CODA: Enabling Co-location of Computation and Data for Multiple GPU Systems

HYOJONG KIM, RAMYAD HADIDI, LIFENG NAI, and HYESOON KIM,
Georgia Institute of Technology
NUWAN JAYASENA, YASUKO ECKERT, ONUR KAYIRAN, and GABRIEL LOH,
Advanced Micro Devices, Inc.

To exploit parallelism and scalability of multiple GPUs in a system, it is critical to place compute and data together. However, two key techniques that have been used to hide memory latency and improve thread-level parallelism (TLP), memory interleaving, and thread block scheduling, in traditional GPU systems are at odds with efficient use of multiple GPUs. Distributing data across multiple GPUs to improve overall memory bandwidth utilization incurs high remote traffic when the data and compute are misaligned. Nondeterministic thread block scheduling to improve compute resource utilization impedes co-placement of compute and data. Our goal in this work is to enable co-placement of compute and data in the presence of fine-grained interleaved memory with a low-cost approach.

To this end, we propose a mechanism that identifies exclusively accessed data and place the data along with the thread block that accesses it in the same GPU. The key ideas are (1) the amount of data exclusively used by a thread block can be estimated, and that exclusive data (of any size) can be localized to one GPU with coarse-grained interleaved pages; (2) using the affinity-based thread block scheduling policy, we can co-place compute and data together; and (3) by using dual address mode with lightweight changes to virtual to physical page mappings, we can selectively choose different interleaved memory pages for each data structure. Our evaluations across a wide range of workloads show that the proposed mechanism improves performance by 31% and reduces 38% remote traffic over a baseline system.

CCS Concepts: • **Computer systems organization → Single instruction, multiple data**; **Distributed architectures**; • **Hardware → Emerging architectures**; **Memory and dense storage**; • **Software and its engineering → Compilers**;

Additional Key Words and Phrases: Multiple GPUs, hybrid data layout, compute and data localization, compiler technique, profiling

ACM Reference format:
Hyojong Kim, Ramyad Hadidi, Lifeng Nai, Hyesoon Kim, Nuwan Jayasena, Yasuko Eckert, Onur Kayiran, and Gabriel Loh. 2018. CODA: Enabling Co-location of Computation and Data for Multiple GPU Systems. *ACM Trans. Archit. Code Optim.* 15, 3, Article 32 (August 2018), 23 pages.
https://doi.org/10.1145/3232521

Authors' addresses: H. Kim; email: hyojong.kim@gatech.edu; R. Hadidi; email: rhadidi@gatech.edu; L. Nai; email: lnai@google.com; H. Kim, Georgia Institute of Technology; email: hyesoon.kim@gatech.edu; N. Jayasena; email: nuwan.jayasena@amd.com; Y. Eckert; email: yasuko.eckert@amd.com; O. Kayiran; email: onur.kayiran@amd.com; G. Loh, Advanced Micro Devices, Inc.; email: gabriel.loh@amd.com.

1544-3566/2018/08-ART32 $15.00
https://doi.org/10.1145/3232521

1 INTRODUCTION

In parallel programming models, such as the general-purpose graphics processing unit (GPGPU) programming model, the key to achieving high performance is to exploit thread-level parallelism (TLP). One way to accomplish this is to have each thread process a distinctive part of data such that the data process can be parallelized to the max. The effectiveness of this approach in achieving high-performance critically depends on whether the system can hide memory latency and exploit all available compute resources. To hide memory latency and better exploit compute resources, modern GPU systems take two orthogonal approaches: memory interleaving and thread block scheduling, respectively. Memory interleaving is a technique that stripes small chunks of the physical address space across different memory modules, thereby increasing memory bandwidth utilization. Thread block scheduling determines to which GPU core each thread block is scheduled. Dispatching a thread block to an available GPU core in a round-robin order would be the best way to provide load balancing, thereby increasing resource utilization. While these techniques have been effective in traditional GPU systems, in this article, we question their efficacy in systems with multiple GPUs, since they might disrupt co-locating code and data.

Suppose a system has four GPUs, each with its own memory. GPUs are connected with the processor-centric topology [25], constituting the GPU memory address space. While a GPU can transparently access data in other GPUs, such an access uses the low-bandwidth off-chip links and traverses the interconnect, incurring higher latency and leading to lower performance and energy efficiency. However, a local data access, which occurs when a GPU accesses data in its local memory, utilizes high memory bandwidth, incurring lower latency and leading to higher performance and energy efficiency. The GPU physical address space is interleaved at a fine granularity to help improve memory bandwidth utilization. Let us take the transpose computation shown in Figure 1 as an example to examine the impact that memory interleaving and thread block scheduling have on performance. As its name suggests, this kernel transposes the `in` array and saves the result in the `out` array. Each thread processes distinctive `nfeatures` elements (line 4) from `(pid` $\times$ `nfeatures)-th` element of the `in` array (line 5).

Figure 2 and Figure 3 represent two cases where code and data are misaligned due to memory interleaving and thread block scheduling. For both of them, we assume interleaving granularity of 256B, so four consecutive cache lines (cache line size is assumed to be 64B) are placed in a GPU, and the next consecutive four cache lines are placed in the next GPU. We also assume the fair-round-robin thread block scheduling policy. A thread block is color-coded based on the GPU it is scheduled to, and a cache line is also color-coded based on which thread block accesses it. For example, thread blocks 0 and 4 have the same color, and lines 0, 1, 8, and 9 have the same color as thread blocks 0 and 4.

Figure 2 depicts a case in which each thread block processes two cache lines worth of elements of the `in` array. Note that the number of elements that each thread block processes is determined based on `nfeatures`. Accesses to lines 0 and 1 from thread block 0 are local, and hence efficient, but accesses to lines 8 and 9 from thread block 4 are remote and hence inefficient. Fortunately, this misalignment can be easily solved by scheduling thread block 4 to GPU 2, where lines 8 and 9 are allocated. Figure 3 depicts a slightly different case in which each thread block processes three cache lines worth of elements of the `in` array. Now, it is not as simple as the case of Figure 2, since some of the accesses from a thread block are local and some are remote. Therefore, this misalignment cannot be solved just by scheduling a thread block to another GPU.

Our goal in this article is to reduce such code and data misalignment, thereby achieving better performance. First, to place code and the data that they access together, we identify which data (and which part of it) each thread block accesses. We make two observations. First, the amount of data

```
1  __global__ void transpose(float *in, float *out, int npoints, int
       nfeatures) {
2    int pid = blockDim.x * blockIdx.x + threadIdx.x;
3    if (pid < npoints) {
4      for (int i = 0; i < nfeatures; i++)
5        out[i * npoints + pid] = in[pid * nfeatures + i];
6    }
7  }
```

Fig. 1. Code snippet from K-means clustering.

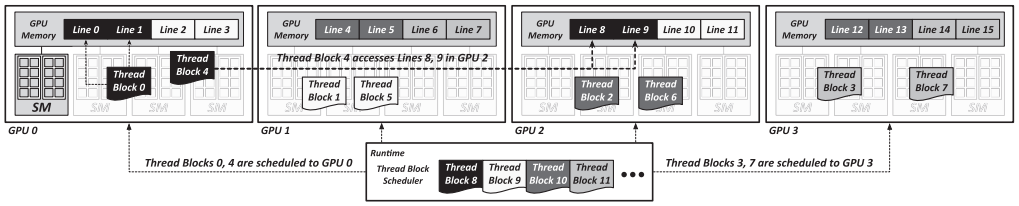


Fig. 2. A case where code and data misalignment can be solved with thread block scheduling. Cache lines 8 and 9 are placed in GPU 2 due to memory interleaving. Thread block 4, which accesses them, is scheduled to GPU 0. This misalignment can be easily solved by scheduling thread block 4 to GPU 2.

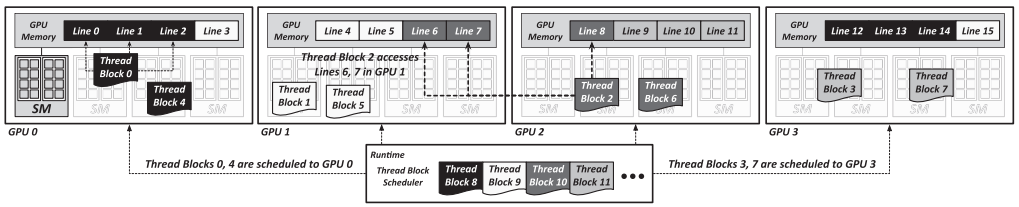


Fig. 3. A case where code and data misalignment cannot be solved with just thread block scheduling. Cache lines 6 and 7 are placed in GPU 1, and cache line 8 is placed in GPU 2 due to memory interleaving. Thread block 2, which accesses them, is scheduled to GPU 2. Scheduling thread block 2 to GPU 1 makes accesses to cache line 8 inefficient.

used by one thread block is often determined by the number of threads in a thread block and the amount of data each thread accesses. The latter can be estimated by either compile-time analyses (for input-independent access patterns) or profiler-assisted techniques (for input-dependent access patterns). Second, although the number of threads in a thread block is often input dependent, it is determined before kernel invocation (specifically, even before data structures are allocated). With these observations combined, we come to the conclusion that the amount of data used by one thread block *can* be estimated. For these reasons, we utilize a compiler-based and profiler-assisted technique to analyze the access pattern for each data structure and determine how each should be layered across GPUs.

Second, to place all the data that a thread block accesses in the same GPU as the thread block even in the presence of fine-grained memory interleaving, we make a slight change in hardware and the operating system (OS) to realize coarser-grained (the OS page size) memory interleaving in addition to the fine-grained (256B) memory interleaving. The key idea is to use different sets of address mapping bits for each memory page depending on its anticipated access pattern, allowing the two sets of mappings to co-exist; low-order bits are used to distribute data across GPUs,

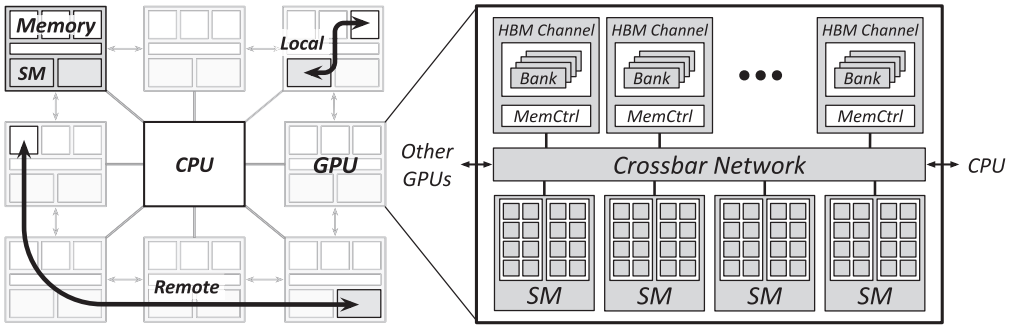


Fig. 4. Overview of a system with multiple GPUs.

whereas high order-bits are used to place an entire page in a single GPU. The granularity information for each memory page is stored in the page table entry (PTE) and translation lookaside buffer (TLB) entry. At the time a virtual address is translated into a physical address and the memory request is sent, our mechanism uses the appropriate address mapping depending on the granularity information. Admittedly, the concept of changing address mapping to change data layout or to increase memory-level parallelism is not new [14, 59]. However, our proposed mechanism is different from previous proposals in that it enables the coexistence of pages with different address mappings while not requiring large-scale page migrations.

Third, to ensure that the code is scheduled to the GPU where the data it accesses is located, we use an affinity-based scheduling mechanism. In traditional GPUs, thread blocks are scheduled to any GPUs (and any SMs in the GPU) in the system in a nondeterministic fashion. To steer a thread block and the data it accesses to the same GPU, we set an affinity between thread blocks and GPUs, and use the information for scheduling.

Our article makes the following contributions:

- We observe that code and data alignment is critical in achieving high performance in a system with multiple GPUs, and traditional memory interleaving and thread block scheduling are at odds with efficient use of multiple GPUs.
- We propose a mechanism that utilizes a compiler-based and profiler-assisted technique to decide whether to localize or distribute each data structure based on its anticipated access pattern.
- We design a lightweight hardware mechanism that supports dual-mode address mapping at a page granularity, so that a page can be either spread across GPUs or localized to a single GPU. This mechanism enables pages with different address mappings to coexist in the same memory space and the amount of each mode can be adjusted at runtime.
- We evaluate our proposed mechanism with a wide range of data-intensive workloads and show that it improves performance by 31% and reduces 38% of remote data accesses over a baseline system that does not have dual-mode address mapping or an affinity-based computation and data co-placement mechanism.

2 BACKGROUND

2.1 Baseline Architecture

Figure 4 shows a high-level diagram of a system with multiple GPUs and the details of a GPU. In this work, we assume that every GPU in the system can communicate with each other. While this assumption may not entirely hold true in modern multi-GPU systems, where not all GPUs can

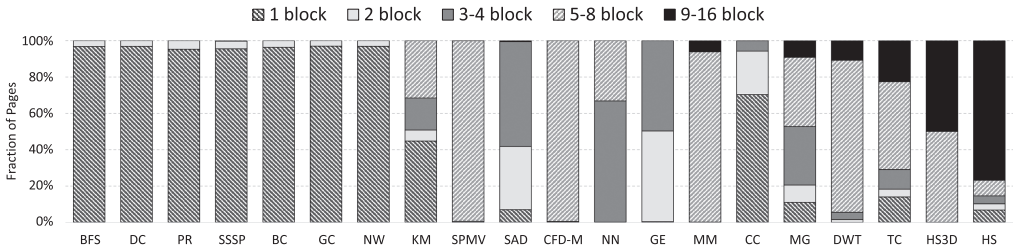


Fig. 5. Distribution of memory pages according to the number of thread-blocks that access each page.

access each other, and peer access is only enabled between parts of them based on the topology, with this assumption, the proposed mechanism can be applicable to not only future multi-GPU systems that feature all-to-all communication, but also Non-Uniform Memory Architectures (NUMA) systems (irrespective of processor types) and multi-chip-module GPU systems [4]. We choose high bandwidth memory (HBM) as our baseline [5] although our mechanism does not rely on any particular memory organization. HBM is composed of multiple memory channels and uses a wide-lane bus interface to achieve high memory bandwidth and low power dissipation. Each GPU has streaming multiprocessors (SMs) and high-speed off-chip links for remote data accesses—to/from other GPUs and the CPU—and a crossbar network that connects SMs and HBM. We assume each GPU is equipped with per-SM hardware TLBs and a highly threaded shared memory management unit (MMU) that accesses page tables and is capable of performing virtual address translation [47, 48].

We assume the single-instruction multiple-thread (SIMT) execution model for our multiple GPU system [13, 32, 39]. The CPU launches GPU kernels, and the runtime system partitions and distributes thread blocks across GPUs (and the SMs in the GPU) in the system. Up to the number of SMs × the number of thread blocks per SM are concurrently executed in each GPU. There are two kinds of networks in our system: (1) a network among GPUs (denoted as Remote in Figure 4), and (2) a network that connects SMs in a GPU to their local memory (denoted as Local in Figure 4). Such multiple GPUs with remote and local memory accesses are common in near-data processing (NDP) [22, 57, 58].

2.2 Address Interleaving

To increase memory-level parallelism, or to reduce channel/rank/bank conflicts, fine-grained memory interleaving is typically used in modern memory systems by striping small chunks of the physical address space (often the size of a few cache lines) across different banks, ranks, and channels. In a system with multiple GPUs, a page can be striped across multiple GPUs with fine-grained memory interleaving, or the entire page can be allocated in a single GPU with coarse-grained memory interleaving. Complex address decoding schemes have been studied before [46, 59]; however, for brevity, we assume a simple address mapping scheme. We discuss the applicability of our mechanism in systems with complex address mapping schemes in Section 7.1.

3 MOTIVATION

Figure 5 shows distribution of memory pages according to the number of thread blocks that access each memory page for various data-intensive workloads from publicly available GPU benchmark suites [9, 37, 54]. It is observed that for some workloads, such as BFS, DC, PR, SSSP, BC, GC, and NW, most pages are accessed by only one or two thread blocks. In traditional GPU systems, which have one GPU and its local memory, distributing pages irrespective of which and how many thread blocks access them helps improve the utilization of memory interfaces by distributing the memory

traffic. However, in a system with multiple GPUs, where there is a big discrepancy between local memory and remote GPU memory, distributing such pages across GPUs incurs lots of remote traffic. Therefore, it is imperative to place such pages (exclusively used data) and the thread blocks (computations) that access them in individual GPUs. In contrast, in the case of HS3D and HS, most pages are accessed by almost all thread blocks. Even in the presence of multiple GPUs, it is better to distribute such pages (shared data) across GPUs to reduce memory bandwidth contention.

From this, we make two observations. First, some pages are accessed exclusively by a few thread blocks, while other pages are accessed, or shared, by many thread blocks. The exclusively used pages should be placed in individual GPUs with the thread blocks that access them to eliminate remote traffic, and the shared pages should be distributed across GPUs to reduce memory bandwidth contention. Second, each application has different distribution of exclusive and shared pages. For example, most pages in BFS are exclusively used, so the memory system should be capable of localizing all of them. However, most pages in HS are shared, so the memory system should also be capable of distributing all of them. These observations motivate the need for a mechanism that can allocate localized pages versus distributed pages *flexibly* based on an application's needs.

4 MECHANISM

In this section, we describe our mechanism that enables co-location of compute and data in a system with multiple GPUs. Section 4.1 demonstrates how our mechanism can improve the case where code and data misalignment cannot be easily solved with just thread block scheduling, as shown in Figure 3. Section 4.2 describes a mechanism that either distributes data across GPUs or localizes data to a single GPU at a page granularity. Section 4.3 describes a mechanism that utilizes a compiler-based and profiler-assisted technique to decide whether to localize or distribute each memory page based on its anticipated access pattern and introduces an affinity-based scheduling algorithm that steers thread blocks to the GPU where the data they access is located.

4.1 Demonstration

Figure 3 in Section 1 depicts a case where each thread block accesses three consecutive cache lines, and thread blocks are scheduled with the fair-round-robin scheduling policy. In that example, due to the misalignment, just by improving the thread block scheduler cannot eliminate remote accesses. For example, accesses to cache lines 6 and 7 from thread block 2 are remote, and hence inefficient, whereas accesses to cache line 8 is local, and hence efficient. Scheduling thread block 2 to GPU 1 converts accesses to cache lines 6 and 7 to local, but the previous local access (access to cache line 8) becomes remote.

Figure 6 demonstrates how our mechanism solves this misalignment. First, our mechanism identifies which cache lines are accessed by which thread blocks (Section 4.3). Second, based on the identification and analyses, it decides whether to distribute the data across GPUs with fine-grained memory interleaving or allocate them in a single GPU with coarse-grained memory interleaving. Our mechanism enables selective coarse-grained allocation on top of fine-grained interleaved memory (Section 4.2).

4.2 Dual-Mode Address Mapping

Hardware Support. To localize exclusively used pages in the presence of fine-grained memory interleaving, we use different sets of bits for address mapping for each memory page depending on the anticipated access patterns, allowing the two sets of mappings to co-exist. The default (fine-grained) address mapping distributes a page across GPUs (as is done today), and the alternative (coarse-grained) address mapping allocates (or localizes) an entire page in a single GPU (as is desirable for exclusively used data). We refer to the distributed page as **FGP** (fine-grained interleaved

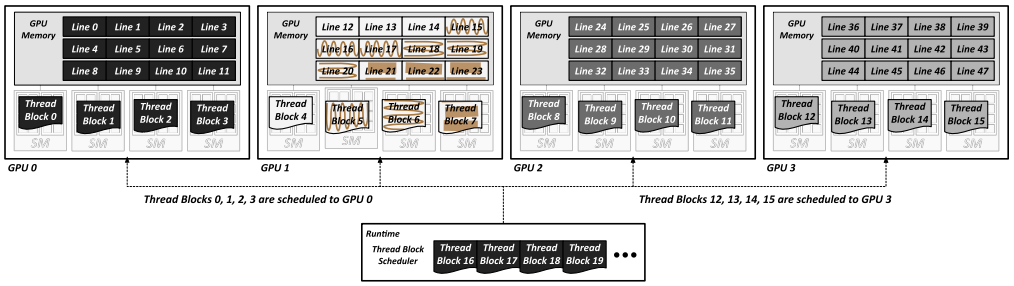


Fig. 6. Representation of what our mechanism can do in the case where code and data misalignment cannot be easily solved with just thread block scheduling. It allocates consecutive cache lines 0-11 in GPU 0 and schedules thread blocks 0-3 to GPU 0 so that all the accesses to these cache lines will be efficient. Cache lines 12-23 are marked to represent which thread blocks access them.

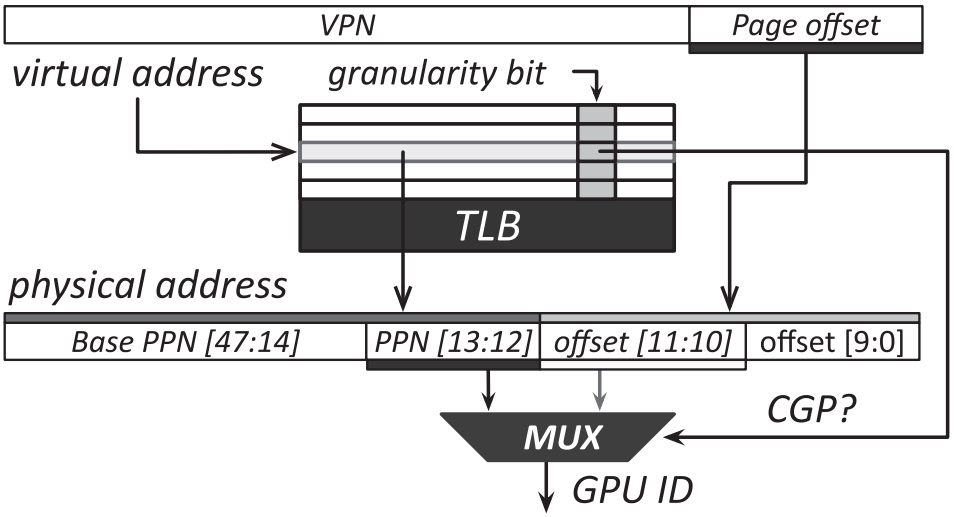
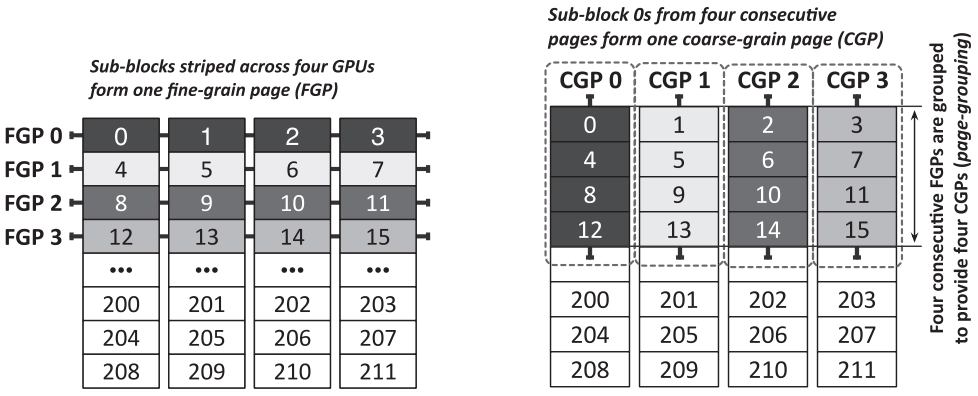


Fig. 7. Hardware for a dual-mode address mapping.

page) and the localized page as **CGP** (coarse-grained interleaved page). FGP is better suited for the data that are shared among (or accessed by) multiple GPUs. However, CGP is better suited for the data that are exclusively accessed by a single GPU. Note that once hardware provides the ability to map an entire page to a GPU (as is enabled by our selective use of coarse-grained address mapping), the OS could allocate arbitrarily large objects within a GPU by mapping all the virtual pages of that object to the physical pages (CGPs) in the GPU.

PTEs, TLB entries, and cache lines are extended to indicate the granularity information, fine-grained or coarse-grained, for each page, as shown in Figure 7. The granularity bit in a PTE is set by the OS when a CGP is allocated, and the granularity bit in a cache line is set when the cache line is allocated. When the granularity bit is set, indicating CGP, the lowest bits from the Physical Page Number (PPN) are used to index GPU, whereas the highest bits from the page offset are used for FGPs. For example, in a system with four GPUs, when a cache line is evicted from the last level cache (LLC), a write-back request is sent to the memory indexed by either the bits [13:12] when the granularity bit is set (for CGPs) or the bits [11:10] when the granularity bit is not set (for FGPs). Be assured that we only change the mapping of the physical address to memory and not the



(a) Fine-grain page layout (b) Coarse-grain page layout

Fig. 8. Conceptual diagram of page-group. The number indicates a sub-block address and the sub-blocks of the same color belong to the same OS page.

physical address itself. Thus, cache is accessed with the original physical address, irrespective of the granularity information, and our mechanism does not have any impact on the cache coherence protocol or virtual address translation.

System Software Support. The OS should be aware of the dual-mode address mapping (1) to indicate the granularity information in the PTEs and TLB entries, and (2) for page management, such as free page management or page replacement. It is important to note that it requires a set of adjacent FGPs to allocate a CGP (technically, a set of CGPs are allocated together). Consider a system where an FGP spans N consecutive GPUs, occupying a contiguous block of M bytes in each GPU memory. In that system, a CGP occupies $N \times M$ contiguous bytes within a single GPU memory. Therefore, a single CGP occupies the space that would have been utilized by N different FGPs within one GPU memory (but does not utilize any of the space those N FGPs would have occupied in other GPU memories). As a result, each block of N contiguous pages must uniformly be configured as FGP or CGP to avoid data layout conflicts. However, different blocks of N pages may be independently configured as FGP or CGP based on application or OS requirements.

For example, when FGP 0, in Figure 8(a), consisting of sub-blocks 0, 1, 2, and 3, is converted to a CGP, there are conflicts with sub-blocks 4, 8, and 12 from the three subsequent FGPs (each from FGP 1, FGP 2, and FGP 3, respectively). Therefore, those four FGPs must be converted to CGPs together, as shown in Figure 8(b). We use the term *page-group* to refer to a set of pages that must be converted together. Hence, the OS should decide between FGP and CGP at a page-group granularity and can switch between FGP and CGP only when all the pages in the page-group are free.

4.3 Compute-Data Co-location Algorithm

In traditional GPUs, thread blocks can be scheduled in any order, as they are supposed to run concurrently. The number of thread blocks that can run together in one SM is determined by thread block resource constraints. Normally, thread blocks are scheduled in order and as soon as one thread block retires, next thread block is scheduled to any available SM in any available GPU. However, to benefit from careful data placement, as is enabled by our dual-mode address mapping mechanism, thread blocks and the data they access must be co-located in the same GPU. To steer thread blocks and the data they access to the same GPU, we set an *affinity* between thread blocks and GPUs.

4.3.1 Affinity-based Work Scheduling Algorithm. We compute which GPU each thread block has affinity to using the following equation:

$$\text{affinity} = \left(\frac{\text{block\_id}}{N_{\text{blocks\_per\_GPU}}} \right) \bmod N_{\text{GPUs}}, \tag{1}$$

where block_id is flattened for multi-dimensional data based on row-major ordering (i.e., `blockIdx.y` × `blockDim.x` + `blockIdx.x`). $N_{\text{blocks\_per\_GPU}}$ is the number of thread blocks that can run concurrently in one GPU. For example, if one GPU has four SMs and each of which can run six thread blocks, then $N_{\text{blocks\_per\_GPU}}$ is 24. When N is the number of GPUs and T is the total number of thread blocks, T/N thread blocks have the same affinity. With this affinity information, whenever an SM is available, instead of assigning any unscheduled thread block to it, the scheduler picks one that has affinity to that GPU.[1] This may potentially lead to load imbalance compared to the baseline of assigning any available thread block to any SM in the system. However, the number of thread blocks typically being much greater than the number of GPUs reduces the likelihood of load imbalance.

The hardware and runtime system must be extended to support this modified scheduling scheme. The scheduling algorithm could be optimized further to select thread blocks from other GPUs when a GPU does not have any work left to do, similarly to the work-stealing algorithm. However, in our 20 evaluated benchmarks, only one suffered performance degradation due to the affinity-based scheduling algorithm. Therefore, we did not implement the work-stealing optimization.

4.3.2 Data Placement Algorithm. While the dual-mode address mapping enables the ability to localize an entire page in a single GPU, the question of how to identify the exclusively accessed or shared pages remains. This identification is particularly difficult for GPU systems, because data structures are allocated by the CPU before kernel invocations and are used by all threads in the kernel later.[2] To this end, we propose a compiler-based and profiler-assisted technique that identifies the amount of data used by one thread block for each data structure and decides which address mapping is desirable for the data structure (technically, for the pages in which the data structure is allocated). It is based on the following four observations. First, the amount of data used by one thread block is often determined by the number of threads in a thread block and the size of data structure that each thread accesses. Second, compile-time (symbolic) analyses can be used to detect if there exists a regular access pattern for each data structure. Third, profiler-assisted techniques can be used to estimate input-dependent accesses (more on this is explained later). Fourth, although the number of threads in a thread block is often input dependent, it is determined before a kernel invocation (generally, even before data structures are allocated).

Based on these observations, we implement a compile-time analysis on LLVM infrastructure [29]. We extend the FunctionPass, which enables traversing all the kernel functions at compile time, and perform the symbolic analysis. For all the memory accesses inside kernels, we analyze the "GetElementPtrInst" LLVM instruction, which performs the index computation. Based on the index expression and the types of variables it uses, we examine if there exists a runtime-constant stride (RCS) between two consecutive thread blocks. In this examination, we check if an expression uses only the (1) kernel-invocation-constants, such as parameters, block/grid dimensions, or global constants, which are determined before kernel invocation and remain constant throughout

[1]This scheduling algorithm is conceptually similar to the guided scheduling policy in OpenMP, where the programmer specifies chunk size (the number of loop iterations that one thread executes).

[2]We only discuss global data structures, which may be accessed by all the threads in the system, since local data structures are easily identifiable with specific keywords.

the kernel execution, (2) thread index, thread block index, and/or loop index (for local loops in the kernel). If such a stride is found, then we insert instructions in the CPU code to compute the stride distance between two consecutive thread blocks at runtime. We use profiler-assisted techniques for the case where the access pattern is input dependent *and* only when the input is not changed frequently (e.g., graph computing workloads). Note that the profiler performs a similar examination as the compile-time analysis. Our mechanism also uses FGPs for irregularly accessed data, shared data, or parameter objects, as they are accessed by many thread blocks.

Algorithm 1 shows the compile-time analysis algorithm. We use the definition-use (DU) chains to trace back to the definition of all the source operands of each GetElementPtrInst LLVM instruction. To simply handle control flows within the IR, we consider only the initial value that reaches a PHI node, which is used by the LLVM IR to represent an static single assignment (SSA) form such that every use has exactly one reaching definition. That is, among the incoming values, we only consider the instruction that dominates the PHI node. For this purpose, we use the "DominatorTreeWrapperPass" analysis of LLVM. Here, for brevity, we only consider the case where index does not use value loaded from previous memory instruction. Also, we rule out the case where the RCS expression cannot be algebraically simplified to constants during compile-time. To handle these cases, we postpone the decision to runtime (before the kernel is launched) and instrument the code to compute the RCS for some random thread blocks. We take advantage of the fact that an accurate analysis is not necessary, since it is just used to decide the memory layout, not impacting correctness.

ALGORITHM 1: Compile-time Runtime-Constant Stride (RCS) Analysis

Input: LLVM IR representation of GPU compute kernels

for each array index instruction: GetElementPtrInst **do**
  for each source operand **do**
    Recursively trace back to the root definition until *blockIdx* is found
    if A memory load is found **then** // uses value loaded from previous memory instruction
      Skip this instruction // *RCS* cannot be computed
    else if *blockIdx* is not found **then**
      Finish computation // *RCS* is zero
    else
      Continue
    end if
  end for
  if An $RCS \neq 0$ is found **then**
    Clone GetElementPtrInst to $index_K$, and replace *blockIdx* with a constant K
    Create another clone of GetElementPtrInst to $index_{K-1}$, and replace *blockIdx* with a constant $K - 1$
    Create a subtract instruction $\Delta = index_K - index_{K-1}$, and perform algebraic simplification on it
    if Δ contains only kernel-invocation-constants, and the block and thread indices are canceled out **then**
      return *RCS* // Runtime-Constant Stride is found
    end if
  end if
end for

Where data should be located can also be computed, as the affinity-based work scheduling algorithm determines where computation will be performed. For example, if one thread block accesses the first B bytes of a data structure and N consecutive thread blocks will be scheduled to the SMs in a GPU, the mapping algorithm allocates contiguous chunks of $B \times N$ bytes on each GPU. The equations to compute chunk_size and stack_id are as follows:

$$\text{chunk\_size} = min(4\text{KB}, B \times N_{\text{blocks\_per\_GPU}}), \tag{2}$$

$$ID_{\text{GPU}} = \left(\frac{\text{virtual\_addr} - \text{obj\_start\_addr}}{\text{chunk\_size}} \right) \bmod N_{\text{GPUs}}. \tag{3}$$

Please note that the chunk_size is upper-bounded by 4KB, since an arbitrary number of pages can be allocated in a single GPU for any large object with hardware support to map an entire page to a single GPU with CGP. obj_start_addr is the starting virtual address of an object. When the chunk_size is not a multiple of physical page size, we round up to the next multiple of pages. The resulting misaligned pages will be shared by SMs from two consecutive GPUs, but this is still better than un-aligned distribution of data across all GPUs. Commonly, $N_{\text{blocks\_per\_GPU}}$ is moderately big, since multiple thread blocks can run concurrently on an SM, which often results in a big chunk_size (greater or close to 4KB). Note that programs often use more than one data structure. Our proposed mechanism supports multiple data structures, since we compute the chunk_size for each data structure using its own B size based on the structure's access pattern.

We demonstrate how our data placement algorithm works with Figure 1, a code snippet from K-means Clustering. The size of each data element can be identified and computed at compile-time, and the first element and the number of consecutive elements that each thread accesses can also be analyzed with our compile-time analysis routine. In this example, each thread accesses `nfeatures` consecutive elements from `(pid * nfeatures)-th` element, as shown in lines 4 and 5 of Figure 1. Since each thread block has `blockDim.x` threads, `blockDim.x * nfeatures * sizeof(float)` is the B value. This means that the first thread block accesses B bytes from the starting address of the `in` array and the second thread block accesses next B bytes. Note that the number of thread blocks and threads per thread block are determined before kernel invocation.

When a `cudaMalloc` function is called, our extended runtime system uses this information and the B value to compute the chunk_size using Equation (2) for the corresponding data structure and decides whether it should be allocated with the FGP or CGP. If a data structure is accessed by multiple kernels, then the information of the first kernel that accesses it is used to compute the number of thread blocks per GPU. Accesses to three-dimensional (3D) data structures are often more complicated than those to 1D or 2D data structures, for which the index is typically computed with both `blockDim.x` and `blockDim.y`. In this article, we focus on 2D data structures and leave the extension to support the 3D data structures and more complex data structures for the future work.

5 EVALUATION METHODOLOGY

5.1 Hardware Configurations

We evaluate our mechanism using SST [52] with MacSim [27], a cycle-level microarchitecture simulator. Low-level DRAM timing constraints are faithfully simulated using DRAMSim2 [53], which was modified to model the HBM 2.0 specification [5]. Our default system configuration comprises the CPU and four GPUs, where each GPU consists of four SMs and 8GB HBM memory. We model the GPU based on the NVIDIA Fermi architecture [40]. More details on the simulated system configuration are provided in Table 1. We use 128B interleaving and 4KB interleaving to form the FGP and CGP, respectively. Each HBM channel is modeled to provide 32GB/s of peak

Table 1. Configuration of Simulated System

System	4 GPUs connected to the CPU with processor-centric topology [25]
GPU	Core: 4 2GHz SMs, fair-round-robin/affinity-based thread block scheduling policy Cache: 32KB core-private L1, 8-way, 4-cycle, 1MB shared L2, 16-way, 10-cycle Network: point-to-point network, 256GB/s Internal & 16GB/s Remote bandwidth
Memory	Each GPU has an 8GB HBM (HBM 2.0), interleaved at 128B

Table 2. Benchmark Categories

Category	Benchmarks
Block Exclusive	Breadth-First Search (BFS), Degree Centrality (DC), Page Rank (PR), Single-Source Shortest Path (SSSP), Betweenness Centrality (BC), Graph Coloring (GC), Needleman-Wunsch (NW)
Core Exclusive	*K*-means Clustering (KM), Gaussian Elimination (GE), *k*-Nearest Neighbors (NN), CFD Solver (CFD-M), Sparse-Matrix Dense-Vector Multiplication (SPMV), Sum of Absolute Differences (SAD), Dense Matrix-Matrix Multiply (MM)
Block Majority	Connected Component (CC)
Core Majority	MUMmerGPU (MG), Discrete Wavelet Transform (DWT)
Sharing	Triangle Count (TC), Hotspot3D (HS3D), Hybrid Sort (HS)

memory bandwidth; therefore, 256GB/s of total internal memory bandwidth is exploitable by each GPU. We model a Remote network to provide 16GB/s of memory bandwidth. We also perform detailed sensitivity studies, where we vary the bandwidth of Local and Remote networks.

5.2 Benchmarks

We use 20 memory-intensive benchmarks from GraphBIG [37], Rodinia [9], and Parboil [54]. We use the LLC Misses Per 1000 Instructions (MPKI) as an indicator for the memory-intensiveness. We classify a benchmark as being block-exclusive if almost all pages (>90%) are accessed by only one thread block, core-exclusive if almost all pages (>90%) are accessed by one GPU (i.e., multiple SMs in the same GPU), block-majority if the majority of pages (>60%) are accessed by only one thread block, core-majority if the majority of pages (>60%) are accessed by one GPU, and sharing if most of the pages are accessed by multiple GPUs. Table 2 summarizes the benchmarks and the category they belong to.

6 EVALUATION RESULTS

6.1 Performance

Figure 9 shows the performance improvement of CODA for the benchmarks described in Table 2. FGP-Only represents the baseline where every page is interleaved at 128B across GPUs, and CGP-Only represents the case where consecutive 4KB pages are allocated in consecutive GPUs in a circular order; this represents *affinity-unaware* data placement even when coarse-grained data allocation is available. CGP-Only+First-Touch-based Allocation (FTA) represents the case where an entire page is allocated to the GPU that first touches the page. We ignore the accesses from the CPU in determining the first access, since all pages are initially allocated by the CPU before

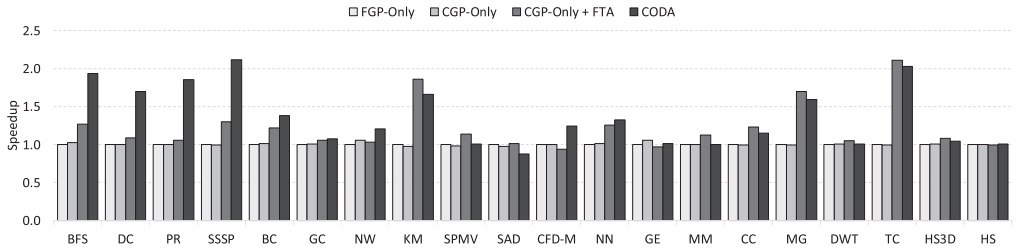


Fig. 9. Speedup over FGP-Only, CGP-Only, and an ideal first-touch-based allocation scheme (CGP-Only + FTA).

kernel invocation.[3] Even though this is not a practical implementation due to the lack of first-touch information at the time data are allocated (and often initialized) by the CPU, this can be a good indicator of the potential effectiveness of coarse-grained allocation for each benchmark. One simple way to implement first-touch-based allocation is to migrate pages on first access. We observed that this migration-based first-touch allocation is not very effective (not shown, 7% speedup, as opposed to 19% speedup of CGP-Only+FTA) mainly due to small number of reuses of memory pages after migrations (due to burst and clustered access patterns); that is, the migration overhead is not mitigated. This makes a case for better data allocation rather than reactive data movement.

Our evaluation results show that CODA outperforms both FGP-Only and CGP-Only by 31%. CODA even outperforms CGP-Only+FTA for most benchmarks. For pages that are exclusively accessed by a single GPU, allocating those pages on that GPU brings a substantial reduction in remote data accesses and increase in local data accesses. This variation in remote and local data accesses directly leads to the performance improvement, as remote data accesses are limited by the low bandwidth of the off-chip links, whereas local data accesses exploit the large internal memory bandwidth. Perhaps more importantly, such bandwidth discrepancy becomes even more pronounced as the interconnection network becomes overwhelmed with more remote data accesses. Though lower bandwidth of the off-chip links does not necessarily mean longer memory access latency, when coupled with the off-chip communication overheads such as queuing delays and/or external transfer time, average memory access latency can be significantly affected by the number of remote data accesses as well.

Notably, our mechanism localizes accesses *whenever possible* even for the benchmarks classified as sharing, in which most pages are accessed by many SMs (in multiple GPUs), thereby achieving performance improvements. The amount of performance gain each benchmark obtains depends on the distribution of accesses to page types (exclusive pages vs. shared pages). Specifically, if a majority of accesses are made to exclusive pages, the benchmark could gain a significant performance improvement from CODA. This is the case for TC, for example.

Overall, our mechanism achieves 1.56× and 1.13× average performance improvements over the baseline for block-exclusive and core-exclusive benchmarks, respectively. This is particularly effective in graph algorithms with large numbers of neighbor accesses (e.g., `BFS`, `DC`, `PR`, and `SSSP`), which are difficult to handle efficiently.

[3]NVIDIA PASCAL architecture or later GPUs support demand paging and runtime page migration [41, 42]. On those devices along with a newly introduced data allocation API (`cudaMallocManaged`), the statement that all pages are initially allocated by the CPU may not be true. Pages can be allocated by a GPU, generating page faults. While conceptually uncomplicated to use, demand paging and page migration cause significant performance degradation [6, 60]. Therefore, in this study, we assume and evaluate traditional and more general GPU programming model.

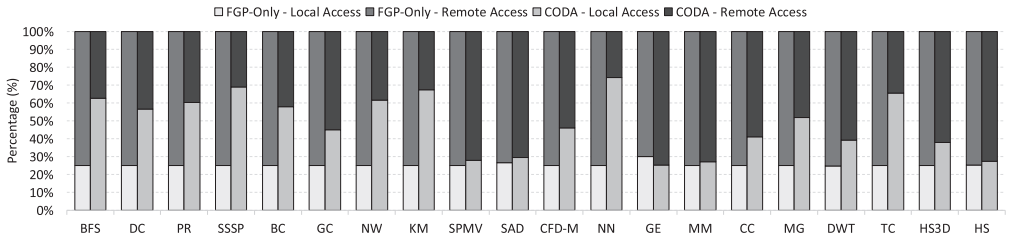


Fig. 10. Comparison of local and remote data accesses between FGP-Only and our mechanism (CODA).

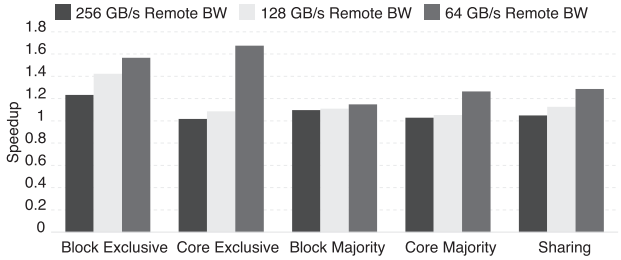


Fig. 11. Speedup with different remote bandwidth among GPUs.

6.2 Local vs. Remote Access

Figure 10 shows distribution of memory accesses, local versus remote, for the baseline and how it varies with our mechanism. Our mechanism significantly reduces remote data accesses for all the evaluated benchmarks but one, GE.[4] A substantial reduction in remote data accesses and an increase in local data accesses contribute to the performance improvement for the following reasons. First, local data accesses can utilize the large internal memory bandwidth, while remote data accesses are limited by the lower memory bandwidth of the off-chip links. Second, for the remote data accesses, a great amount of time could be spent on waiting for network due to the off-chip communication. This can be incurred as a result of limited network bandwidth, but can be exacerbated further due to the artifacts of the off-chip communication, such as queuing delays, routing delays, and so on. Our mechanism significantly reduces remote data accesses, enabling the utilization of large internal memory bandwidth and also mitigating the effect of interconnection network congestion by placing memory pages in the same GPU in which the computation is to be performed.

Our mechanism is especially effective for the block-exclusive and core-exclusive benchmarks. On average, 47% and 34% remote data accesses are reduced, respectively. Even for the sharing benchmarks, by identifying the pages that are accessed by a few thread blocks or SMs, and allocating them where the computation is to be performed, our mechanism reduces 32% remote data accesses.

6.3 Sensitivity to Bandwidth

Even for highly provisioned systems with unrealistically large Remote bandwidth and low remote memory access latency, co-location of thread blocks and the data they access improves performance, as shown in Figure 11. This is because even in such systems, remote memory accesses cannot be completely free from all resource conflicts. Careful data placement, as is enabled by our

[4]As opposed to the case of TC in Section 6.1, GE has a majority of accesses to shared pages, and for this reason, remote data accesses are not reduced a lot with CODA, even though it is classified as core-exclusive, because we classified benchmarks based on the distribution of pages, not based on the distribution of accesses to page types (exclusive page vs. shared page).

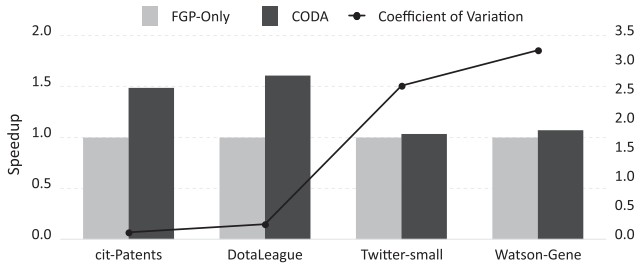


Fig. 12. PageRank performance with different graphs.

mechanism, can significantly reduce the possibility of such conflicts and therefore can contribute to the performance improvement.

This evaluation is to show how sensitive performance is to Remote bandwidth. We can observe that when 256GB/s of Remote bandwidth is available, Remote bandwidth is no longer a performance bottleneck (relative performances of all categories are close to 1, except for the block-exclusive category). The big gap in the core-exclusive category appears, because the applications in that category are limited by Remote bandwidth of 64GB/s and not by 128GB/s Remote bandwidth.

Even when a system has 256GB/s of aggregated Remote bandwidth, our mechanism improves performance by 8% (up to 23%). It should be noticed that as the gap between Local bandwidth and Remote bandwidth increases (Remote bandwidth is decreased while Local bandwidth remains the same), our mechanism provides more benefit by reducing remote data accesses and opening up more opportunity to exploit large internal memory bandwidth, thereby mitigating the performance penalty of the off-chip communication (performance improvement goes up to 15.2% and 37.4%, respectively).

6.4 Sensitivity to Graph Properties

In graph computing, the number of vertices and their neighbors that each thread block accesses highly depends on graph properties. To examine the impact of the graph properties on our proposed mechanism, we differentiate the properties that can be estimated at the time the graph is preprocessed[5] from those that cannot be estimated. Basic graph properties such as the number of vertices and edges can be obtained at the time the graph is preprocessed. These, combined with the number of threads per thread block, which is determined based on the resource constraints of the underlying hardware, can be used to estimate the average number of edges that each thread block accesses (μ) before kernel invocation and the standard deviation (σ) of it. The coefficient of variation of a graph, which can be estimated as σ/μ, is a good indicator of how regular a graph is: A graph with a small coefficient of variation is considered regular. Therefore, the granularity at which the graph should be distributed, or the block stride distance, can be determined.

Figure 12 compares the performance of FGP-Only and CODA, using the PageRank workload. The evaluation is based on four real-world graphs, which have 59K to 9M vertices. Graphs are sorted based on their regularity: Graphs with a smaller coefficient of variation appear toward the left side of the figure. The coefficient of variation of each graph is also depicted. We confirm that the effectiveness of our mechanism depends highly on graph properties. Regular graphs benefit more from our mechanism (55%) than irregular graphs (5%), since the estimation accuracy depends only on the properties that can be estimated at the time graph is preprocessed. Notably,

[5]The term preprocessing generally implies a heavy-weight operation such as a clever partitioning to reduce communication. In this study, however, we only extract basic properties of a graph without scanning through the entire graph.

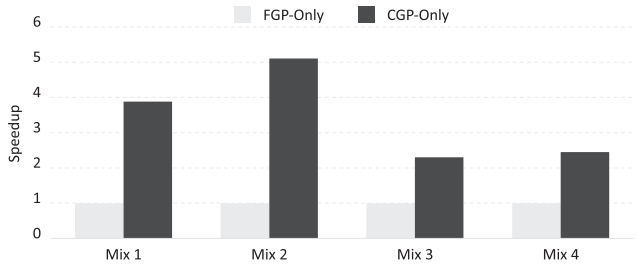


Fig. 13. Performance of multiple applications.

CODA does *not* degrade performance in any case, since it detects the memory pages that are exclusively accessed by one GPU and localizes them with CGP, while distributing other memory pages with FGP, as in the case of FGP-Only.

6.5 Multiprogrammed Workloads

To further analyze the impact of having hardware that provides the ability to map an entire page to a single GPU using CGP, we evaluate our CGP-Only configuration with four mixes of multiprogrammed workloads. Each benchmark is chosen randomly from each category to construct a multiprogrammed workload. Figure 13 compares the performance of CGP-Only with that of FGP-Only, showing that the CGP-Only outperforms the FGP-Only for all the workloads. With FGP-Only hardware, every memory page is distributed across all GPUs, which results in a significant number of remote data accesses from all applications. With hardware that can map an entire page to a single GPU, as enabled by our mechanism, however, memory pages that an application accesses can be allocated to the GPU where the application is executed, and hence, all the accesses can exploit the large internal memory bandwidth within the GPU. This is an important contribution, since it is infeasible or difficult to reduce remote data accesses in the presence of multiple workloads running in a system.[6]

6.6 Impact of Interleaving Granularity

So far we have demonstrated the necessity of the coarse-grained memory interleaving (technically, selective use of CGP and FGP) for the efficient use of multiple GPUs. One might consider using *just* coarse-grained memory interleaving in a system with multiple GPUs. However, in this section we present the performance of FGP-Only and CGP-Only with a centralized GPU that has the same overall compute capability as that of all the GPUs in the multiple GPU system to demonstrate the necessity of the fine-grained memory interleaving as well. When an application runs on the centralized GPU, as in traditional GPU systems, it is desirable that the memory objects it accesses are distributed across multiple memories to achieve maximum memory bandwidth utilization by distributing concurrent accesses across all available memory interfaces. Figure 14 shows the performance of the GPU with memories interleaved at different granularities. FGP-Only and CGP-Only indicate the use of fine-grained interleaved memory and coarse-grained interleaved memory, respectively. Our evaluation results show that FGP-Only outperforms the CGP-Only by 1.48× due to better memory bandwidth utilization.

[6]Please note that this experiment is intended to show the necessity of having hardware that provides the ability to map an entire page to a single GPU, not to compare the performance of the baseline configuration (FGP-Only) and CODA, although it can be easily expected that CODA would perform as well as CGP-Only.

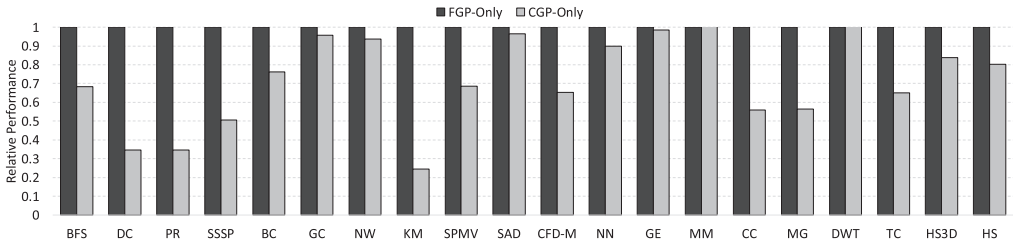


Fig. 14. Performance impact of interleaving granularity.

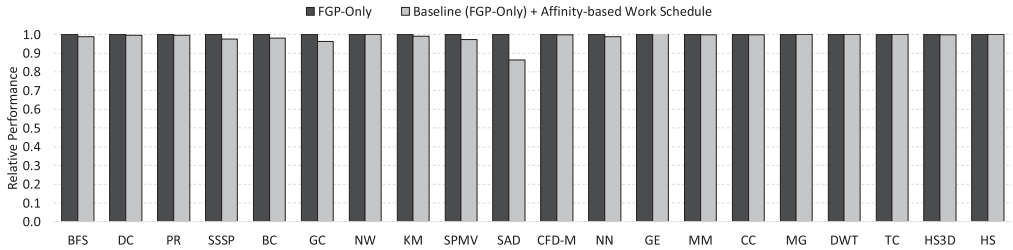


Fig. 15. Performance impact of an affinity-based work scheduling mechanism.

6.7 Impact of Affinity-Based Scheduling

Thread blocks cannot be scheduled to any SM with our affinity-based work scheduling mechanism. In this section, we evaluate the performance impact of the affinity-based work scheduling mechanism. Figure 15 compares the performance of the affinity-based work scheduling mechanism (FGP-Only + Affinity-based Work Schedule) and that of the baseline (FGP-Only). All our evaluated benchmarks are virtually unaffected by the restricted scheduling mechanism, as expected, except for one benchmark, SAD. The reason why the performance of SAD is degraded by the affinity-based work scheduling is that the number of thread blocks is relatively small (61) considering the number of GPUs and available SMs (16). Maintaining load balancing across all available compute resources might be more crucial than carefully co-locating thread blocks and the data they access, when compute resource bounds the overall performance. This problem can be alleviated with resource-monitoring-based schemes.

7 DISCUSSIONS

7.1 Complex Address Mapping

So far, we have assumed a simple address mapping scheme for ease of explanation. Modern processors, however, use more complex address mapping schemes such as XORing multiple bits (not necessarily consecutive) for channel selection [46]. In this section, we discuss the applicability of our dual-mode address mapping mechanism in such systems. Note that computation and data co-location algorithm presented in Section 4.3 is orthogonal to the address mapping scheme used in the underlying system. Although the detailed address mapping scheme differs for different architectures, the mappings can be classified into those that use the channel-selection bits exclusively (i.e., they are not used as part of the row- or column-selection) and those that do not (i.e., at least one bit from the channel-selection bits is used as part of the row- or column-selection). Our dual-mode address mapping mechanism can be easily extended to support a system with the former class of mappings, where channel-selection bits are used exclusively, by swapping the channel-selection bits with other higher order bits after XOR operation. However, it is not trivial to support a system

with the latter class of mappings, where channel-selection bits are not exclusively used. One way might be to identify which bits are used exclusively for the channel-selection and which bits are not, and then carefully swapping the channel-selection bits with those that are not used for channel-selection. This requires further investigation and is a part of our future work.

7.2 Large Page and Memory Management

Large pages have been used to mitigate address translation overheads by reducing the number of PTEs to maintain and increasing TLB hit rates. However, it comes at a cost, such as internal fragmentation, memory bloat, and increased load-to-use latency [6]. In this section, we discuss the applicability of our dual-mode address mapping mechanism for the large pages. Again, the computation and data co-location algorithm presented in Section 4.3 is orthogonal to the page size. First, our dual-mode address mapping can be easily extended for the large pages. For 2MB pages, for example, address bits [22:21] can be used (instead of address bits [13:12] in the case of 4KB page) to index GPUs to allocate the entire page in a single GPU. However, the key challenge in supporting large page is not about choosing which bits to use for GPU selection but about dealing with fragmentation issues. Although our mechanism may complicate page management and potentially increase fragmentation issues, we believe that if page-groups are small (e.g., four or eight pages), this is likely to not be significantly more complicated than normal page management. Also, the memory manager can be modified to deal with page-groups for most operations (e.g., flushing out to disk) for better memory management. This requires further exploration and is a part of our future work.

7.3 PTE Extension

Our proposed mechanism requires a modification to the PTE format. X86 ISA reserves 3 bits [11:9] for future usage [23], so we can use one of the bits to indicate the granularity information. When a system employs large pages, extra bits are available in the PTE, which gives more freedom to modify PTE contents.

7.4 NUMA or NUCA Systems

In this section, we discuss the difference and uniqueness of our system from the conventional NUMA [8] or NUCA (Non-Uniform Cache Access) [21] systems in CPUs. First, in NUMA systems, memory policies such as node-local or interleave can be specified and (relatively) easily controlled. For example, the first-touch based page allocation has already been used in NUMA systems. On the contrary, the first-touch based page allocation *cannot* be used in GPU systems (GPUs prior to NVIDIA Pascal architecture [41]) due to the lack of first-touch information (recall that data structures are allocated and initialized by the CPU before kernel invocation). Even if the first-touch information *were* available, a memory page could not be allocated in a single GPU without hardware support for the localization. Furthermore, since shared data across multiple cores are often cached in the CPUs, the penalty of NUMA is often reduced. However, in GPUs, the cache size is much smaller than CPUs, so caches cannot hide the penalty of NUMA easily. Second, NUCA systems (e.g., R-NUCA [21]) rely on data migration after an access pattern is identified. The migration overhead is much smaller in NUCA systems than in our multiple GPU system, because the former migrates data within a single device (e.g., a tiled L2 cache architecture), whereas the latter migrates data across multiple devices connected via comparatively low-bandwidth, high-latency interconnect links.

8 RELATED WORK

Multiple GPUs. Static-time data allocation has also been researched in the context of multiple GPUs. A system with multiple GPUs is closer to an MPI-based system, since each GPU has its own memory and physical address space is not interleaved across multiple GPU memories. In this sense, several algorithms were proposed to automatically partition data among multiple devices (e.g., multiple GPUs or CPUs and GPUs) [7, 16, 28, 30, 31, 33, 50]. The focus of our work is to enable data partitioning among GPU memories via selective use of coarse-grained interleaving (hardware mechanism) and to enable co-location of computations with the data they access (software mechanism). Ziabari et al. [61] have proposed a mechanism that supports seamless data transfer across all the devices (a CPU and one or more GPUs) in the system, while creating a hierarchical view between the memory of the GPUs and the host memory. Kim et al. [26] have proposed a GPU memory network to simplify data sharing between discrete GPUs. Arunkumar et al. [4] have demonstrated that package-level integration of multiple GPU modules (GPMs) can enable continuous performance scaling and proposed a technique to improve GPM data locality and minimize the sensitivity on inter-GPM bandwidth. They used the Pascal and later architectures where unified memory and demand paging are supported, in which when a page is first accessed in a kernel, a page fault is detected, and the page fault handling procedure is performed. Realizing the first-touch based allocation on those architectures is not revolutionary, since it can be done by modifying GPU driver where page allocation is performed. However, it requires applications to use unified memory in the first place, which indicates that traditional GPU applications, in which CPU allocates data and GPU consumes or processes it after kernel(s) are launched, cannot gain the benefit of the mechanism as is. However, our mechanism, which does not require any application modifications, can support any applications for better compute and data co-placement.

Memory-Level Parallelism. Zurawski et al. [62] presented an address bit swapping scheme to increase memory-level parallelism by reducing the row buffer conflicts in traditional DRAM systems, which is used in AlphaStation 600 5-series workstations. Zhang et al. [59] proposed a permutation-based page interleaving scheme to reduce row-buffer conflicts and to exploit data access locality in the row-buffer. Ghasempour et al. [14] proposed a hardware mechanism to dynamically change the address mapping to increase bank-level parallelism at the cost of a significant amount of page migration overhead. While our proposed mechanism also uses address bit swapping scheme, it is different from these works in two ways. First, our mechanism applies address mapping scheme at a page granularity such that pages with different address mappings co-exist in the same memory space. Our mechanism is lightweight in a sense that it incurs negligible performance overhead and does not have any impact on the cache coherence protocol or virtual address translation. Second, our mechanism does not require large-scale page migrations; only a few (e.g., four or eight, depending on the number of memory stacks) pages are affected, since we selectively use CGP at the page-group granularity.

Static-time Data Alignment. Static-time data allocation has a long history of research. High Performance Fortran (HPF) provides compiler directives to specify data alignment among processors [51]. Although our mechanism shares the same philosophy with the HPF directives such as *block* or *cyclic*, they are different in the sense that the HPF directives are applied at virtual address space, whereas it is done in the physical memory space in our mechanism, since the source of non-uniformity of memory access pattern is caused when a virtual page is mapped to the physical memory domain. Sung et al. [55] presented a formulation and language extension that enables automatic data layout transformation for structured grid codes in CUDA. It distributes concurrent memory requests evenly to DRAM channels and banks, thereby achieving significant speedup. Thanh-Hoang et al. [56] recently proposed an architectural solution called Data Layout Transformation (DLT) for optimizing data movement across system components. While their

accelerator can make good use of memory bandwidth for data movement, it requires application changes to use their instructions. Our mechanism, however, does not require any application modifications and provides high data locality with slight changes in virtual to physical address mapping.

Processing in memory. Processing in memory was proposed decades ago [11, 12, 15, 20, 24, 34, 43–45]. Recent advances in 3D stacking technology have given a boost to PIM research [1–3, 10, 17–19, 22, 35, 36, 38, 57, 58] to accelerate workloads in various domains (e.g., large-scale graph processing workloads [1, 35], Map-Reduce workloads [49], and HPC applications [57]). Akin et al. [3] proposed solutions for efficient data reorganization, combining a DRAM-aware reshape accelerator integrated within 3D-stacked DRAM, and a mathematical framework that is used to represent and optimize the reorganization operations. Hsieh et al. [22] (TOM) addressed the issue of local and remote memory accesses in a system with multiple PIM memory stacks. It performs runtime profiling to learn best address mapping for data accessed by offloading candidates and distributes that data with the discovered mapping. Although our work focus is on a system with multiple GPUs, it is worth comparing our mechanism against theirs in the context of a system with distributed GPUs, irrespective of whether they are multiple full-fledged GPUs or GPUs in memory stacks. In contrast to our proposal, this work (1) essentially delays and decelerates the regular kernel execution, because it tests all different address mappings (10 mappings, sweeping from bit position 7 to bit position 16) for all the data accessed by offloading candidates during the runtime learning phase, and (2) implicitly assumes a hardware mechanism to distribute data with different mappings.

9 CONCLUSION

We introduce CODA that enables co-location of compute and data in a system with multiple GPUs. Our mechanism is built on the key observation that code and data alignment is one of the most important factors in achieving high performance in multiple GPU systems. The key idea of CODA is to identify exclusively accessed data and place the data along with the thread block that accesses it in the same GPU. For identification, we implement a compile-time analysis pass on the LLVM infrastructure and estimate runtime constant stride between thread blocks. We also employ fine-grained interleaved memory for the data that is shared by all thread blocks. For coarse-grained data placement on top of fine-grained interleaved memory, we make lightweight changes to the virtual to physical address mapping (not the physical address itself), such that both fine-grained and coarse-grained interleaved pages can co-exist while not affecting the cache coherence protocol or virtual address translation. For data-aware thread block scheduling, we use an affinity-based scheduling policy. We have shown that CODA improves performance by 31% and reduces 38% remote traffic over a baseline across a wide range of workloads.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their valuable comments. We gratefully acknowledge the support of National Science Foundation XPS-1337177 and XPS-1533767. AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

REFERENCES

[1] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*.

[2] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*.

[3] Berkin Akin, Franz Franchetti, and James C. Hoe. 2015. Data reorganization in memory using 3D-stacked DRAM. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*.

[4] Akhil Arunkumar, Evgeny Bolotin, Benjamin Cho, Ugljesa Milic, Eiman Ebrahimi, Oreste Villa, Aamer Jaleel, Carole-Jean Wu, and David Nellans. 2017. MCM-GPU: Multi-chip-module GPUs for continued performance scalability. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'15)*.

[5] JEDEC Solid State Technology Association. 2015. High Bandwidth Memory (HBM) DRAM. JESD235A (November 2015).

[6] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J. Rossbach, and Onur Mutlu. 2017. Mosaic: A GPU memory manager with application-transparent support for multiple page sizes. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'17)*.

[7] Javier Cabezas, Lluís Vilanova, Isaac Gelado, Thomas B. Jablin, Nacho Navarro, and Wen-mei Hwu. 2014. Automatic execution of single-GPU computations across multiple GPUs. In *Proceedings of the 2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT'14)*.

[8] Rohit Chandra, Scott Devine, Ben Verghese, Anoop Gupta, and Mendel Rosenblum. 1994. Scheduling and page migration for multiprocessor compute servers. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'94)*.

[9] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC'09)*.

[10] Michael Chu, Nuwan Jayasena, Dongping Zhang, and Mike Ignatowski. 2013. High-level programming model abstractions for processing in memory. In *Proceedings of the 1st Workshop on Near-Data Processing (WoNDP'13)*.

[11] Duncan Elliott, W. Martin Snelgrove, and Michael Stumm. 1992. Computational ram: A memory-simd hybrid and its application to Dsp. In *Proceedings of the IEEE Custom Integrated Circuits Conference 1992*.

[12] Duncan Elliott, Michael Stumm, W. Martin Snelgrove, Christian Cojocaru, and Robert McKenzie. 1999. Computational RAM: Implementing processors in memory. *IEEE Des. Test* 16, 1 (Jan. 1999), 32–41.

[13] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. 2007. Dynamic warp formation and scheduling for efficient GPU control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'07)*.

[14] Mohsen Ghasempour, Aamer Jaleel, Jim D. Garside, and Mikel Luján. 2016. DReAM: Dynamic re-arrangement of address mapping to improve the performance of DRAMs. In *Proceedings of the International Symposium on Memory Systems (MEMSYS'16)*.

[15] Maya Gokhale, Bill Holmes, and Ken Iobst. 1995. Processing in memory: The terasys massively parallel PIM array. *Computer* 28, 4 (April 1995), 23–31.

[16] Dominik Grewe and Michael F. P. O'Boyle. 2011. A static task partitioning approach for heterogeneous systems using OpenCL. In *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software (CC/ETAPS'11)*.

[17] Ramyad Hadidi, Bahar Asgari, Burhan Ahmad Mudassar, Saibal Mukhopadhyay, Sudhakar Yalamanchili, and Hyesoon Kim. 2017. Demystifying the characteristics of 3D-stacked memories: A case study for hybrid memory cube. In *2017 IEEE International Symposium on Workload Characterization (IISWC'17)*.

[18] Ramyad Hadidi, Bahar Asgari, Jeffrey Young, Burhan Ahmad Mudassar, Kartikay Garg, Tushar Krishna, and Hyesoon Kim. 2018. Performance implications of NoCs on 3D-stacked memories: Insights from the hybrid memory cube. In *Proceedings of the 2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'18)*.

[19] Ramyad Hadidi, Lifeng Nai, Hyojong Kim, and Hyesoon Kim. 2017. CAIRO: A compiler-assisted technique for enabling instruction-level offloading of processing-in-memory. *ACM Trans. Arch. Code Optimiz.* 14, 4, Article 48 (Dec. 2017), 25 pages.

[20] Mary Hall, Peter Kogge, Jeff Koller, Pedro Diniz, Jacqueline Chame, Jeff Draper, Jeff LaCoss, John Granacki, Jay Brockman, Apoorv Srivastava, William Athas, Vincent Freeh, Jaewook Shin, and Joonseok Park. 1999. Mapping irregular applications to DIVA, a PIM-based data-intensive architecture. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing (SC'99)*.

[21] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. 2009. Reactive NUCA: Near-optimal block placement and replication in distributed caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*.

[22] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike OConnor, Nandita Vijaykumar, Onur Mutlu, and Stephen W. Keckler. 2016. Transparent offloading and mapping (TOM): Enabling programmer-transparent

near-data processing in GPU systems. In *Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA'16)*.

[23] Intel Corporation. 2007. *Intel®64 and IA-32 Architectures Software Developer's Manual*.

[24] Yi Kang, Wei Huang, Seung-Moon Yoo, Diana Keen, Zhenzhou Ge, Vinh Lam, Pratap Pattnaik, and Josep Torrellas. 1999. FlexRAM: Toward an advanced intelligent memory system. In *Proceedings of the 2012 IEEE 30th International Conference on Computer Design (ICCD'99)*.

[25] Gwangsun Kim, John Kim, Jung Ho Ahn, and Jaeha Kim. 2013. Memory-centric system interconnect design with hybrid memory cubes. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT'13)*.

[26] Gwangsun Kim, Minseok Lee, Jiyun Jeong, and John Kim. 2014. Multi-GPU system design with memory networks. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'14)*.

[27] Hyesoon Kim, Jaekyu Lee, Nagesh B. Lakshminarayana, Jaewoong Sim, Jieun Lim, Tri Pho, Hyojong Kim, and Ramyad Hadidi. 2012. *MacSim: A CPU-GPU Heterogeneous Simulation Framework User Guide*.

[28] Jungwon Kim, Honggyu Kim, Joo Hwan Lee, and Jaejin Lee. 2011. Achieving a single compute device image in OpenCL for multiple GPUs. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'11)*.

[29] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO'04)*.

[30] Janghaeng Lee, Mehrzad Samadi, Yongjun Park, and Scott Mahlke. 2013. Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT'13)*.

[31] Janghaeng Lee, Mehrzad Samadi, Yongjun Park, and Scott Mahlke. 2015. SKMD: Single kernel on multiple devices for transparent CPU-GPU collaboration. *ACM Trans. Comput. Syst.* (2015).

[32] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. 2008. NVIDIA tesla: A unified graphics and computing architecture. *IEEE Micro* 28, 2 (March 2008), 39–55.

[33] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. 2009. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*.

[34] Richard C. Murphy, Peter M. Kogge, and Arun Rodrigues. 2000. The characterization of data intensive memory workloads on distributed PIM systems. In *Revised Papers from the Second International Workshop on Intelligent Memory Systems (IMS'00)*.

[35] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. 2017. GraphPIM: Enabling instruction-level PIM offloading in graph computing frameworks. In *Proceedings of the 2017 IEEE 23rd International Symposium on High Performance Computer Architecture (HPCA'17)*.

[36] Lifeng Nai, Ramyad Hadidi, He Xiao, Hyojong Kim, Jaewoong Sim, and Hyesoon Kim. 2018. CoolPIM: Thermal-aware source throttling for efficient PIM instruction offloading. In *Proceedings of the 2018 IEEE 32nd International Symposium on Parallel and Distributed Processing (IPDPS'18)*.

[37] Lifeng Nai, Yinglong Xia, Ilie G. Tanase, Kim Hyesoon, and Ching-Yung Lin. 2015. GraphBIG: Understanding graph computing in the context of industrial solutions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*.

[38] Ravi Nair, Samuel Antao, Carlo Bertolli, Pradip Bose, Jose Brunheroto, Tong Chen, Chen-Yong Cher, Carlos Costa, Jun Doi, Constantinos Evangelinos, Bruce Fleischer, Thomas Fox, Diego Gallo, Leopold Grinberg, John Gunnels, Arpith Jacob, Philip Jacob, Hans Jacobson, Tejas Karkhanis, Changhoan Kim, Jaime Moreno, Kevin O'Brien, Martin Ohmacht, Yoonho Park, Daniel Prener, Bryan Rosenburg, Kyung Ryu, Olivier Sallenave, Mauricio Serrano, Patrick Siegl, Krishnan Sugavanam, and Zehra Sura. 2015. Active memory cube: A processing-in-memory architecture for exascale systems. *IBM J. Res. Dev.* 59, 2/3 (March 2015), 17:1–17:14.

[39] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. 2011. Improving GPU performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'11)*.

[40] NVIDIA Corp. 2009. NVIDIA's Next Generation CUDA™ Compute Architecture: Fermi™. Retrieved from https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[41] NVIDIA Corp. 2016. NVIDIA Tesla P100. Retrieved from https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[42] NVIDIA Corp. 2017. NVIDIA Tesla V100. Retrieved from http://www.nvidia.com/object/volta-architecture-whitepaper.html.

[43] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. 1998. Active pages: A computation model for intelligent memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA'98)*.

[44] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. 1997. Intelligent RAM (IRAM): Chips that remember and compute. In *Proceedings of the IEEE International Solids-State Circuits Conference (ISSCC'97)*.

[45] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. 1997. A case for intelligent RAM. *IEEE Micro* (1997).

[46] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM addressing for cross-CPU attacks. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security'16)*.

[47] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. 2014. Architectural support for address translation on GPUs: Designing memory management units for CPU/GPUs with unified address spaces. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*.

[48] Jason Power, Mark D. Hill, and David A. Wood. 2014. Supporting x86-64 address translation for 100s of GPU lanes. In *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14)*.

[49] Seth H. Pugsley, Jeffrey Jestes, Huihui Zhang, Rajeev Balasubramonian, Vijayalakshmi Srinivasan, Alper Buyuktosunoglu, Al Davis, and Feifei Li. 2014. NDC: Analyzing the impact of 3D-stacked memory+logic devices on MapReduce workloads. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'14)*.

[50] Thejas Ramashekar and Uday Bondhugula. 2013. Automatic data allocation and buffer management for multi-GPU machines. *ACM Trans. Arch. Code Optimiz.* (2013).

[51] Rice University, CORPORATE. 1993. High Performance Fortran Language Specification. *SIGPLAN Fortran Forum* 12, 4 (Dec. 1993), 1–86. https://doi.org/10.1145/174223.158909.

[52] Arun F. Rodrigues, K. Scott Hemmert, Brian W. Barrett, Chad Kersey, Ron A. Oldfield, Marlo Weston, R. Risen, Jonathan Cook, Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. 2011. The structural simulation toolkit. *ACM SIGMETRICS Performance Evaluation Review* (2011).

[53] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. 2011. DRAMSim2: A cycle accurate memory system simulator. *IEEE Comput. Arch. Lett.* (2011).

[54] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen mei W. Hwu. 2012. *Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. IMPACT Technical Report* (2012).

[55] I-Jui Sung, John A. Stratton, and Wen-Mei W. Hwu. 2010. 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 (PACT'10)*.

[56] Tung Thanh-Hoang, Amirali Shambayati, and Andrew A. Chien. 2016. A data layout transformation (DLT) accelerator: Architectural support for data movement optimization in accelerated-centric heterogeneous systems. In *Proceedings of the 2016 Design, Automation Test in Europe Conference Exhibition (DATE'16)*.

[57] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L. Greathouse, Lifan Xu, and Michael Ignatowski. 2014. TOP-PIM: Throughput-oriented programmable processing in memory. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing (HPDC'14)*.

[58] Dong Ping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph Greathouse, Mitesh Meswani, Mark Nutter, and Mike Ignatowski. 2013. A new perspective on processing-in-memory architecture design. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC'13)*.

[59] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. 2000. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO'00)*.

[60] Tianhao Zheng, David W. Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W. Keckler. 2016. Towards high performance paged memory for GPUs. In *Proceedings of the 2016 IEEE 22nd International Symposium on High Performance Computer Architecture (HPCA'16)*.

[61] Amir Kavyan Ziabari, Yifan Sun, Yenai Ma, Dana Schaa, José L. Abellán, Rafael Ubal, John Kim, Ajay Joshi, and David Kaeli. 2016. UMH: A hardware-based unified memory hierarchy for systems with multiple discrete GPUs. *ACM Trans. Arch. Code Optimiz.* (2016).

[62] John H. Zurawski, John E. Murray, and Paul J. Lemmon. 1995. The design and verification of the alphastation 600 5-series workstation. *Dig. Techn. J.* (1995).

Received February 2018; revised May 2018; accepted June 2018