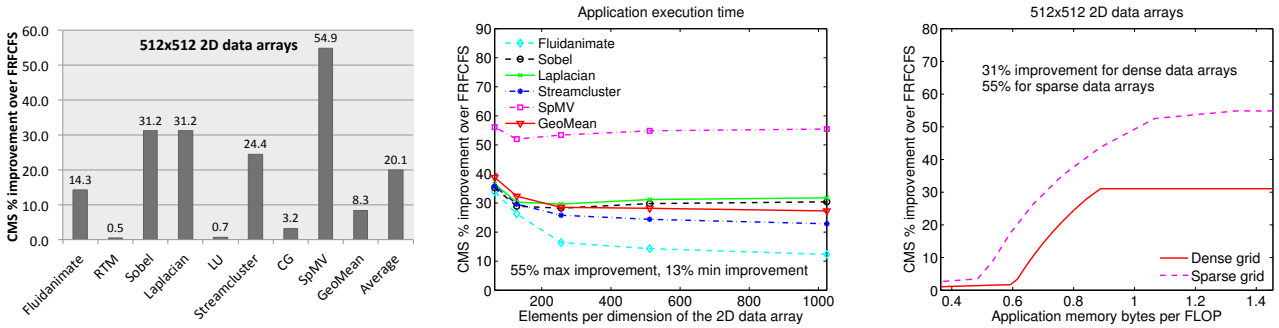

Figure 11: Application execution time improvement of CMS compared to FRFCFS (uncoordinated).

by adding one more stage. The one extra cycle delay is negligible compared to the duration of a collective transfer. The data layout array and handling logic, shown in Figure 6, may further increase area. However, they will not increase cycle time because array access and the handling logic can be heavily pipelined since one memory access covers multiple CMS engine cycles due to the DRAM's burst length.

CMS can simplify other system components in systems that predominantly perform collective transfers. That is because CMS requires only a simple FIFO memory scheduler with just enough transaction queue entries for memory pipelining. Compared to modern memory controllers, this is a significant reduction in cycle time because modern controllers typically hold a few tens of transactions [32] and perform an associative comparison of all requests in their transaction queue every cycle (therefore requiring comparators for every queue entry). They then issue a transaction from any position in the queue based on multi-level priority and other complex schemes [45, 32]. CMS also does not use DMA or prefetch engines in *each* processor allowing for simpler designs. Even without these cost reductions, CMS engines are inexpensive enough to be included in general purpose systems (a CMS engine is required per memory controller instead of per processor). When those systems do not perform collective transfers, CMS engines remain inactive, similar to any other accelerator.

5. DISCUSSION

CMS is targeted at bulk-synchronous SPMD execution models that transfer data arrays to and from memory, and is not intended to address irregular multi-processing workloads. We believe that the kinds of algorithms that are the largest drivers for improved computational performance and multi-core are in fact SPMD kernels such as data-parallel kernels that are seen in image processing, face recognition, machine learning, fluid dynamics, linear algebra, kinetics simulation, and numerous other applications in platforms from HPC to embedded architectures [26, 28, 10, 17]. Many future applications in multi-cores are expected to follow data-parallelism even for consumer and mobile applications [7]

The programming interface to CMS operates in the virtual address space but the CMS hardware engine uses physical addresses. This, however, does not require modifications to existing virtual to physical memory address translation mechanisms such as the TLB in each processor. Even though data placed contiguously in the virtual address space may not be contiguous in the physical address space, CMS guarantees that whether the DRAM page (row) accessed next is the next contiguous page in the physical address space or not will not affect performance and power so long as every line within that page is used (to the extent made possible by the application) after that page is open. Even if the TLB relocates the page, it will not affect the CMS engine's ability to make maximal use of the data within that open page.

Idle processors can be programmed to mimic the functionality of the CMS engine to avoid dedicated CMS engines. However, this makes some processors unavailable to the application, performs collective transfer at a much higher energy cost, and requires processors with an unrealistically high number of outstanding memory requests to cover the bandwidth–delay product to memory.

Highly threaded SMs within GPUs are similarly challenged by data-parallel applications and the desire to coalesce independent thread accesses into a limited number of accesses to memory [1]. CMS can be used to replace or augment the existing functionality within an SM when data is loaded either into shared memory or the register files. CMS can also be used to view the union of transfers performed by the active set of SMs as a collective and move data to and from their respective load stores. GPUs have programming constructs capable of expressing collective transfers, such as HTAs which have been implemented for OpenCL [47].

In applications that allow a subset of processors to make progress faster than others, CMS would force the fast processors to stall due to the implicit barriers. This typically does not degrade execution time because application performance is commonly dictated by the slowest processors, as is the case with our sparse data array application benchmarks. However, barrier calls are already typical in data-parallel applications that are the focus of CMS [17].

While in local-store architectures such as STI Cell [36] we choose to identify collective transfers by using a software API that replaces DMA function calls, typical cache-coherent CMPs can use hardware prefetch units. In such systems, individual prefetch units in each processor can transmit their predictions to the CMS engine, which can identify collective prefetching opportunities. This would require modifications to the cache coherency protocol to allow L1 caches to receive data they did not request and to identify data to write back to the memory before it is evicted. Prefetch decisions can also be performed by the CMS engine by observing the memory access stream, without prefetch engines at each processor. Alternatively, compilers can also recognize collective transfer opportunities abstractly from the programmer and transfer the same data layout information that the programmer provides through HTAs in our current implementation to the CMS hardware engine.

6. RELATED WORK

Past work has researched collective data transfer techniques in very different contexts. In wide-area TCP/IP networks, coordinating the nodes to send their data to a common destination with a common transfer schedule that avoids conflicts substantially reduces network congestion [8]. Alternative techniques for wide-area networks focus on heterogeneity and use of shared resources by transferring different chunks of the same file from replicas [24]. Collective data transfers have also been applied to server disk-directed I/O, because the access bandwidth for magnetic hard disk drives significantly improves with sequential accesses [37].

Vector machines such as the Cray-1 [35] overcome the inefficiencies of DRAM *overfetch* and access granularity by using massive bank-switching to offer word-granularity accesses. However, vector core designs and memory controllers are costly due to their limited market and sizable engineering costs [13]. VIRAM can also exploit data-level parallelism to overcome the wiring costs of massive bank-switching [25], but the memory capacities offered by the various processor-in-memory approaches are impractically small for the commercial market. Moreover, variations of DMA engines [22] still perform transfers between only two components, thus creating out of order access streams to memory.

The Impulse memory controller reorganizes the memory address stream so that non-contiguous address patterns appear contiguously in the cache hierarchy [46]. However, with Impulse the data arrays remain scattered in the DRAM, thereby leading to inefficient DRAM performance. Moreover, even though data-to-memory address layouts alternative to row- or column-major can produce a more favorable memory access stream for some applications without CMS [16, 30, 14], complex layouts are counter-productive for programming. Such data layout transformations still cannot outperform CMS because CMS guarantees in-order memory access. CMS also makes communication-avoiding optimizations unnecessary, which removes the need for extra local storage or redundant computation which is typical from such optimizations [17, 10].

Sophisticated memory schedulers use complex scheduling policies and can be thread-aware [32, 18, 40, 29, 11]. Many schedulers also perform limited reordering by attempting to exploit row buffer locality and bank parallelism among other metrics [29]. Still, even a memory controller with an ideal policy is inherently incapable of fully reconstructing the memory access stream. That is because controllers are *passive* elements which do not control the order requests arrive to them and decide which one to serve next only from within their transaction queues. In contrast, CMS guarantees in-order memory access by actively controlling the transfer and *pushing* or *pulling* data to and from processor local stores or L1 caches. CMS has similar goals with "memory access scheduling" proposed for stream processors, but memory access scheduling is merely an algorithm that applies to the memory controller, and thus is inherently limited by the size of the memory controller's transaction queue [31]. As we explain in Section 4.1, we compare against FRFCFS with an open-row policy because FRFCFS maximizes throughput compared to many other controllers.

Past work has simplified memory controllers by using the on-chip routers to reorder requests [45]. However, because decisions are made with local knowledge and processors still issue requests independently, this scheme performs slightly lower than a FRFCFS scheduler. Alternative work uses admission control to inject only requests for open DRAM rows [27]. However, this uses a centralized scheme and thus faces limited scalability, and also risks idling memory due to propagation delay. Frequently-accessed data can be placed in the same row to favor open row DRAM policies [41]. Modifications to DRAM internals have also been proposed to reduce the negative power effects of random-order sequences, by avoiding to activate all the bitlines in a row before the exact read request is known [42].

While local and last-level cache (LLC) caches can reduce DRAM accesses during the computation phase of a loop, data is still retrieved from main memory when loading new and storing old data arrays. CMS focuses on fetching new and storing old data, and not on cache interference across tiles during computation. In the wide variety of memory bandwidth-bound applications discussed in this paper, retrieving new and storing old data suffice to saturate memory bandwidth. LLCs may potentially assist in reducing redundant memory reads, which however is not the dominant factor for CMS. LLCs can also partially reconstruct address order for writes with a write back policy, especially with mechanisms such as the virtual write cache which relies on the memory controller having idle cycles to fetch data from the LLC in address order [39]. However, in memory-bandwidth bound applications memory controllers are hardly idle. Also, streaming (write-through) writes are preferable to write back policies in data-parallel computations to prevent polluting higher-level caches because the results of a computation loop are not reused in the next iteration [10, 17].

Prefetching is currently the defacto solution for latency hiding on modern CPU architectures. However, prefetching typically focuses on reducing latency and offers little benefit in systems that are bound by memory bandwidth. In general there are two forms — cache prefetchers (move data from cache to processors) and DRAM prefetchers (move data from DRAM to the LLC). The performance of cache prefetchers often suffers on bulk synchronous applications that cache block dense arrays because the contiguous address stream length is often short. As such, prefetcher latency is not amortized and significant overfetch can occur (a processor inappropriately prefetches data from the next cache block), stressing memory bandwidth. Cache prefetchers also typically perform predictions independently at each processor and thus create out-of-order access patterns [21]. In addition, the performance of a DRAM prefetcher can be particularly sensitive to the number of active threads, how separated their address streams are, and the number of streams the prefetcher can track. Finally, prefetching lacks knowledge from the applications and is thus prone to errors where the wrong data is fetched instead of the data the processor actually requires. This is detrimental to both the memory because it served erroneous requests, but also to the processor. CMS addresses the challenges of both cache and DRAM prefetchers by distilling the core collective memory access pattern and avoiding mispredictions.

7. CONCLUSION

To make optimal use of the limited memory bandwidth of current and future systems, we present CMS that provides a shared responsibility mechanism between the software and an inexpensive hardware engine to coordinate parallel data accesses in a CMP such that data arrays are read from or

written to the DRAM in strict memory address order and distributed to or collected from processors. CMS is essentially a memory access accelerator that is inexpensive to include even in general-purpose systems. CMS actively takes control of collective data transfers by *pushing* or *pulling* data to or from on-chip L1 caches or local stores. CMS maximizes memory throughput beyond that possible even by the most aggressive transaction schedulers in modern memory controllers, reduces memory power and latency, simplifies the API to manage bulk-synchronous DMA operations, and alleviates network congestion. CMS reduces application time by up to 55% (20% average), memory read power by up to 2.2×, and memory write power by up to 50%.

8. REFERENCES

[1] A. Abdelfattah *et al.*, "Optimizing memory-bound numerical kernels on GPU hardware accelerators," ser. VECPAR, Kobe, Japan, 2012.

[2] D. Abts *et al.*, "Achieving predictable performance through better memory controller placement in many-core CMPs," ser. ISCA, 2009.

[3] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[4] U. Bondhugula *et al.*, "A practical automatic polyhedral parallelizer and locality optimizer," ser. PLDI, 2008.

[5] S. Borkar and A. A. Chien, "The future of microprocessors," *Communications of the ACM*, vol. 54, no. 5, 2011.

[6] A. Buluc *et al.*, "Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication," ser. IPDPS, 2011.

[7] Y.-K. Chen *et al.*, "Convergence of recognition, mining, and synthesis workloads and its implications," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 790–807, 2008.

[8] W. C. Cheng *et al.*, "A coordinated data collection approach: design, evaluation, and comparison," *IEEE Journal on Selected Areas in Communications*, vol. 22, no. 10, 2006.

[9] W. J. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., 2003.

[10] K. Datta *et al.*, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," ser. SC, 2008.

[11] E. Ebrahimi *et al.*, "Parallel application memory scheduling," ser. MICRO, 2011.

[12] W. Gao *et al.*, "An improved sobel edge detection," ser. ICCSIT, vol. 5, 2010.

[13] J. Gebis *et al.*, "Improving memory subsystem performance using ViVA: Virtual Vector Architecture," ser. ARCS. Berlin, Heidelberg: Springer-Verlag, 2009.

[14] C. Gou, G. Kuzmanov, and G. N. Gaydadjiev, "SAMS multi-layout memory: Providing multiple views of data to boost SIMD performance," ser. ICS, 2010.

[15] J. Guo *et al.*, "Writing productive stencil codes with overlapped tiling," *Journal on Concurrency and Computation: Practice and Experience*, vol. 21, no. 1, 2009.

[16] T. Henretty *et al.*, "Data layout transformation for stencil computations on short-vector SIMD architectures," ser. CC/ETAPS, 2011.

[17] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on GPU architectures," ser. ICS, 2012.

[18] Y. Ishii, M. Inaba, and K. Hiraki, "Unified memory optimizing architecture: memory subsystem control with a unified predictor," ser. ICS, 2012.

[19] A. Kaiser *et al.*, "TORCH computational reference kernels: A testbed for computer science research," Tech. Rep. UCB/EECS-2010-144, Dec 2010.

[20] S. Kamil *et al.*, "An auto-tuning framework for parallel multicore stencil computations," ser. IPDPS, 2010.

[21] M. Kandemir, Y. Zhang, and O. Ozturk, "Adaptive prefetching for shared cache based chip multiprocessors," ser. DATE '09, 2009.

[22] H. Kavianipour and C. Bohm, "High performance FPGA-based scatter/gather DMA interface for PCIe," ser. NSS/MIC '12, 2012.

[23] K. Keahey, P. Fasel, and S. Mniszewski, "PAWS: collective interactions and data transfers," ser. HPDC, 2001.

[24] G. Khanna *et al.*, "A dynamic scheduling approach for coordinated wide-area data transfers using GridFTP," ser. IPDPS, 2008.

[25] C. Kozyrakis and D. Patterson, "Scalable, vector processors for embedded systems," *IEEE Micro*, vol. 23, no. 6, 2003.

[26] J. Krueger *et al.*, "Hardware/software co-design for energy-efficient seismic modeling," ser. SC, 2011.

[27] D. Lee, S. Yoo, and K. Choi, "Entry control in network-on-chip for memory power reduction," ser. ISLPED, 2008.

[28] M. Mohiyuddin *et al.*, "A design methodology for domain-optimized power-efficient supercomputing," ser. SC, 2009.

[29] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems," ser. ISCA, 2008.

[30] L. Peng *et al.*, "High-order stencil computations on multicore clusters," ser. IPDPS, 2009.

[31] S. Rixner, "A bandwidth-efficient architecture for a streaming media processor," Ph.D. dissertation, Massachusetts Institute of Technology, 2001.

[32] S. Rixner *et al.*, "Memory access scheduling," ser. ISCA, 2000.

[33] B. M. Rogers *et al.*, "Scaling the bandwidth wall: challenges in and avenues for CMP scaling," ser. ISCA, 2009.

[34] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A cycle accurate memory system simulator," *IEEE Computer Architecture Letters*, vol. 10, no. 1, 2011.

[35] R. M. Russell, "The CRAY-1 computer system," *Communications of the ACM*, vol. 21, no. 1, 1978.

[36] S. Schneider, J.-S. Yeom, and D. S. Nikolopoulos, "Programming multiprocessors with explicitly managed memory hierarchies," *IEEE Computer*, vol. 42, no. 12, 2009.

[37] K. Seamons *et al.*, "Server-directed collective I/O in Panda," ser. SC, 1995.

[38] J. Shalf, S. S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," ser. VECPAR, 2010.

[39] J. Stuecheli *et al.*, "The virtual write queue: Coordinating DRAM and last-level cache policies," ser. ISCA, 2010.

[40] L. Subramanian *et al.*, "MISE: Providing performance predictability and improving fairness in shared main memory systems," ser. HPCA, 2013.

[41] K. Sudan *et al.*, "Micro-pages: increasing DRAM efficiency with locality-aware data placement," ser. ASPLOS, 2010.

[42] A. N. Udipi *et al.*, "Rethinking DRAM design and organization for energy-constrained multi-cores," ser. ISCA, 2010.

[43] A. Vega *et al.*, "Breaking the bandwidth wall in chip multiprocessors," ser. SAMOS, 2011.

[44] D. T. Wang, "Memory DRAM memory systems: performance analysis and a high performance, power-constrained DRAM scheduling algorithm," Ph.D. dissertation, University of Maryland, 2005.

[45] G. L. Yuan, A. Bakhoda, and T. M. Aamodt, "Complexity effective memory access scheduling for many-core accelerator architectures," ser. MICRO, 2009.

[46] L. Zhang *et al.*, "The impulse memory controller," *IEEE Transactions on Computers*, vol. 50, no. 11, 2001.

[47] X. Zhou *et al.*, "Hierarchical overlapped tiling," ser. CGO, 2012.