

# Collective Memory Transfers for Multi-Core Chips

George Michelogiannakis, Alexander Williams, John Shalf

Lawrence Berkeley National Laboratory, 1 Cyclotron Road, Berkeley, CA 94720

Email: {mihelog,awilliams,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 data caches and scratchpads. However, DRAM performance suffers under the non-streaming access patterns generated by many independent cores. We propose collective memory scheduling (CMS) that actively takes control of collective memory transfers such that requests arrive in a sequential and predictable fashion to the memory controller. CMS uses the hierarchically tiled arrays formalism to compactly express collective operations, which greatly improves programmability over conventional prefetch or list-DMA approaches. CMS reduces application execution time by up to 32% and DRAM read power by  $2.2\times$ , compared to a baseline DMA architecture such as STI Cell.

## 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 [12, 36]. As we adopt a more aggressive many-core strategy, the throughput, latency, and cost of DRAM has emerged to the forefront of research. Memory bandwidth is not scaling rapidly enough to satisfy the increasing number of processors, making the performance of a wide variety of applications constrained by memory bandwidth [15, 31]. In fact, current projections state that chip pins increase by 10% every year whereas on-chip processors double every 18 months [31]. Examples of data-parallel memory bandwidth-bound applications include the Laplacian and wave equations stencil kernel (used in a variety of applications such as seismic simulation), combustion simulation, face recognition, image processing, fluid simulation, embedded applications, and many others. As another example, media applications have been reported to require up to 300GB/s of bandwidth to utilize just 48 processors [29]. Even SPEC benchmarks can saturate memory bandwidth in just eight-core chip multiprocessors (CMPs) [20]. In memory bandwidth-bound applications, techniques that increase memory bandwidth have a direct effect on execution time [29, 30].

Memory power consumption is also crucial, given the limited power budget of large-scale chips. In current technology, reading double-precision operands from DRAM for an addition costs about 2000pJ, while the operation itself consumes approximately 100pJ [36]. This problem has already surfaced in datacenters, where 25%–40% of total power is attributed to DRAM [38]. Therefore, maximizing DRAM efficiency is critical, especially for future systems where DRAM’s contribution will likely be proportionally larger than today [4].

Numerous important 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. 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 memory subsystems.

Modern DRAMs are most efficient when presented with ordered unit-stride access patterns [29, 30, 42]. However, current chip multi-processors 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 [42], which degrades performance and power [38]. Even though a plethora of memory controllers have been proposed, they are typically passive elements which do not control the order requests arrive to them. Therefore, their degree of freedom is limited to the entries in their finite-size transaction queues [27, 30, 41].

In this paper, we demonstrate a hardware approach to coordinating on-chip data movement named collective memory scheduling (CMS), and the programming constructs to make access to this capability efficient and easy to express using the hierarchically tiled array (HTA) abstraction [13]. We demonstrate the effectiveness of CMS for stencil-based computations which are crucial for applications ranging from image processing in consumer or embedded devices, to the largest-scale high performance computing (HPC) applications such as climate modeling. We believe that the kinds of algorithms that are the largest drivers for improved computational performance are in fact SPMD kernels that are seen in image processing, face recognition, machine learning, kinetics simulation, and others. We also show that the CMS engine is inexpensive enough to be included even in general-purpose systems. When not in use, such as with applications without collective transfers, the CMS engine can power down like any accelerator.

In summary, CMS makes the following contributions:

- Provides a simple hardware extension to coordinate complex access patterns across multiple processors to re-establish a streaming access pattern for DRAM to achieve optimal throughput, latency, and power. CMS reduces application execution time by up to 32%, compared to independent direct memory access (DMA) operations in each processor.
- Also due to re-establishing a streaming access pattern, CMS reduces DRAM read power by  $2.2\times$  and DRAM write power by 50%.
- Eliminates network congestion by replacing many independent read and write requests with a handful of control packets.
- Modifies the HTA representation [13] to simplify the application programming interface (API) since the same *collective* function call is made by all processors, with no need to calculate individual DMA address ranges, as in STI Cell [34].

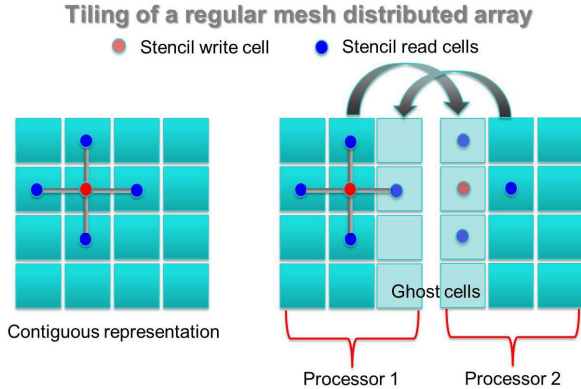


Figure 1: Tiling divides a contiguous distributed array into tiles. Each tile is assigned to a processor. Tiles may include read-only ghost zones that replicate neighboring data. An example 5-point stencil is shown.

```

while (data_remaining)
{
    load_next_tile(); // DMA load
    operate_on_tile(); // Local computation
    write_resulting_tile(); // DMA write
}

```

Figure 2: A computation loop for a local-store architecture.

## 2. Background

### 2.1 Stencil Computations

Domain decomposition is commonly used to expose parallelism for SPMD algorithms that range from linear algebra to stencil algorithms, but poses significant challenges to memory performance of CMPs and graphical processor units (GPUs). To illustrate the benefits of CMS, we focus on stencil algorithms because of their broad applicability, the memory bandwidth sensitivity of their kernels [17], and their ubiquitous usage [28]. In particular, stencil algorithms constitute a large fraction of consumer, embedded, HPC and scientific applications in such diverse areas as image processing, seismic imaging, heat diffusion, electromagnetics, fluid dynamics, and climate modeling [25]. In a stencil operation, each point in a multi-dimensional grid is updated with weighted contributions from a subset of its neighbors in both time and space, thereby representing the coefficients of the partial differential equation (PDE) for that data element. Stencil sizes range from considering only its immediate neighbors to 9-, 13-, 21- and 27-point stencils.

Stencil calculations perform global sweeps through data structures that are typically much larger than the available data caches. As a result, data from main memory often cannot be transferred fast enough to avoid stalling the computational units on modern microprocessors [41]. Reorganizing these computations to fit into the caches has principally focused on tiling optimizations. Tiling for a regular mesh data array and a 5-point stencil that is used for the heat PDE [5, 26] is shown in Figure 1.

With tiling, each processor is typically assigned a contiguous block of stencils (a tile) to operate on within the local high-speed L1 caches. However, stencils at the edge of a tile require data that belongs to tiles of neighboring processors. Therefore, each processor's tile is extended to include read-only *ghost zones* at the edges, which are owned and writable by a neighbor processor. Ghost zones are also shown in Figure 1.

```

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 3: Example HTA declaration code.

An abstract computational loop is shown in Figure 2. In each processor, each iteration operates on a different tile. Because tiles are sized to fit in local caches, there is typically no data reuse across iterations (across tiles of the same processor). Operations in a computation loop can be pipelined by writing the previous iteration's results, computing on the current iteration, and loading the next iteration's tile simultaneously; this requires triple buffering.

### 2.2 Memory Access Streams and Efficiency

Loading a tile causes processors to generate read requests to the memory controller independently of other processors. This is done with local independent hardware prefetch [19] or cache fill streams for a cache-coherent CMP, a list of outstanding load-store requests for a massively multithreaded architecture like a GPU, or via a sequence of DMA requests for a local store architecture like STI Cell [34]. In *all* of these cases, requests are sent independently over an unpredictable network and thus arrive in nearly random order to memory [38, 42].

Random access patterns degrade DRAM performance and power [30, 38, 39] because they cannot take advantage of pre-activated rows and therefore cause more row activations compared to sequential access patterns. Depending on the access pattern, only 14%–97% of memory bandwidth can actually be utilized [29]. As a result, in many workloads the number of times an open row is used before being closed due to a conflict is often one or two [38]. This penalizes both latency and power because opening a new row includes charging bit lines, amplification by sense amplifiers, and then writing bits back to the cells. In addition, 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 and efficiently expresses distributed tile arrays [13]. An example declaration is shown in Figure 3. This declaration divides a  $6 \times 6$  array into  $2 \times 2$  tiles and maps those tiles to a  $3 \times 3$  array of processors, as shown in Figure 4. The HTA library translates data operations to remote data accesses if needed.

## 3. Collective Memory Transfers

### 3.1 Programming Interface

The CMS programming interface is responsible for making the collective transfer capabilities of the hardware CMS engine accessible to the programmer. For the CMS API we adopt the HTA syntax [13] to define a 2D plane of data that a CMS operation handles. We also *modify* the simple HTA syntax to compactly express ghost zones by adding a parameter to denote the number of ghost zone cells in each dimension.

Our extension to HTA is shown in Figure 5. The resulting mapping is illustrated in Figure 4. HTAs have been extended to offer an alternative and more complex but also more powerful syntax to declare ghost zones of arbitrary shapes and sizes [13].

### 3.2 API

We choose to provide access to CMS functionality using a library that exposes an API similar to DMA function calls [34]. This leaves

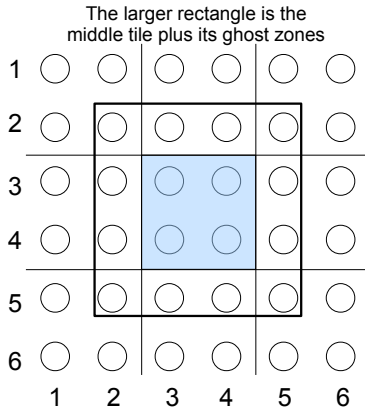


Figure 4: The mapping from our example declaration. Only the ghost zones for the shaded tile are shown.

```
Array = hta(name, {[1,3,5],[1,3,5]},
            1, // One ghost zone cell in each dimension
            [3,3]);
```

Figure 5: The added parameter denotes that there is one ghost zone cell in each dimension.

```
Loading a HTA with a CMS read
HTA_instance = CMS_read (Starting_address,
                        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 6: Without CMS, the programmer needs to calculate starting and ending address for each tile line in a local-store architecture, including ghost zones.

the programming style intact. Once a CMS function call is made, the processor generates a ready packet in the manner described later in this Section. Similar to DMA function calls, the processor also reserves local storage in architectures like STI Cell for incoming data for read operations.

A CMS read or write function call requires only the HTA instance and its starting address in memory as parameters. Since the caller’s identifier is implicit and the HTA instance contains all the tiling and layout information, the CMS library translates virtual to physical memory addresses if necessary and infers exactly what address ranges each processor requires for reading and writing, or only for reading (for its ghost zones). 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, where the programmer has to calculate address ranges individually in order to configure each processor’s DMA engine [34]. In the common case that a processor’s tile consists of non-contiguous memory addresses [14],

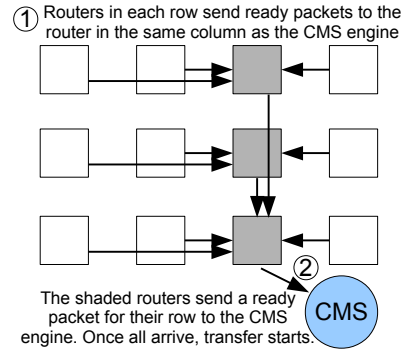


Figure 7: Initiating a synchronous read CMS operation.

a potentially large number of DMA calls is required, which in turn require deep transaction queues in each DMA engine. In contrast, the equivalent CMS operation requires only one function call, as shown in Figure 6.

Although we demonstrate CMS in a local-store architecture with an explicit API, this is not a requirement. GPU programming languages can identify collective transfers abstractly from the programmer. Also, compilers or run-time systems can analyze memory access patterns and data structure layouts to identify collective operations. Finally, in hardware-managed cache-coherent CMPs, prefetching and cache miss handling can be performed collectively at a HTA granularity instead of locally by each processor.

### 3.3 Read Operations

In reads, the CMS engine reads memory sequentially and distributes data to the appropriate processors according to the HTA mapping. We implement synchronous and asynchronous reads. In the synchronous case, the CMS engine initiates the transfer when all processors make the synchronous read function call for the same HTA. Asynchronous reads are used when the implicit barrier of synchronous reads is not desired. With asynchronous reads, the transfer initiates when the first processor is ready. This requires non-ready processors to buffer the next iteration’s HTA.

To coordinate operation start in the synchronous case we employ a simple hierarchical communication pattern, shown in Figure 7. As shown, a processor sends its ready packet—generated after making the CMS read function call—to the processor which shares a dimension (e.g., column) with the CMS engine. Once these intermediate processors receive a ready packet from their entire row, they send a collective ready packet on behalf of their row to the CMS engine. Transfer initiates when the CMS engine receives a collective ready packet from each row. Ready packets contain the base address, transfer count and the HTA information (such as the tiling and layout).

Once transfer initiates, the CMS engine reads memory sequentially and sends each tile line to the appropriate processor as specified by the HTA declaration. For ghost zone data, the CMS engine sends a copy of the packet that it sent to the owner processor, to the reader processor. This occurs as data is read from memory, ensuring that all data is read only once.

### 3.4 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 is *reading* data from the processors and writing it into memory. When the processor that holds the first tile line of the HTA is ready to write its tile, it sends a write ready packet to the CMS engine containing the HTA information

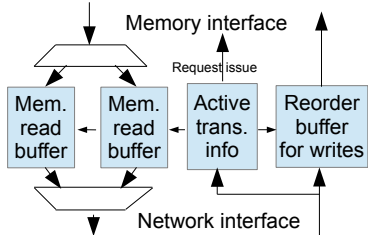


Figure 8: CMS engine outline.

to initiate the write operation. That information includes the base address, transfer count and HTA information such as tiling and layout. The CMS engine then sends read requests in units of tile lines to retrieve the HTA in memory address order. In the mapping of Figure 4, 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.

Because the network guarantees no ordering, the CMS engine uses a small reorder buffer to enqueue read replies write to memory in address order. The number of outstanding read requests defines the size of the reorder buffer, which needs to be deep enough to eliminate memory idle cycles. In our  $8 \times 8$  2D mesh, the optimal size for the reorder buffer is six transactions for HTAs of  $512 \times 512$  elements, four transactions for  $1024 \times 1024$ , and three for  $2048 \times 2048$  HTAs.

### 3.5 Collective Memory Scheduling Engine

We implement the CMS engine as a “stencil engine” atop a typical DMA engine. As illustrated in Figure 8, the CMS engine has a memory interface side and a network interface side. When a valid read or write command appears at the network interface, the CMS engine records the HTA’s starting address and its 4 dimensions (elements in a tile’s row, elements in a tile’s column, tiles in a HTA row, tiles in a HTA column). The engine then breaks the large operation into smaller memory-sized ones and tracks its position in the operation with counters representing each dimension of the HTA. 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 from the network interface side.

The allowed number of pending memory transactions depends on the size of the stencil engine’s buffers. Read operations use two small  $16 \times 128$  bit buffers (“mem. read buffers”) for the outstanding DRAM read requests and to permit duplicating ghost zone packets (in this implementation the memory controller interface is 128 bits).

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 when data is read from the reorder buffer and written to memory.

The CMS engine can be integrated into the memory controller instead of remaining a separate entity like a DMA engine, but we leave this for future work. Furthermore, to reduce communication delay, we co-locate a CMS engine with each memory controller. 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. The additional complexity of the CMS engine compared to a typical DMA engine is outweighed by the vastly reduced memory controller complexity compared to modern memory controllers with large transaction queues and complex scheduling policies [27, 30]. Moreover, CMS engines *replace* individual processor DMA engines or prefetch units because the CMS engine performs the entire operation instead of individual processors. Simplifying the memory controller to FIFO scheduling and removing prefetch units may be inefficient in systems that do not predominantly use collective data transfers. That’s because the performance degradation for non-collective transfers may become a significant factor in system performance. However, as we show in Section 4.2.5, CMS engines are inexpensive enough to be included even in general purpose systems that do not frequently perform collective transfers. When those systems execute non-stencil algorithms, CMS engines remain inactive and can power down, similar to any other accelerator.

## 4. Evaluation

### 4.1 Methodology

We use a heavily-modified version of the Booksim network simulator to model a local-store architecture similar to STI Cell [34] including processors, memory controllers, and local storage [16]. Initially we simulate writes and synchronous reads operations of single HTAs. HTAs are 2D and range from  $64 \times 64$  to  $2048 \times 2048$ . Variables are 8-byte double precision. We use 5-point stencils such as for the heat PDE [5]. Therefore, each processor tile requires two ghost zone elements per row and two per column (one element on each side).

We then present application results for the following important stencil-based applications: fluid animation from the PARSEC benchmark suite [3], geometric multi-grid calculations (GMG) [40], seismic wave propagation simulation (RTM) [23], the SOBEL filter used extensively for image processing [10], and a collection of Laplacian stencil kernels [18]. For the application results, we model Intel Phi co-processors. For each application, we calculate the processing time per variable as well as the ghost zone sizes, and simulate ten iterations of each application’s execution loop, shown in Figure 2. We use the typically-used row-major mapping of each distributed array to memory (column-major mapping would produce comparable results).

Our default proxy CMP consists of an  $8 \times 8$  grid of processors. Four memory controllers are placed at the corners. Each memory controller is co-located with a CMS engine. We use static address-based mapping to map tile lines (memory addresses) to memory controllers. A 2D mesh on-chip network is used with dimension-order routing (DOR) and four-stage input-buffered routers [8]. 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 use DRAMSim2 to simulate a Micron 16MB DDR3 1600MHz memory module with a 64-bit data path and two ranks with 8 banks each [32]. There is a single memory controller for the two ranks. The memory controller has 32-slot transaction and DRAM command reorder queues, and First Ready First Come First Served (FRFCFS) scheduling [30, 42]. Our FRFCFS scheduler uses an open-row policy which respects row buffer locality by prioritizing transactions to open DRAM rows. We compare CMS against FRFCFS because FRFCFS maximizes memory throughput compared to a variety of other controllers [30]. FRFCFS does not necessarily minimize application execution time because maximizing memory throughput may be

unfair to threads [27]. However, we do not model and therefore hold these adversary effects against the baseline case. We assume the same frequency of 1600MHz for the simple cores and the network.

## 4.2 Results

### 4.2.1 Memory Throughput Degradation:

First, we illustrate 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 the transaction queue.

To set up this experiment, we use DRAMsim2 [32] to simulate a synthetic 16MB in-order trace of loads to represent the “coordinated” CMS case, and an out-of-order trace to simulate the uncoordinated case where loads or stores are presented to the memory controller in random order. The uncoordinated requests are randomly-ordered in sizes of 128 bytes, representing one tile line. Experiments with access traces larger than 16MB produce comparable results.

Our results show that for the uncoordinated access pattern (baseline), 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 maximum theoretical throughput, 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 due to a similar decrease in row buffer hit rates). 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 [30, 38, 39]. 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 [38]. For the rest of our evaluations, we use a 25% lower DRAM read throughput for the baseline case, and 41% for write operations.

### 4.2.2 Operation Completion Time:

Figure 9 (left) shows execution times for completing a single read or write CMS operation. Compared to the baseline with FRFCFS, CMS reduces completion time by 39% for reads and 38% for writes. These gains are due to:

- The lower throughput the DRAM provides with random (uncoordinated) access patterns.
- Eliminating redundant memory reads in read CMS operations compared to the baseline, since data is read only once and submitted to the owner and reader processors, instead of each processor retrieving its ghost zones separately. With a 256×256 HTA, there are 12% fewer reads with CMS.

We then repeat the experiments, but with a uniform random (UR) background traffic pattern with a 10% flit injection 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 9 (center), the reduction in execution time for CMS in the mesh is 46% for reads and 36% for writes. While

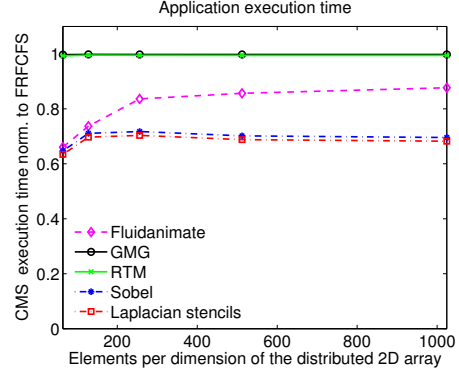


Figure 10: Application speedup for CMS.

background traffic degrades performance for CMS, it is more adversary to the baseline because read and write requests are queued in the network. Figure 9 (right) better illustrates the impact to the background traffic. Baseline operations saturate the network and raise the average background traffic latency to thousands or millions of clock cycles in our simulations (latencies in saturated networks are unbounded). In contrast, CMS keeps the average background traffic latency to 30–50 cycles. These gains for CMS are due to:

- Replacing the large number of read or write requests in the uncoordinated case with a few ready packets for the entire transfer. This alleviates contention in the network.
- Bounding the number of in-flight read reply packets, which carry data from processors to memory in CMS write operations, because of the reorder buffer in the CMS engine. This also alleviates network congestion.

### 4.2.3 Impact on Application Execution Time:

We show our application benchmark results in Figure 10. The gains depend on the ratio of the time spent computing in each iteration versus completing a read and a write operation. Applications that are compute-bound in our system (RTM and GMG) receive minimal (0%–1%) execution time benefit from CMS. In contrast, memory bandwidth-bound applications directly benefit from CMS. Specifically, by average across HTA sizes, fluidanimate requires 21% fewer cycles, the Sobel filter 31% fewer cycles and the Laplacian stencils kernel 32% fewer cycles. The energy reduction benefits from CMS remain for both compute-bound *and* memory bandwidth-bound applications.

### 4.2.4 Sensitivity to System Configuration:

We then repeat our operation completion experiments with a 144-processor system and then the original 64-processor system with a ghost zone of twice the size using a 9-point stencil such as for S3D which models turbulent combustion [6]. Operation completion gains for CMS are comparable to Section 4.2.2. This remains true except for a large ghost zone size to tile size ratio which benefits CMS, because CMS’s benefit of reading data destined to two processors only once is amplified with larger ghost zones. Finally, repeating our experiments with only one or two memory controllers slightly favors CMS because the baseline case produces more severe network hotspots.

### 4.2.5 CMS Engine Implementation Results and Energy:

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 also synthesize the same

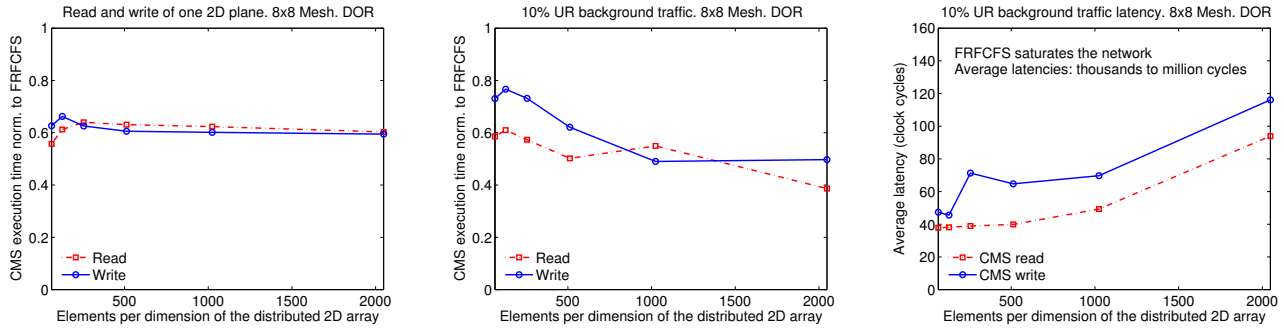


Figure 9: CMS read and write operation completion time normalized to the baseline, and the impact on background traffic.

Table 1: RTL synthesis results.

	DMA	CMS
ASIC		
Combinational area ( $\mu\text{m}^2$ )	743	16231
Non-combinational area ( $\mu\text{m}^2$ )	419	61313
Minimum cycle time (ns)	0.6	0.75
FPGA		
LUTs for logic	245	856
Minimum cycle time (ns)	4.4	5.1

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 reorder buffer for write operations is sized to hold eight transactions of  $16 \times 128$  bits each, for a total of 2KB. 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 results are presented in Table 1.

As shown, cycle time for the CMS engine increases by 25% in the ASIC flow and 16% in the FPGA flow. To make the CMS engine operate at the same clock frequency as the DMA engine, we can simply pipeline the CMS engine by adding one more stage. The one extra cycle is a negligible timing overhead compared to the duration of an operation.

Despite the increased complexity of the CMS engine, CMS can significantly simplify other parts of the system. Specifically, when performing collective operations, 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 [30] and perform an associative comparison of all requests in the transaction queue every cycle (therefore requiring comparators for every queue entry), and then issue a transaction from any position in the queue based on multi-level priority and other complex schemes [30, 42]. As an example of the savings obtained by the shallower transaction queue, a ternary content addressable memory (TCAM) that holds eight transactions has a 10%–130% shorter cycle time, is  $4 \times$  smaller, and requires up to  $1.5 \times -4 \times$  less access energy compared to a transaction queue of 32 entries which is typical for a modern memory controller transaction queue TCAM [2, 30]. CMS also replaces DMA engines in *each* processor.

CMS also reduces DRAM access power because the baseline case consumes  $2.2 \times$  more dynamic power for reads and 50% more for writes. This is because of the extra row activations in the baseline case, which increases activate and precharge power. 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 [36]. These projections, combined with attributing 25%–40% of total datacenter power to the DRAM system [38], make CMS’s DRAM power reductions critical.

## 5. Discussion

CMS readily applies to GPU architectures, due to their similarity with our local-store architecture and the wide variety of stencil algorithms they execute with similar memory access patterns as our evaluations, such as image processing [1]. Furthermore, while in local-store architectures such as STI Cell [34] 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 transfer opportunities. Prefetch decisions can also be performed in the CMS engine by observing the access stream, without prefetch engines at each processor.

Future work on the concept of CMS will focus on more flexible memory access scheduling by having the software construct the address-order memory access pattern in a distributed manner, such as the communication pattern of Figure 7. This way, the CMS hardware engine will simply execute the generated stream and need not make any assumptions for data layout, such that CMS can apply to more than stencil-based computations.

## 6. Related Work

Past work has researched similar collective data transfer techniques in very different contexts. In wide-area networks, coordinating the nodes in a TCP/IP network to send their data to a common destination in with a common transfer schedule that avoids conflicts substantially reduces network congestion [7]. Alternative techniques for wide-area networks focus on heterogeneity and use of shared resources by transferring different chunks of the same file from replicas and taking network bandwidth into account [22]. Collective data transfers have also been applied for server disk-directed I/O, because the access bandwidth for traditional hard disk drives significantly improves with sequential accesses [35].

Classic vector machines such as the Cray-1 [33] 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 [11]. In addition, the Impulse memory controller overcomes the inefficiency of sparse access patterns due to cache-line granularity issues by reorganizing the memory address stream so that sparse address pattern appears contiguously in the cache hierarchy [43]. However, with Impulse the data arrays remain scattered in the DRAM, thereby leading to inefficient DRAM performance due to overfetch.

Sophisticated memory schedulers use complex scheduling policies, and can use different policies for threads according to their memory access characteristics or quality of service guarantees [27, 30, 41]. Many schedulers, such as PAR-BS, perform limited re-ordering by attempting to exploit row buffer locality and bank parallelism among other metrics [27]. 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. 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 [29]. Since transaction queues are not of infinite size, the result is far from the complete memory address order that CMS achieves because CMS *actively takes charge* of collective memory operations. 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

Because sophisticated memory schedulers require associative comparison of all queued transactions every cycle, past work has simplified memory controllers by using the on-chip routers to re-order requests [42]. 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 [24]. 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 [37]. Modifications to DRAM internals have been proposed to mitigate the negative power effects of random-order sequences, by avoiding activating all the bitlines in a row before the exact read request is known [38].

Past work has repeatedly reported that a wide variety of applications are constrained by memory bandwidth [9, 15, 31, 36]. In those cases, while local and last-level caches can eliminate DRAM accesses during the computation phase of a loop, data is still retrieved from main memory when loading new and storing old HTAs, which is the focus of CMS. Last-level caches can partially reconstruct address order for writes with a write back policy. However, streaming (write-through) writes are preferable to write back policies in stencil-based computations to avoid polluting higher-level caches because the results of a computation loop are not reused in the next iteration [15]. Even with a write-back policy, caches are constrained by their size and the unpredictability of the incoming packers, similar to a memory controller. Memory prefetching techniques focus on reducing latency and offer little benefit in systems that are bound by memory bandwidth. Prefetching techniques typically perform predictions independently at each processor and thus create out-of-order access patterns [19].

Polyhedral representations alternative to HTAs are also applicable to CMS [21]. Polyhedral representations are not a prerequisite for CMS because collective memory transfers can be expressed even with basic language constructs.

## 7. Conclusion

To make optimal use of the limited memory bandwidth of current and future systems, we present CMS to coordinate parallel data access in a chip multi-processor such that distributed arrays of data are read from or written to the DRAM in strict memory address order. CMS is a hardware technique that programming constructs access. 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 of SPMD codes, and alleviates network congestion. These gains result in up to 32% lower application execution time, up to  $2.2\times$  less power for memory reads, and 50% less power for memory writes.

## Acknowledgments

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

## Disclaimer

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor the Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or the Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or the Regents of the University of California.

## Copyright Notice

This manuscript has been authored by an author at Lawrence Berkeley National Laboratory under Contract No. DE-AC02-05CH11231 with the U.S. Department of Energy. The U.S. Government retains, and the publisher, by accepting the article for publication, acknowledges, that the U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for U.S. Government purposes.

## References

- [1] A. Abdelfattah, J. Dongarra, D. Keyes, and H. Ltaief. Optimizing memory-bound numerical kernels on GPU hardware accelerators. July 2012.
- [2] B. Agrawal and T. Sherwood. Ternary CAM power and delay model: Extensions and uses. *IEEE Transactions on VLSI*, 16(5):554–564, 2008. .
- [3] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.
- [4] S. Borkar and A. A. Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, 2011. ISSN 0001-0782. .
- [5] R. Chamberlain. Solving partial differential equations on a parallel supercomputer. In *IEEE Colloquium on Parallel Processing: Industrial and scientific applications*, 1991.
- [6] J. H. Chen, A. Choudhary, B. de Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W. K. Liao, K. L. Ma, J. Mellor-Crummey, N. Podhorszki, R. Sankaran, S. Shende, and C. S. Yoo. Terascale direct numerical simulations of turbulent combustion using S3D. *Computational Science and Discovery*, 2(1):015001, 2009.
- [7] W. C. Cheng, C.-F. Chou, L. Golubchik, S. Khuller, and Y.-C. Wan. A coordinated data collection approach: design, evaluation, and comparison. *IEEE Journal on Selected Areas in Communications*, 22(10): 2004–2018, 2006. ISSN 0733-8716. .

- [8] W. J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., 2003. ISBN 0122007514.
- [9] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing, SC*, pages 4:1–4:12, 2008.
- [10] W. Gao, X. Zhang, L. Yang, and H. Liu. An improved sobel edge detection. In *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on*, volume 5, pages 67–71, 2010.
- [11] J. Gebis, L. Oliker, J. Shalf, S. Williams, and K. Yelick. Improving memory subsystem performance using ViVA: Virtual vector architecture. In *Proceedings of the 22nd International Conference on Architecture of Computing Systems, ARCS '09*, pages 146–158, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-00453-7.
- [12] M. Ghasemazar, E. Pakbaznia, and M. Pedram. Minimizing the power consumption of a chip multiprocessor under an average throughput constraint. In *International Symposium on Quality Electronic Design, ISQED '10*, pages 362–371, 2010. ISBN 978-1-4244-6455-5.
- [13] J. Guo, G. Bikshandi, B. B. Fraguera, and D. Padua. Writing productive stencil codes with overlapped tiling. *Journal on Concurrency and Computation: Practice and Experience*, 21(1):25–39, 2009.
- [14] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan. Data layout transformation for stencil computations on short-vector SIMD architectures. In *Proceedings of the 20th international conference on Compiler construction: part of the joint European conferences on theory and practice of software, CC'11/ETAPS'11*, pages 225–245, 2011. ISBN 978-3-642-19860-1.
- [15] J. Holewinski, L.-N. Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on GPU architectures. In *Proceedings of the 26th ACM international conference on Supercomputing, ICS*, pages 311–320, 2012.
- [16] N. Jiang, D. Becker, G. Michelogiannakis, J. Balfour, B. Towles, D. Shaw, J. Kim, and W. Dally. A detailed and flexible cycle-accurate network-on-chip simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS '13*, pages 86–96, 2013.
- [17] S. Kamil, C. Chan, S. Williams, L. Oliker, J. Shalf, M. Howison, E. W. Bether, and Prabhat. A generalized framework for auto-tuning stencil computations. In *Cray User Group*, 2009.
- [18] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *IEEE International Symposium on Parallel Distributed Processing*, pages 1–12, 2010.
- [19] M. Kandemir, Y. Zhang, and O. Ozturk. Adaptive prefetching for shared cache based chip multiprocessors. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, 2009.
- [20] D. Kaseridis, J. Stuecheli, and L. K. John. Minimalist open-page: a dram page-mode scheduling policy for the many-core era. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-44 '11*, pages 24–35, 2011.
- [21] K. Keahey, P. Fasel, and S. Mniszewski. PAWS: collective interactions and data transfers. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing*, pages 47–54, 2001.
- [22] G. Khanna, U. Catalyurek, T. Kurc, R. Kettimuthu, P. Sadayappan, and J. Saltz. A dynamic scheduling approach for coordinated wide-area data transfers using GridFTP. In *IEEE International Symposium on Parallel and Distributed Processing, IPDPS '08*, pages 1–12, 2008.
- [23] J. Krueger, D. Donofrio, J. Shalf, M. Mohiyuddin, S. Williams, L. Oliker, and F.-J. Pfreund. Hardware/software co-design for energy-efficient seismic modeling. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11*, pages 73:1–73:12, 2011.
- [24] D. Lee, S. Yoo, and K. Choi. Entry control in network-on-chip for memory power reduction. In *Proceedings of the 13th international symposium on Low power electronics and design, ISLPED*, pages 171–176, 2008. ISBN 978-1-60558-109-5.
- [25] J. Meng and K. Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In *Proceedings of the 23rd international conference on Supercomputing, ICS '09*, pages 256–265, 2009.
- [26] M. Mohiyuddin, M. Murphy, L. Oliker, J. Shalf, J. Wawrzynek, and S. Williams. A design methodology for domain-optimized power-efficient supercomputing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC*, 2009.
- [27] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *Proceedings of the 35th Annual International Symposium on Computer Architecture, ISCA '08*, pages 63–74, 2008.
- [28] L. Peng, R. Seymour, K.-i. Nomura, R. K. Kalia, A. Nakano, P. Vashishta, A. Loddock, M. Netzband, W. Volz, and C. Wong. High-order stencil computations on multicore clusters. In *IEEE International Symposium on Parallel Distributed Processing, IPDPS '09*, pages 1–11, 2009.
- [29] S. Rixner. *A bandwidth-efficient architecture for a streaming media processor*. PhD thesis, 2001. AAI0803043.
- [30] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *Proceedings of the 27th annual international symposium on Computer architecture, ISCA '00*, pages 128–138, 2000. ISBN 1-58113-232-8.
- [31] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin. Scaling the bandwidth wall: challenges in and avenues for CMP scaling. In *Proceedings of the 36th annual international symposium on Computer architecture, ISCA '09*, pages 371–382, 2009.
- [32] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters*, 10(1):16–19, 2011.
- [33] R. M. Russell. The CRAY-1 computer system. *Communications of the ACM*, 21(1):63–72, 1978. ISSN 0001-0782.
- [34] S. Schneider, J.-S. Yeom, and D. S. Nikolopoulos. Programming multiprocessors with explicitly managed memory hierarchies. *IEEE Computer*, 42(12):28–34, 2009. ISSN 0018-9162.
- [35] K. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '95*, 1995.
- [36] J. Shalf, S. S. Dossanjh, and J. Morrison. Exascale computing technology challenges. In *International Meeting on High Performance Computing for Computational Science, VECPAR '10*, 2010.
- [37] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis. Micro-pages: increasing DRAM efficiency with locality-aware data placement. In *Proceedings of the 15th edition of ASPLOS on Architectural support for programming languages and operating systems, ASPLOS '10*, pages 219–230, 2010.
- [38] A. N. Udipi, N. Muralimanohar, N. Chatterjee, R. Balasubramonian, A. Davis, and N. P. Jouppi. Rethinking DRAM design and organization for energy-constrained multi-cores. In *Proceedings of the 37th annual international symposium on Computer architecture, ISCA '10*, pages 175–186, 2010. ISBN 978-1-4503-0053-7.
- [39] D. T. Wang. *Memory DRAM memory systems: performance analysis and a high performance, power-constrained DRAM scheduling algorithm*. PhD thesis, University of Maryland, 2005.
- [40] S. Williams, D. Kalamkar, A. Singh, A. Deshpande, B. Van Straalen, M. Smelyanskiy, A. Almgren, P. Dubey, J. Shalf, and L. Oliker. Optimization of geometric multigrid for emerging multi- and manycore processors. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11, 2012.
- [41] D. Xu, C. Wu, and P.-C. Yew. On mitigating memory bandwidth contention through bandwidth-aware scheduling. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques, PACT '10*, pages 237–248, 2010.
- [42] G. L. Yuan, A. Bakhoda, and T. M. Aamodt. Complexity effective memory access scheduling for many-core accelerator architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO*, pages 34–44, 2009.
- [43] L. Zhang, Z. Fang, M. Parker, B. K. Mathew, L. Schaelicke, J. B. Carter, W. C. Hsieh, and S. A. McKee. The impulse memory controller. *IEEE Transactions on Computers*, 50(11):1117–1132, 2001.