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×, 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 [20, 21, 64, 25, 29, 3]. Single-chip high-performance multi-processors, such as chip multiprocessors (CMPs) and graphical processor units (GPUs), are anticipated to having thousands of processing elements as soon as 2018 [8, 9, 68]. These systems are crucial for applications that range from consumer electronics and high-performance embedded to high performance computing (HPC), where performance improvements depend on the parallel efficiency of key algorithms.

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 [70, 66, 59, 12, 32, 18, 19, 28, 30, 2, 55]. In fact, current projections state that chip pins increase by 10% every year whereas on-chip processors double every 18 months [59]. 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; this limits performance in latency-sensitive systems [10, 48]. 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 [46]), combustion simulation [12], face recognition [38], image processing [1], fluid simulation [53], embedded applications [75, 45], and many others [36, 18, 12, 1]. Media applications have also been reported to require up to 300GB/s of bandwidth to utilize just 48 processors [57]. Even SPEC benchmarks can saturate memory bandwidth in just eight-core CMPs [39]. In memory bandwidth-bound applications, techniques that increase memory bandwidth have a direct effect on execution time [65, 70].

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 [64]. This problem has already surfaced in datacenters, where 25%–40% of total power is attributed to DRAM [69]. Therefore, maximizing DRAM efficiency is critical, especially for future systems where DRAM's contribution will likely be proportionally larger than today [10, 21].

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. Examples include sparse and dense linear algebra kernels, FFTs used in spectral methods for filtering, stencil kernels for image processing and seismic imaging [46], and fluid dynamics simulation [12, 53]. 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 [58, 76, 31, 57]. 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 [69, 76], which degrades performance and power [5]. 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 [74, 58, 31, 65, 44, 19, 54].

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 [22, 26]. CMS coordinates processors such that distributed data arrays are read from or written to the DRAM as a unit, and transfered to or from the appropriate processors. Memory access and distribution are handled by the CMS hardware engine which replaces individual processor prefetch or direct memory access (DMA) engines. Contrary to memory controllers which are passive elements, the CMS engine *actively controls* collective data transfers to guarantee address ordering of requests to memory.

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 HPC applications such as climate modeling. CMS also applies non-stencil distributed data algorithms such as distributed matrix multiplication. 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.

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 the completion time for a read or write operation of a distributed array by 39% and 38% respectively, which results in an up to 32% reduction in application executing time, compared to independent DMA or prefetch operations in each processor.
- Also due to re-establishing a streaming access pattern, CMS reduces DRAM read power by 2.2× and DRAM write power by 50%.
- In systems that predominantly perform collective transfers, CMS eliminates the need for costly and deep transaction queues, which modern memory controllers use to partially recover a streaming access pattern [44, 71, 76, 31, 58, 54].
- Eliminates network congestion by replacing many independent read and write requests with a handful of control packets. Performing independent DMA operations saturates background traffic, whereas with CMS the average background latency remains just 30–50 cycles.
- Modifies the HTA representation [22, 26] which defines distributed data arrays and their mapping to processors, 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 [62]. While CMS can use alternative software constructs, HTAs simplify the programming interface.

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 GPUs. To illustrate the benefits of CMS, we focus on stencil algorithms because of their broad applicability, the memory bandwidth sensitivity of their kernels [36, 18, 12, 1], and their ubiquitous usage [55]. In particular, stencil algorithms constitute a large fraction of consumer, embedded, HPC and scientific applications in such diverse areas as image processing, seismic imaging [46], heat diffusion, electromagnetics, fluid dynamics, and climate modeling [51, 52, 78, 56]. These applications often use iterative finite-difference techniques, which sweep over a spatial grid, performing nearest neighbor computations called stencils. 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 [14, 11, 78, 56].

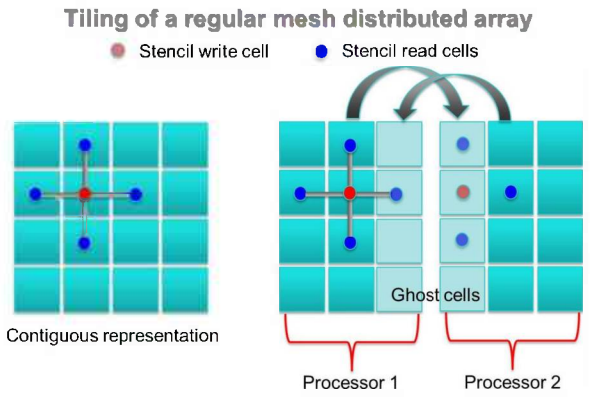

Figure 1: Tiling divides a contiguous distributed array into tiles. Each tile is assigned to a processor. Tiles may include read-only ghost zones that replicate neighboring data. An example 5-point stencil is shown.

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 [74, 18, 12, 70, 66]. Reorganizing these computations to fit into the caches has principally focused on tiling optimizations that exploit locality by performing operations on cache-sized blocks of data in each processor before moving on to the next block [56]. This way, computing the new values for a cache-sized block does not generate memory accesses, but the memory accesses to load new data and store old data are still limited by memory bandwidth, making the entire application limited by memory bandwidth for a wide variety of applications [18]. Domain decomposition for stencils is typically referred to as "tiling" due to the appearance of the domain decomposed grid. Breaking data arrays up into these tiles expresses parallelism and improves spatial locality. Tiling for a regular mesh data array and a 5-point stencil that is used for the heat PDE [52, 11] is shown in Figure 1.

With tiling, each processor is typically assigned a contigu-

```
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.

ous 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.

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. Reducing this to double buffering requires that the previous iteration's tile must be fully written to memory before reading the next iteration's tile.

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 [37] 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 [62]. In *all* of these cases, requests are sent independently over an unpredictable network and thus arrive in nearly random order to memory [76, 69]. Therefore, the memory access stream contains little locality.

Unpredictable memory access streams make extracting benefits from memory prefetching difficult [37, 57]. Depending on the access pattern, only 14%–97% of memory bandwidth can actually be utilized [57]. Random access patterns degrade DRAM performance and power [71, 58, 5, 69] because they cannot take advantage of pre-activated rows and therefore cause more row activations compared to sequential access patterns. The result is that requests are more likely to activate a new row. This is known as *overfetch* [5, 69]. 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 [69]. 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. We quantify this performance and power efficiency loss in Section 4.2.1.

Uncoordinated requests also cause redundant memory accesses, because ghost zones are read multiple times—once by the owner processor and one by each read-only neighbor. Finally, multiple independent requests congest the network

```
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.

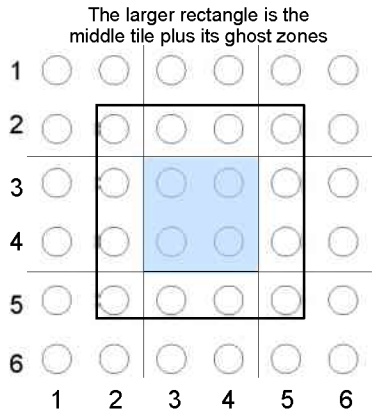

Figure 4: The mapping from our example declaration. Only the ghost zones for the shaded tile are shown. Vertical and horizontal lines are tile boundaries.

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 [22]. The declaration of HTAs includes how the distributed array is tiled and how the tiles map to processors. An example declaration is shown in Figure 3. This declaration divides a 6×6 array into 2×2 tiles and maps those tiles to a 3×3 array of processors, as shown in Figure 4. The first two parameters define tile boundaries, while the last parameter defines the dimensions of the processor array. HTAs overload data operations and array indexing to resemble those of local arrays. 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 [22] 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. We choose HTAs for the CMS API for simplicity. Collective operations can be expressed in software with other polyhedral representations instead, or even with basic language constructs.

Our extension to HTA is shown in Figure 5. The added parameter denotes how many ghost zone cells in each dimension each tile in the HTA has. The resulting mapping is illustrated in Figure 4. HTAs have been extended to offer an alternative

```
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.

and more complex but also more powerful syntax to declare ghost zones of arbitrary shapes and sizes [26].

3.2. API

We choose to provide access to CMS functionality using a library that exposes an API similar to DMA function calls [62]. This leaves the programming style intact and simply requires the programmer to use CMS function calls instead of DMA function calls. 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, or points to the local data that needs to be transfered to memory for write 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 [62]. In the common case that a processor's tile consists of non-contiguous memory addresses [27, 55, 32, 67], a potentially large number of DMA calls is required, which in turn require deep transaction queues in each DMA engine [40]. As an example, to transfer a tile in a 64-core system from a 2048×2048 HTA without architectural support for strided memory access, similar to the STI Cell [62], each processor requires 256 separate

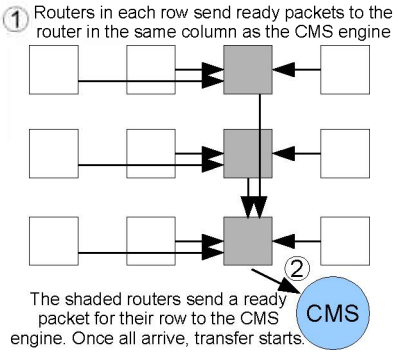

Figure 7: Initiating a synchronous read CMS operation.

DMA transfers. That is because each processor's tile contains 256 rows and a DMA transfer can only fetch a single row since different rows of the same tile are disjoint in memory address order with row- or column-major mappings, which are typically used. This is aggravated with ghost zones due to the additional transfers. 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. Therefore, processors require double buffering because they cannot receive the next iteration's HTA before completing their computation on the previous iteration's HTA. However, processors may block and wait for others to become ready. Because barrier calls are typical in computation loops [28], synchronous reads introduce no additional waiting and can replace barrier calls.

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. Therefore, processors require triple buffering.

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

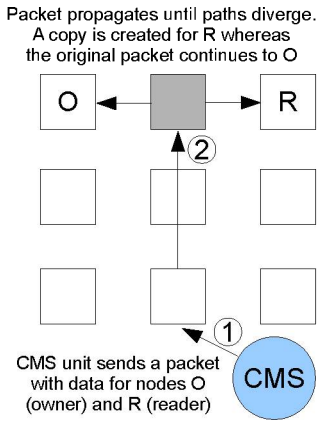

Figure 8: Using multicast to reduce propagation of bits.

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), or simply instruct the CMS engine to reuse the tiling layout it used for its last operation but with a different starting address; this reduces the size of ready packets.

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. Multicast Packets

For ghost zone data, the energy cost can be reduced by adding multicast functionality [33]. With multicast, ghost zone data is sent only once, instead of with different packets to different destinations. At the router where the paths to the owner and reader processors diverge, routers create a copy of the packet for the reader processor, while the original packet continues to the owner processor. This is shown in Figure 8. Multicast reduces propagation energy and contention in the network because only one packet traverses the common path between the CMS engine and the two destinations, instead of two. Efficient implementations of multicast in on-chip routers only extend cycle time by 1%, area by 5%, and power by 2% [72].

3.5. 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 to initiate the write operation. That information includes the base address, transfer count and HTA information such as tiling and layout. Similar to read

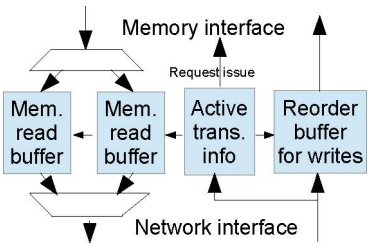

Figure 9: CMS engine outline.

operations, the control packet can instruct the CMS engine to reuse the previous operation's HTA information, except for the starting address. 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. Processors may delay their response until they produce the data.

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 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. However, excessively large reorder buffers are not only costly but also stress the network since they allow more read replies to be in transit, causing contention.

In our 8×8 2D mesh, the optimal size for the reorder buffer is six transactions for HTAs of 512×512 elements, four transactions for 1024×1024, and three for 2048×2048 HTAs. These are the minimum numbers to ensure that the CMS engine constantly has data to write to memory. In the first case there are $\frac{512}{8}\times6=384$ HTA elements (variables) in flight, in the second case 512 elements, and in the third case 768 elements. Larger HTAs increase tile line size and therefore fewer pending reads are required to cover the round-trip latency.

3.6. Collective Memory Scheduling Engine

We implement the CMS engine as a "stencil engine" atop a typical DMA engine. As illustrated in Figure 9, 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. The information of the active transfer, progress, and list of memory-sized transactions remaining reside in the "active trans. info" block until the operation completes. 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. Once the internal command counter reaches its limit, the DMA engine returns to idle, waiting to accept a new operation.

The allowed number of pending memory transactions depends on the size of the stencil engine's buffers. Read operations use two small 16×128 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). No new DRAM read requests may be issued until one of the two buffers is free. With the HTAs evaluated in this paper, two small 16×128 bit buffers were enough to prevent stalls on the memory side. A small state machine handles ghost zone packet duplication. The CMS engine does not return to idle (indicating operation completion) until packets are sent to all participating processors.

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 reorder buffer size depends on the allowed number of outstanding read requests in flight.

The CMS engine can also include a small queue to store pending collective transfer requests, but only one is active at a time. Moreover, the CMS engine is not invoked for non-collective transfers. 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. In local-store architectures, CMS does not require hardware support in processors since the required functionality is implemented in software.

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 [58, 31, 65, 44, 19, 54]. 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 4.2.6, 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, similar to any other accelerator. In addition, a wide-variety of important systems and applications use predominantly collective data transfers, such as such as STI Cell, GPUs, image processing, face recognition, combustion simulation, fluid modelling, wave equation processing, and Laplacian stencil kernels.

3.7. Collective Operation Overview

In summary, a CMS operation has the following steps:

- The programmer declares the HTA and uses the CMS API function call appropriate for the desired transfer.
- Each processor generates a ready packet containing the starting address and HTA information. Processors follow the communication pattern appropriate to the type of transfer, such as that of Figure 7 for synchronous reads. In case of reads, processors reserve space in their local storage.
- The CMS engine performs the collective transfer.

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 [62] including processors, memory controllers, and local storage [34]. Initially we simulate writes and synchronous reads operations of single HTAs. HTAs are 2D and range from 64×64 to 2048×2048. Variables are 8-byte double precision. We use 5-point stencils such as for the heat PDE [11]. 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 [6], geometric multi-grid calculations (GMG) [73], seismic wave propagation simulation (RTM) [46], the SOBEL filter used extensively for image processing [23], and a collection of Laplacian stencil kernels [35]. For the application results, we model Intel Phi co-processors, which are simple x86-based processors and representative of the simple cores projected for future many-core chips [9, 68]. 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 assume enough local storage for triple buffering (in general-purpose systems the software runtime can resize tiles accordingly). 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 8×8 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. Therefore, each processor requests each tile line from the appropriate memory controller and CMS engine. With a simple location-based mapping, most

processors access only one memory controller. The pin pressure on current large-scale chips make more than four memory controllers a challenge to support [2, 59]. A 2D mesh on-chip network is used with dimension-order routing (DOR) and four-stage input-buffered routers [17]. 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 [60]. 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 [58, 76]. 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 [58, 31, 44, 19, 54]. We compare CMS against FRFCFS because FRFCFS maximizes memory throughput compared to a variety of other controllers [65, 58]. FRFCFS does not necessarily minimize application execution time because maximizing memory throughput may be unfair to threads [44, 54]. However, we do not model and therefore hold these adversary effects against the baseline case. Therefore, our FRFCFS represents an optimized state-of-the-art. 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 since memory controllers can only choose among the transactions in their queue and do not control the order requests arrive in their queue. With CMS, neither complex reordering schemes nor deep queues are required to maximize memory throughput since the memory is always accessed sequentially.

To set up this experiment, we use DRAMsim2 [60] 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. 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. Thus the uncoordinated/random-order traces have two adjacent loads or stores for consecutive addresses. 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), 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. Latency reduction is crucial if DRAM latency is the performance bottleneck [48, 66].

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 [58, 69, 71, 5, 66]. 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 [69]. 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 [69]. 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 10 (left) shows execution times for completing a single read or write CMS operation. In the baseline case, processors send memory requests without coordination. 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 access patterns, created by the baseline case because each processor requests data independently of other processors.
- 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. This additional benefit of reads compared to writes provides them with a speedup comparable to writes even though memory write performance gains are smaller.

For both reads and writes, CMS load balances the network better because it accesses tile lines in an interleaved manner, whereas in the baseline case hotspots can be created depending on the order requests arrive to the memory controller. For example, processors closer to the memory controller are more likely to have their requests satisfied first, but that creates a larger number of replies to the same processor, resulting in transient load imbalance and fairness concerns.

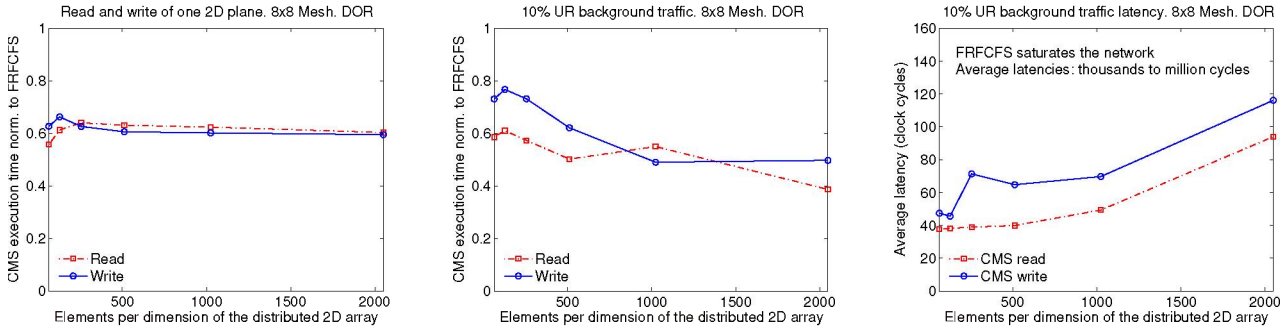

Figure 10: CMS read and write operation completion time normalized to the baseline, and the impact on background traffic.

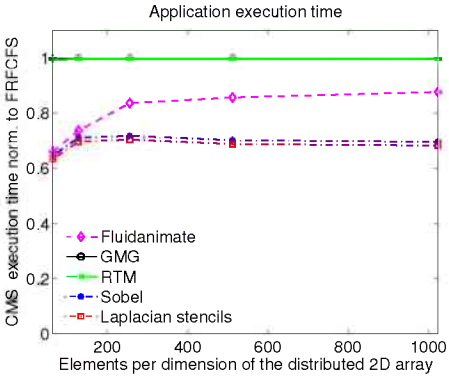

Figure 11: Application speedup for CMS.

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 10 (center), the reduction in execution time for CMS in the mesh is 46% for reads and 36% for writes. While background traffic degrades performance for CMS, it is more adversary to the baseline because read and write requests are queued in the network. Figure 10 (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 because that many requests do not fit in the memory controller's queue. Therefore, they create multi-hop paths of congested packets which degrade other traffic's performance [47].
- 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 11. 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.

CMS has broad applicability because a wide variety of stencil-based kernels are memory bound [65, 70, 32, 36]. Stencil-based kernels are also critical because they comprise the building blocks of applications ranging from image processing in consumer devices to the largest scale HPC applications such as climate modeling and fluid simulations. Media applications can require 300GB/s of memory bandwidth for 48 processings [57]. Other kernels are more memory bandwidth bound, even for just a few processors, and remain so even after a variety of software optimizations, autotuning, and autoparallelization [36, 18]. In addition, numerous image processing applications are also memory bandwidth bound, such as the matrix-vector product in NVIDIA Fermi [1] and face recognition [38]. High graphics processing demands force the Xbox 360 to have 22.4GB/s of GDDR3 bandwidth to satisfy just three processors [70] Finally, CMS reduces memory access latency, which directly benefits latency-sensitive systems [66].

To make matters worse, the number of memory bandwidth-bound applications and systems is expected to increase due to increased application demands and because computational throughput increases faster than memory bandwidth [12]. For example, current projections state that chip pins increase by 10% every year whereas on-chip processors double every 18 months [59]. This way, modern CMPs may require 90% of their area being caches to keep the memory from becoming a bottleneck [59]. Because the performance of a memory-bound application is roughly proportional to the rate at which its memory requests are served [65], techniques to increase memory bandwidth directly impact application execution time [70].

4.2.4. Impact of Multicast: Enabling multicast support increases the benefits for CMS read operations from a 39% operation completion time reduction to 43% with no back-

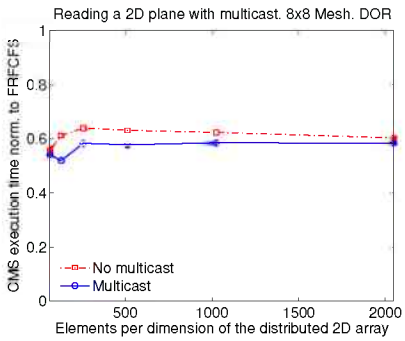

Figure 12: Results for enabling multicast in CMS reads.

ground traffic, as shown in Figure 12. Multicast with CMS reduces congestion as well as the amount of bits propagating through the network because data destined to two processors (the owner and the reader) propagates as a single packet until the point that the paths diverge. Since the owner and reader processors are neighbors in stencil computations, the majority of multicast packets traverse most of their paths as a single packet before a copy is produced. The percentage gains depend on the ratio of ghost zone size and tile size. This reduces the benefits for multicast as the size of the HTA increases in our experiments, since the size of the ghost zones remains constant because our 5-point stencil remains the same.

4.2.5. 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 [14]. 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.

4.2.6. 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 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 16×128 bits each, for a total of 2KB. Eight transactions are more than required to keep the memory busy in the write operations of our evaluations, as discussed in Section 3.5. 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. This is due to the extra complexity of the CMS engine as reflected by the

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

count of logic LUTs, as well as the reorder buffers and the small read buffers. Also due to the buffers, the CMS engine occupies more area. 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. This will delay initialization of operations by a cycle, but will enable the CMS engine to operate at comparable clock frequencies as the DMA engine. 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 [58] 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 [76, 58]. In addition, the reorder buffer for write operations in the CMS engine is much smaller compared to transaction queues used in modern memory controllers; this more than compensates for the additional area of the CMS engine. 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× smaller, and requires up to 1.5×–4× less access energy compared to a transaction queue of 32 entries which is typical for a modern memory controller transaction queue TCAM [4, 58]. CMS also replaces DMA engines in *each* processor.

Using FIFO memory schedulers and removing DMA engines is only prudent in systems that predominantly use collective data transfers, because otherwise the impact of non-collective transfers becomes considerable. However, such bulk-synchronous SPMD systems and other systems that rely on collective transfers comprise a wide variety of architectures and applications, such as STI Cell, GPUs, image processing, face recognition, combustion simulation, fluid modelling, wave equation processing, and Laplacian kernels. Moreover, 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, similar to any other

accelerator.

CMS also reduces DRAM access power because the baseline case consumes 2.2× 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 [64]. These projections, combined with attributing 25%–40% of total datacenter power to the DRAM system [69], make CMS's DRAM power reductions critical.

5. Discussion

CMS is targeted at bulk-synchronous SPMD execution models that transfer distributed 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 are in fact these kinds of SPMD kernels that are seen in image processing, face recognition, machine learning, fluid dynamics, linear algebra, kinetics simulation, and numerous others. Even though mappings alternative to row- or column-major can produce a more favorable memory access stream for some applications without CMS [27, 55, 32, 67], the baseline still cannot outperform CMS even with complex data layout transformations. CMS will also benefit communication-avoiding optimizations which tradeoff reduced memory traffic for redundant computation [51, 28, 18]. CMS can increase performance without the extra local storage, cache space, or computations needed for redundant communication, while better alleviating network congestion and reducing memory power.

CMS also 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]. In such architectures, GPU programming language constructs can identify collective transfers by being aware of the data layout. Furthermore, while in local-store architectures such as STI Cell [62] 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. Alternatively, compilers can also recognize collective transfer opportunities abstractly from the programmer.

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 [15]. 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 [43]. 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 [63].

Classic vector machines such as the Cray-1 [61, 50] 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 [24]. VIRAM [45] can also exploit data-level parallelism at chip level to overcome the wiring costs of massive bank-switching [45, 75], but the memory capacities offered by the various Processor-in-Memory approaches have proven to be impractically small for the commercial market. Moreover, variations of DMA engines, such as scatter-gather DMA [40], still perform transfers between only two components, thus creating out of order access streams to memory.

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 [77]. However, with Impulse the data arrays remain scattered in the DRAM, thereby leading to inefficient DRAM performance due to overfetch. By contrast, CMS schedules what would otherwise be non-contiguous memory accesses so that they are presented as a linear stream of addresses to DRAM. Therefore, the impulse memory controller improves efficiency of the cache hierarchy, but it offers no benefit for memory bandwidth.

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 [58, 31, 65, 44, 19, 54, 74]. Memory controllers can be thread-aware by trying to serve requests at a thread granularity in order to reduce thread stall time. Many schedulers, such as PAR-BS, perform limited reordering by attempting to exploit row buffer locality and bank parallelism among other metrics [54]. 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 [57]. 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 reorder requests [76]. However, because decisions are made with local knowledge and processors still issue requests independently, this scheme performs slightly lower than a FRFCFS scheduler. Such schemes that rely exclusively on the network cannot perfectly reconstruct memory order without blocking packets in the network, affecting other traffic. Alternative work uses admission control to inject only requests for open DRAM rows [49]. 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 [66]. 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 [69].

Past work has repeatedly reported that a wide variety of applications are constrained by memory bandwidth [65, 28, 64, 59, 36, 18, 46, 12, 32, 55]. 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 [18, 28]. Even with a write-back policy, caches are constrained by their size and the unpredictability of the incoming packers, similar to a memory controller. Last-level caches can, to some degree, eliminate redundant memory reads due to ghost zones, but this highly depends on timing and cache size to determine when data is evicted. 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 [37].

CMS does not resolve cache interference across tiles (such as for ghost zones) during a computation iteration. This can be accomplished by hybrid software–hardware cache coherence [16, 42]. CMS focuses on collective transfers to fetch new data or record old data back to memory. We assume tiles optimized for local caches, with no data reuse across loops.

Polyhedral representations alternative to HTAs are also applicable to CMS [41, 13, 7]. 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× 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

Copyright Notice

References

[1] A. Abdelfattah, J. Dongarra, D. Keyes, and H. Ltaief, "Optimizing memory-bound numerical kernels on GPU hardware accelerators," July 2012.

[2] D. Abts, N. D. Enright Jerger, J. Kim, D. Gibson, and M. H. Lipasti, "Achieving predictable performance through better memory controller placement in many-core cmps," in *Proceedings of the 36th annual international symposium on Computer architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 451–461.

[3] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger, "Clock rate versus IPC: the end of the road for conventional microarchitectures," in *Proceedings of the 27th annual international symposium on Computer architecture*, ser. ISCA '00, 2000, pp. 248–259.

[4] B. Agrawal and T. Sherwood, "Ternary CAM power and delay model: Extensions and uses," *IEEE Transactions on VLSI*, vol. 16, no. 5, pp. 554 –564, 2008.

[5] J. H. Ahn, N. P. Jouppi, C. Kozyrakis, J. Leverich, and R. S. Schreiber, "Future scaling of processor-memory interfaces," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09, 2009, pp. 42:1–42:12.

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

[7] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '08. New York, NY, USA: ACM, 2008, pp. 101–113.

[8] S. Borkar, "How to stop interconnects from hindering the future of computing x0021;," in *Optical Interconnects Conference, 2013 IEEE*, 2013, pp. 96–97.

[9] ——, "Thousand core chips: a technology perspective," in *Proceedings of the 44th annual Design Automation Conference*, ser. DAC '07, 2007, pp. 746–749.

[10] S. Borkar and A. A. Chien, "The future of microprocessors," *Communications of the ACM*, vol. 54, no. 5, pp. 67–77, 2011.

[11] R. Chamberlain, "Solving partial differential equations on a parallel supercomputer," in *IEEE Colloquium on Parallel Processing: Industrial and scientific applications*, 1991.

[12] C. Chan, D. Unat, M. Lijewski, W. Zhang, J. Bell, and J. Shalf, "Software design space exploration for exascale combustion co-design," in *International Supercomputing Conference*, 2013.

[13] R. Chandra, D.-K. Chen, R. Cox, D. E. Maydan, N. Nedeljkovic, and J. M. Anderson, "Data distribution support on distributed shared memory multiprocessors," in *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, ser. PLDI '97, 1997, pp. 334–345.

[14] 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*, vol. 2, no. 1, p. 015001, 2009.

[15] 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*, vol. 22, no. 10, pp. 2004–2018, 2006.

[16] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou, "DeNovo: Rethinking the memory hierarchy for disciplined parallelism," in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT, 2011, pp. 155–166.

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

[18] 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*, ser. SC, 2008, pp. 4:1–4:12.

[19] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt, "Parallel application memory scheduling," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO, 2011, pp. 362–373.

[20] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th annual international symposium on Computer architecture*, ser. ISCA '11, 2011, pp. 365–376.

[21] P. K. et al., "Exascale computing study: Technology challenges in achieving exascale systems," http://users.ece.gatech.edu/~mrichard/ExascaleComputingStudyReports/exascale_final_report_100208.pdf, 2008.

[22] B. B. Fraguela, J. Guo, G. Bikshandi, M. J. Garzarán, G. Almási, J. Moreira, and D. Padua, "The hierarchically tiled arrays programming approach," in *Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems*, ser. LCR '04, 2004, pp. 1–12.

[23] 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*, vol. 5, 2010, pp. 67–71.

[24] 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*, ser. ARCS '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 146–158.

[25] 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*, ser. ISQED '10, 2010, pp. 362–371.

[26] J. Guo, G. Bikshandi, B. B. Fraguela, and D. Padua, "Writing productive stencil codes with overlapped tiling," *Journal on Concurrency and Computation: Practice and Experience*, vol. 21, no. 1, pp. 25–39, 2009.

[27] 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*, ser. CC'11/ETAPS'11, 2011, pp. 225–245.

[28] 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*, ser. ICS, 2012, pp. 311–320.

[29] M. Horowitz and W. Dally, "How scaling will change processor architecture," in *Solid-State Circuits Conference, 2004. Digest of Technical Papers. ISSCC. 2004 IEEE International*, 2004, pp. 132 – 133.

[30] J. Huh, D. Burger, and S. W. Keckler, "Exploring the design space of future CMPs," in *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT, 2001, pp. 199–210.

[31] Y. Ishii, M. Inaba, and K. Hiraki, "Unified memory optimizing architecture: memory subsystem control with a unified predictor," in *Proceedings of the 26th ACM international conference on Supercomputing*, ser. ICS, 2012, pp. 267–278.

[32] J. Jaeger and D. Barthou, "Automatic efficient data layout for multithreaded stencil codes on CPUs and GPUs," *19th International Conference on High Performance Computing*, pp. 1–10, 2012.

[33] N. E. Jerger, L.-S. Peh, and M. Lipasti, "Virtual circuit tree multicasting: A case for on-chip hardware multicast support," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA '08, 2008, pp. 229–240.

[34] 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*, ser. ISPASS '13, 2013, pp. 86–96.

[35] 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*, 2010, pp. 1–12.

[36] 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.

[37] 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, pp. 773 –778.

[38] A. Kapoor, S. Baker, S. Basu, and E. Horvitz, "Memory constrained face recognition," in *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, 2012, pp. 2539–2546.

[39] 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*, ser. MICRO-44 '11, 2011, pp. 24–35.

[40] H. Kavianipour and C. Bohm, "High performance FPGA-based scatter/gather DMA interface for PCIe," in *Proceedings of the IEEE Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC)*, 2012, pp. 1517–1520.

[41] 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*, 2001, pp. 47–54.

[42] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta, and S. J. Patel, "Cohesion: a hybrid memory model for accelerators," in *Proceedings of the 37th annual international symposium on Computer architecture*, ser. ISCA, 2010, pp. 429–440.

[43] 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*, ser. IPDPS '08, 2008, pp. 1 –12.

[44] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO, 2010, pp. 65 –76.

[45] C. Kozyrakis and D. Patterson, "Scalable, vector processors for embedded systems," *IEEE Micro*, vol. 23, no. 6, pp. 36–45, 2003.

[46] 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*, ser. SC '11, 2011, pp. 73:1–73:12.

[47] C. P. Kruskal and M. Snir, "The performance of multistage interconnection networks for multiprocessors," *IEEE Transanctions on Computers*, pp. 1091–1098, December 1983.

[48] D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu, "Tiered-latency DRAM: A low latency and low cost DRAM architecture," in *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture*, ser. HPCA '13, 2013, pp. 615–626.

[49] 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*, ser. ISLPED, 2008, pp. 171–176.

[50] B. K. Mathew, S. A. McKee, J. B. Carter, and A. Davis, "Algorithmic foundations for a parallel vector access memory system," in *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, ser. SPAA '00. New York, NY, USA: ACM, 2000, pp. 156–165.

[51] 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*, ser. ICS '09, 2009, pp. 256–265.

[52] 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*, ser. SC, 2009.

[53] M. Müller, D. Charypar, and M. Gross, "Particle-based fluid simulation for interactive applications," in *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, ser. SCA '03, 2003, pp. 154–159.

[54] 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*, ser. ISCA '08, 2008, pp. 63–74.

[55] L. Peng, R. Seymour, K.-i. Nomura, R. K. Kalia, A. Nakano, P. Vashishta, A. Loddoch, M. Netzband, W. Volz, and C. Wong, "High-order stencil computations on multicore clusters," in *IEEE International Symposium on Parallel Distributed Processing*, ser. IPDPS '09, 2009, pp. 1–11.

[56] S. M. F. Rahman, Q. Yi, and A. Qasem, "Understanding stencil code performance on multicore architectures," in *Proceedings of the 8th ACM International Conference on Computing Frontiers*, ser. CF '11, 2011, pp. 30:1–30:10.

[57] S. Rixner, "A bandwidth-efficient architecture for a streaming media processor," Ph.D. dissertation, 2001, aAI0803043.

[58] 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*, ser. ISCA '00, 2000, pp. 128–138.

[59] 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*, ser. ISCA, 2009, pp. 371–382.

[60] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A cycle accurate memory system simulator," *IEEE Computer Architecture Letters*, vol. 10, no. 1, pp. 16 –19, 2011.

[61] R. M. Russell, "The CRAY-1 computer system," *Communications of the ACM*, vol. 21, no. 1, pp. 63–72, 1978.

[62] S. Schneider, J.-S. Yeom, and D. S. Nikolopoulos, "Programming multiprocessors with explicitly managed memory hierarchies," *IEEE Computer*, vol. 42, no. 12, pp. 28–34, 2009.

[63] 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*, ser. SC '95, 1995.

[64] J. Shalf, S. S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," in *International Meeting on High Performance Computing for Computational Science*, ser. VECPAR '10, 2010.

[65] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu, "MISE: Providing performance predictability and improving fairness in shared main memory systems," in *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, ser. HPCA '13, 2013, pp. 639–650.

[66] 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*, ser. ASPLOS '10, 2010, pp. 219–230.

[67] I.-J. Sung, J. A. Stratton, and W.-M. W. Hwu, "Data layout transformation exploiting memory-level parallelism in structured grid many-core applications," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, ser. PACT '10. ACM, 2010, pp. 513–522.

[68] J. Torrellas, "How to build a useful thousand-core manycore system?" *International Parallel and Distributed Processing Symposium*, 2009.

[69] 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*, ser. ISCA '10, 2010, pp. 175–186.

[70] A. Vega, F. Cabarcas, A. Ramirez, and M. Valero, "Breaking the bandwidth wall in chip multiprocessors," in *International Conference on Embedded Computer Systems*, ser. SAMOS '11, 2011, pp. 255 –262.

[71] 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.

[72] X. Wang, M. Yang, Y. Jiang, and P. Liu, "On an efficient NoC multicasting scheme in support of multiple applications running on irregular sub-networks," *Microprocessors and Microsystems*, vol. 35, no. 2, pp. 119–129, 2011.

[73] 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*, 2012, pp. 1–11.

[74] 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*, ser. PACT '10, 2010, pp. 237–248.

[75] P. Yiannacouras, J. Steffan, and J. Rose, "Portable, flexible, and scalable soft vector processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 8, pp. 1429–1442, 2012.

[76] 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*, ser. MICRO, 2009, pp. 34–44.

[77] 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*, vol. 50, no. 11, pp. 1117–1132, 2001.

[78] Y. Zhang and F. Mueller, "Auto-generation and auto-tuning of 3D stencil codes on GPU clusters," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12, 2012, pp. 155–164.