Locality-Centric Data and Threadblock Management for Massive GPUs

Mahmoud Khairy
Purdue University
abdallm@purdue.edu

Vadim Nikiforov
Purdue University
vnikifor@purdue.edu

David Nellans
NVIDIA
dnellans@nvidia.com

Timothy G. Rogers
Purdue University
timrogers@purdue.edu

***Abstract*—Recent work has shown that building GPUs with hundreds of SMs in a single monolithic chip will not be practical due to slowing growth in transistor density, low chip yields, and photoreticle limitations. To maintain performance scalability, proposals exist to aggregate discrete GPUs into a larger virtual GPU and decompose a single GPU into multiple-chip-modules with increased aggregate die area. These approaches introduce non-uniform memory access (NUMA) effects and lead to decreased performance and energy-efficiency if not managed appropriately. To overcome these effects, we propose a holistic *Locality-Aware Data Management* (LADM) system designed to operate on massive logical GPUs composed of multiple discrete devices, which are themselves composed of chiplets. LADM has three key components: a threadblock-centric index analysis, a runtime system that performs data placement and threadblock scheduling, and an adaptive cache insertion policy. The runtime combines information from the static analysis with topology information to proactively optimize data placement, threadblock scheduling, and remote data caching, minimizing off-chip traffic. Compared to state-of-the-art multi-GPU scheduling, LADM reduces inter-chip memory traffic by $4\times$ and improves system performance by $1.8\times$ on a future multi-GPU system.**

***Index Terms*—GPGPU, Multi-GPU, NUMA**

I. Introduction

GPU accelerated workloads are commonly used in deep learning and exascale computing systems [79], [81]. These workloads exhibit high levels of implicit parallelism, which enables performance scalability, but only if GPUs can continue to scale their hardware resources. Over the past decade, GPUs have more than quadrupled the number of Streaming Multiprocessors (SMs) in their designs, while simultaneously increasing their on-chip transistors by an order of magnitude. Prior work by Arunkumar et al. [5] demonstrates linear performance scalability if GPU resources (SMs, SM-interconnect bandwidth, registers, caches, and DRAM bandwidth) are able to scale proportionally. However, building a GPU with hundreds of SMs in a single monolithic GPU die will not be possible due to low manufacturing yields and the high cost of building large chips at small technology nodes [5], [33].

To overcome these problems and enable continuous performance scaling as Moore's law slows [13], [73], researchers have proposed increasing GPU transistor count by aggregating multiple GPUs together (as a single logical GPU) as well as disaggregating single-GPUs into scalable multi-chip-modules [5], [51], [53], [63]. Compared to single-chip systems, chiplet based architectures [1] are desirable because they provide a larger aggregate chip perimeter for I/O, enabling a higher number of DRAM interfaces to be connected to the system and thus scale memory bandwidth and capacity alongside compute resources [5], [6], [28], [32], [48], [79].

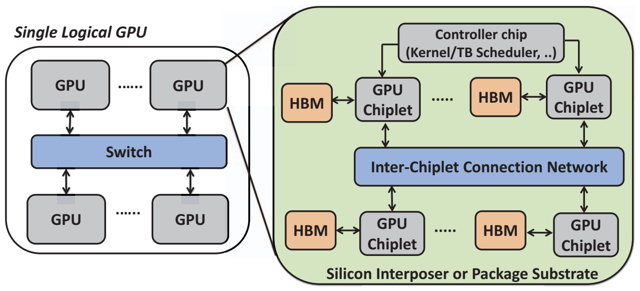


Fig. 1: Future massive logical GPU containing multiple discrete GPUs, which are themselves composed of chiplets in a hierarchical interconnect.

Future chiplet-based designs will be limited by the size of silicon interposers or the short haul, high bandwidth interconnects needed to traverse a small printed circuit board. Compared to chiplets, multiple GPUs are more easily combined into large coordinated systems but suffer from lower inter-GPU bandwidth, which increases the NUMA performance penalty. As shown in Figure 1, these approaches are complimentary and it is likely that both will be employed in future systems with hierarchical interconnects to create a massive logical GPU.

Architecture and runtime systems must coordinate to maintain the existing single-GPU programming model and support transparent scaling for current CUDA programs. The goal is to create a *single programmer-visible GPU* that may be comprised of hierarchical locality domains. Maintaining this illusion enables rapid software development on small local GPU resources while allowing scalable performance on larger and more complex GPU systems. Transparently overcoming locality effects will be a challenging problem for GPUs over the next decade. Such an extreme NUMA scale requires new techniques to place pages, cache data, and schedule the many thousands of threads managed in such systems.

[1]In this paper, chiplet and multi-chip-module (MCM) are used interchangeably.

Recent work on static analysis for transparent multi-GPU programs, CODA [36], is a step in the right direction. Using the compiler to perform index analysis, CODA calculates the width of the data accessed by one threadblock and ensures that threadblocks and the data they access are placed on the same GPU for the locality types they can identify. However, a more robust analysis of the code is required to exploit different GPU access patterns on hierarchical GPUs. In this work, we deconstruct the accesses patterns observed across a diverse set of GPU applications and detail which patterns are captured by recent state-of-the-art NUMA-GPU mechanisms and those that remain unexploited. We show that many of the previously unexploited patterns can be successfully detected by static analysis, which we use to drive data placement, caching, and thread scheduling decisions in our Locality-Aware Data Management (LADM) system.

Static index analysis has been extensively used in sequential code to perform affine loop transformations, eliminate data dependencies, and partition work for automatic parallelization [3], [46]. In many ways, a static analysis of a GPU program is more straightforward than a sequential one, as the parallelism in CUDA programs is inherent to the programming model. A parallel GPU program can be transformed into a sequential program by converting the threadblock and grid dimensions into loop index variables on data-parallel outer loops. Once this transformation is made, any sequential program analysis can be applied to the GPU code. However, it is less obvious how the nature of the hierarchical programming model (i.e., threadblocks and kernels) can be combined with sequential locality analysis to map schedulable chunks of work (i.e., threads within the same threadblock) to data structure accesses. To tackle this issue, we introduce the concept of datablock locality analysis that maps each threadblock in a kernel to chunks of data we predict it will access.

Fundamentally, the programming model for GPUs is different than for CPUs. Due to their massively-multithreaded nature, GPU programs are composed of many fine-grained threads, where each individual thread exhibits little spatial or temporal locality to global memory. This, combined with the expressiveness of thread IDs in the CUDA programming model creates both a new challenge and an interesting opportunity to apply static analysis for NUMA data placement and thread scheduling. We make three primary contributions over the state-of-the-art NUMA-GPU systems:

- We perform a detailed analysis of the locality types present in GPU programs and show that no state-of-the-art NUMA-GPU system can exploit them all. We propose LADM, which uses static index analysis to inform runtime data placement, threadblock scheduling, and remote caching decisions by exploiting a new logical abstraction called the *GPU datablock*.
- We leverage this automatic analysis to perform threadblock and datablock co-placement within hierarchical GPUs. By pre-calculating an optimized data layout in the compiler, LADM can orchestrate prefetching that negates on-demand page-faulting effects and adjust the threadblock schedule based on dynamic data structure sizes.
- Building on our program analysis, we architect a novel compiler-informed cache organization that selectively inserts requests into each L2 cache partition based on the memory request's origin relative to the data's home node and its likelihood for reuse. By understanding the expected reuse patterns for datablocks, LADM's cache hierarchy minimizes both inter-GPU and inter-chiplet bandwidth, the primary factor influencing the scalability of future GPUs.

II. Motivation and Background

Figure 1 depicts what next-generation exascale GPU compute accelerators may look like in the future. Within a single GPU, monolithic GPU dies will be subdivided into disaggregated chiplets, where each chiplet is composed of a group of SMs associated with its own local High Bandwidth Memory (HBM) and hardware thread block scheduler. Several different ways to connect these chiplets have been proposed. Interposer-based through-silicon vias (TSVs), similar to those proposed in AMD's future exascale node [53], [79], high rate signaling through organic substrate-based connections similar to NVIDIA's Ground Reference Signaling (GRS) [5], [65], [70], Intel's Embedded Multi-die Interconnect Bridge (EMIB) [27] or waferscale integration using Silicon Interconnection Fabric (Si-IF) [62], [63] are all possible solutions. While these links may provide high enough bandwidth to alleviate the NUMA-GPU performance penalty [79], such solutions are likely to be expensive and hard to manufacture. These same technologies could be conservatively applied to provide cost-effective, bandwidth restricted interconnections [5]. Architectural and software techniques are necessary to reduce off-chiplet traffic and mitigate the performance loss due to bandwidth constraints. While reducing off-chiplet traffic across exotic high-speed connections may not lead to performance improvement, LADM still improves overall energy efficiency by minimizing data movement among the chiplets [6].

NUMA-GPU designs will not only exist within on-package solutions. With the arrival of high bandwidth switch-connected GPUs such as NVIDIA's DGX-2 and NVLink [55], [74] interconnect, aggregating multiple discrete GPUs into a large virtual GPU is now being considered [51]. Because these GPUs may operate as both individual GPUs and in aggregate (as a single GPU), this aggregation must be done with more limited hardware support, primarily by the GPU runtime software. In addition, the type of hierarchical NUMA present in Figure 1 must be accounted for both page placement and threadblock scheduling. Previous, hierarchy-oblivious approaches [5], [36], [51] to NUMA-GPU should be applied recursively, accounting for the fact that chiplets on the same discrete GPU will have greater peer-to-peer bandwidth than chiplets that reside on different GPUs.

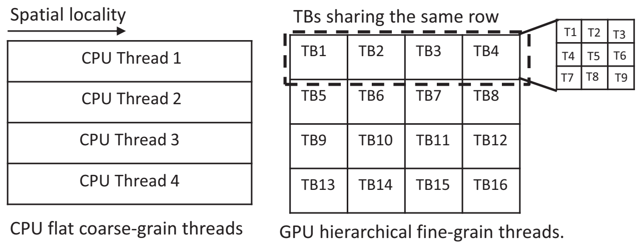


Fig. 2: OpenMP vs CUDA thread mapping for *sgemm* [75].

A. NUMA Locality in CPUs vs GPUs

Parallel programming on NUMA multi-processor and on-chip multi-core CPU systems is a well studied problem. Many proposals attempt to minimize NUMA memory access latency transparently through software memory allocation and migration policies [12], [20], [22], [26], [69] or thread-to-core mapping [11], [19], [45], [78] techniques. Most of these works are reactive solutions, wherein they detect locality and congestion at runtime, then they perform page migration, replication and thread-clustering based on runtime observations. Although reactive systems can be applied to GPUs, they introduce a substantial performance penalty that can outweigh the benefits. For example, data replication increases memory capacity pressure, which is a scarce resource in contemporary GPUs [84]. First-touch page placement policy can reduce performance significantly, stalling SMs for 20-50 microseconds [85]. Furthermore, the sheer number of threads in flight makes reactive work re-distribution intractable, and the cost of page migration in bandwidth-limited GPU workloads is high [1], [7]. These all motivate a *proactive*, prediction-based solution based on static program analysis.

The GPU programming model introduces new challenges in the design space for NUMA systems that did not exist in traditional NUMA-based multi-processor systems. Since GPUs are programmed using a huge number of threads, the work done by each individual thread is small, resulting in far more thread scheduling decisions. To manage all these threads, they are grouped into a multi-dimensional grid of threadblocks, where each block operates on one multi-dimensional chunk of the data structure. This is in contrast to the coarse-grain nature of CPU threads, where far fewer threads do much more work each. Figure 2 illustrates how threads in CPUs and GPUs typically access a row-based data structure with an example from the Parboil benchmark suite [75]. In the coarse-grained CPU case, each thread has significant spatial locality and static index analysis of the single-threaded code can easily determine the access pattern of each thread. In the fine-grained GPU case, the same per-thread analysis can be applied. However, the reach of each individual thread is minimal, as each thread will access very few (or even just one) elements in the row. In order to capture the locality pattern in GPUs, an inter-thread analysis must be performed, to account for both the hierarchy of the grid (i.e. the presence of threadblocks) and the dimensionality of the thread grid. This type of inter-thread analysis is what we propose in LADM.

TABLE I: LADM vs state-of-the-art techniques

	Batch+FT [5]	Kernel-wide [51]	CODA [36]	LD / TB clustering [43], [76], [80]	LADM
Page placement policy	First-Touch	Kernel-wide chunks	Sub-page round robin	Hand-tuned APIs	LASP
Threadblock scheduling policy	Static batched round robin	Kernel-wide chunks	Alignment-aware batched round robin	Hand-tuned APIs	LASP
Page alignment	✗	✓	✓	✓	✓
Threadblock-stride aware	✓	✗	✗	✓	✓
Row sharing	✗	✓	✗	✓	✓
Col sharing	✗	✗	✗	✓	✓
Adjacent locality (stencil)	✗	✓	✗	✓	✓
Intra-thread loc	✓	✗	✓	✗	✓
Input size aware	✗	✗	✗	✓	✓
Overhead	+First-touch page faulting	-	+Hardware for sub-pages	+APIs	-
Transparency	✓	✓	✓	✗	✓
Hierarchical-aware	✗	✗	✗	✗	✓

In addition, there is little intra-thread locality in highly-optimized GPU applications with regular access patterns. Instead of repeatedly accessing values on the same cache line, GPU programs typically access values on the same line in different coalesced threads. Optimized GPU programs also make extensive use of a scratchpad memory, which effectively prevents a large portion of global data from being accessed more than once. The end result is that there is very little global data reuse in GPU programs, making the initial decision on where a page should be placed extremely important, since temporal locality in upper-level caches is rare.

B. Existing NUMA-GPU Optimizations

In this section, we qualitatively and quantitatively study state-of-the-art NUMA-GPU page placement and threadblock scheduling techniques for both MCM [5] and Multi-GPU [36], [51], [80] configurations, teasing out the fundamental properties they exploit and highlighting opportunities they miss.

The first work on chiplet-based GPUs by Arunkumar et al. [5] optimizes per-chiplet locality through a synergistic approach of statically batching threadblocks and performing a reactive first-touch (Batch+FT) page placement policy. While optimizing for locality, Batch+FT relies on the GPU unified virtual memory (UVM) system to page fault data to the chiplet on which the data is first accessed. While effective for improving data locality, relying on the UVM first-touch page placement policy can introduce a substantial performance penalty as data must be page-faulted into GPU memory from system memory, stalling SMs for 20-50 microseconds [85]. An ideal GPU locality management system should *make an educated guess* about threadblock and data locality before execution begins, so that data and computation can *proactively* be pushed to the right location whenever possible.

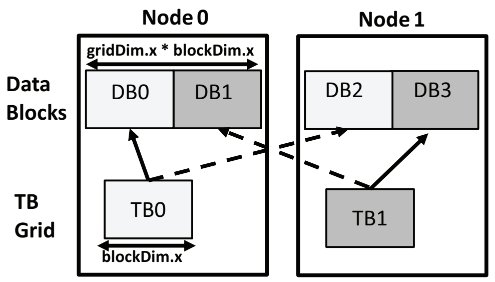


Fig. 3: Behavior of kernel-wide partitioning in a 2-node system with 2 threadblocks that access a 4 datablock data structure with a stride of one datablock.

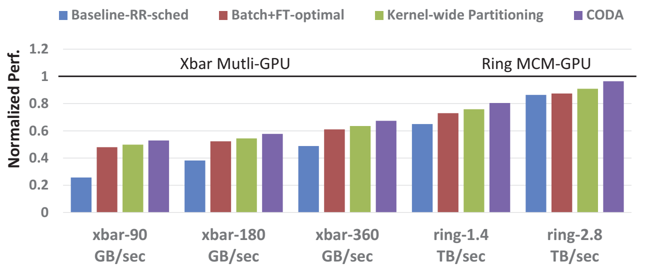


Fig. 4: Bandwidth sensitivity analysis of state-of-the-art techniques normalized to a hypothetical monolithic GPU with the same number of SMs. Performance is averaged over the applications listed in Section IV-A.

The second work, by Milic et al. [51], focuses on multiple discrete GPUs. Their solution partitions both the kernels grid and each data structure (i.e., every call to *cudaMalloc*) into N contiguous chunks, where N is the number of GPUs. Each chunk of data and threadblocks are then assigned to each respective GPU. We call this technique kernel-wide grid and data partitioning and it is pictured in Figure 3.

The third class of work, by Vijaykumar et al. [80] and Sun et al. [76], propose a flexible and portable software interface, called the Locality Descriptor (LD), to explicitly express data locality with a series of highly-specific APIs. Similarly, Cabezas et al. [15] rely on programmer input to shape the data placement in the program. Locality-aware threadblock clustering with code annotations was also proposed in a single GPU context [43]. Our proposed research seeks to marry the locality description benefits of these manual APIs with the transparency benefits of the locality-agnostic techniques.

Finally, the most closely related work to our own is CODA by Kim et al. [36]. CODA is a compiler-assisted index analysis framework that calculates the data accessed by each threadblock to ensure page alignment. CODA applies round-robin page and sub-page interleaving and launches static batches of threadblocks that share the same sub-page on the same node. However, CODA is only able to exploit one specific locality pattern and requires hardware changes to support sub-page address mapping.

Table I breaks down a number of common access patterns found in contemporary GPU workloads and details which prior work is able to capture them, preventing off-chip traffic. The first pattern that is *Page alignment*. If the data-placement mechanism and threadblock scheduler are unaware of how much data is accessed by a threadblock, and round-robin threadblocks among chiplets, they may not place contiguous threadblocks accessing the same page on the same chiplet. The Batch+FT scheduler launches a statically-defined batch of threadblocks (4-8 threadblocks) in a loose round-robin fashion across chiplets, in an attempt to load-balance the workload. Without knowing how big the threadblock batch should be, unnecessary off-chip accesses may occur. On the other hand, CODA is explicitly-designed to capture this pattern and to ensure that the batches are page-aligned. Kernel-wide partitioning captures this pattern as well by avoiding a round-robin scheduler and launch threadblocks in coarse-grained chunks.

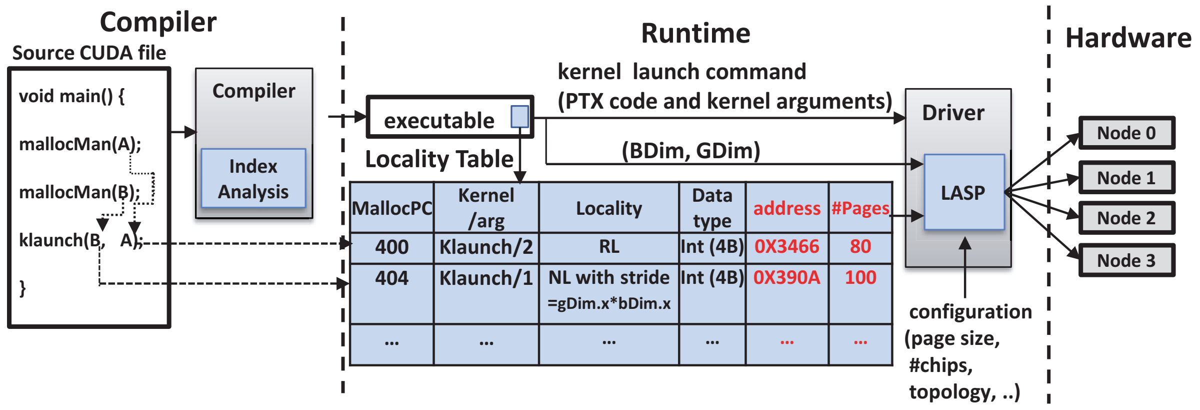
The second pattern is *Threadblock-stride aware*. In this pattern, threadblocks access one chunk of data, then jump with a constant stride to read another chunk of data. Batch+FT is able to capture this pattern since the first-touch page placement policy will bring the page to the correct node. Kernel-wide partitioning and CODA are unaware of this strided behavior and will generate off-chip traffic if the stride does not accidentally match their partitioning. Figure 3 depicts an example of how kernel-wide partitioning works in a simple strided accesses scenario where the stride is misaligned with the system configuration, resulting in 50% off-chip accesses.

The next two patterns: *Row sharing* and *Column sharing* occur when a row or a column of threadblocks in a two-dimensional grid access the same row or column of a data structure. None of the prior techniques account for these sharing patterns, but kernel-wide partitioning is able to exploit row sharing by dividing both the grid and data structures into contiguous row-wise chunks.

The *Adjacency locality* pattern is commonly found in stencil applications where adjacent threads share data on their boundaries. The round-robin nature of Batch+FT and CODA create memory traffic at the edge of every threadblock batch. Since kernel-wide partitioning is scheduled in large chunks, the number of grid cuts is minimized and so is the off-chip traffic in stencil workloads.

The *Intra-thread locality* pattern is often found in irregular workloads that have significant spatial locality in a single thread [67]. Batch+FT naturally captures this locality by moving pages to where they are first accessed. Finally, none of the existing techniques account for the size of a program's data structures and are hence input-size unaware. We explicitly design LADM to exploit all of these characteristics, which we describe in more detail in Section III.

To demonstrate the relative performance of prior techniques across a variety of integration domains, we implement and evaluate several pieces of state-of-the-art work [5], [36], [51], along with a baseline round-robin placement and scheduling mechanism adopted from [79]. Figure 4 shows the average performance of a four GPU NUMA system with 64 SMs on each node for each evaluated technique. All values are normalized to the performance of a hypothetical monolithic GPU (where there is no NUMA access penalty to remote memories) with the same number of SMs (256).



MallocPC	Kernel /arg	Locality	Data type	address	#Pages
400	Klaunch/2	RL	Int (4B)	0X3466	80
404	Klaunch/1	NL with stride =gDim.x*bDim.x	Int (4B)	0X390A	100
...	...	...	...	...	...

Fig. 5: End-to-end overview of our proposed Locality-Aware Data Management System. In the locality table: MallocPC, the kernel/arg tuple, the locality type and data type are filled statically, whereas memory address and #pages are filled dynamically.

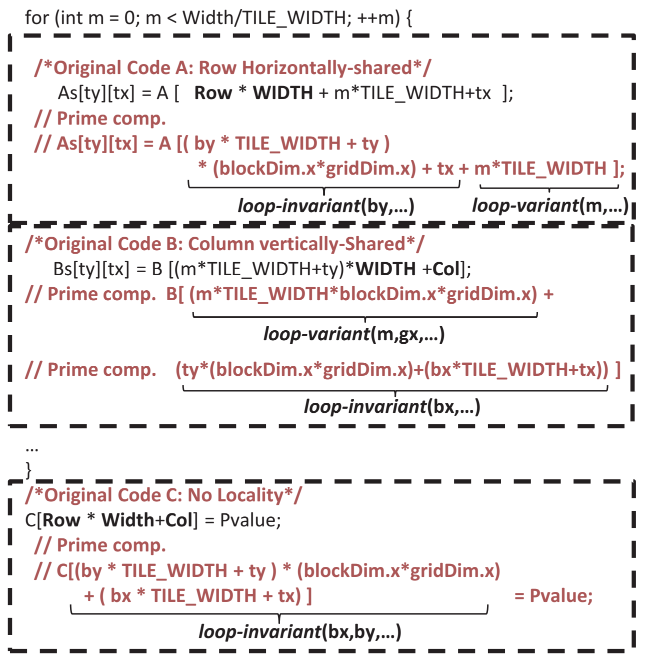
To understand the effect topology and interconnect has on their relative performance, we simulate two different interconnection configurations connecting the four GPU nodes. First, a crossbar inter-GPU switch, similar to an NVSwitch [56], with different link bandwidths. Second, a hypothetical high-speed bi-directional ring with 1.4 and 2.8 TB/sec per-GPU to model an MCM-like topology [5]. We model optimal on-demand paging (Batch+FT-optimal) assuming page faults have zero overhead. Ideally, we would like to achieve the same monolithic chip performance with the cheapest possible interconnection. We observe that uniformly, CODA outperforms Batch+FT-optimal and kernel-wide partitioning, thanks to its alignment-aware static index analysis. Yet CODA only achieves 52% and 80% of the monolithic GPU for the xbar-90 GB/sec and ring-1.4 TB/sec configurations. This implies that while CODA should be considered state-of-the-art versus other policies, there still remains significant room for improvement.

III. Locality-Aware Data Management

The goal of Locality-Aware Data Management is to optimize NUMA-GPU page placement, threadblock scheduling, and GPU cache management based on access patterns derived from a new threadblock-aware compiler pass with unmodified applications.

A. LADM System Design

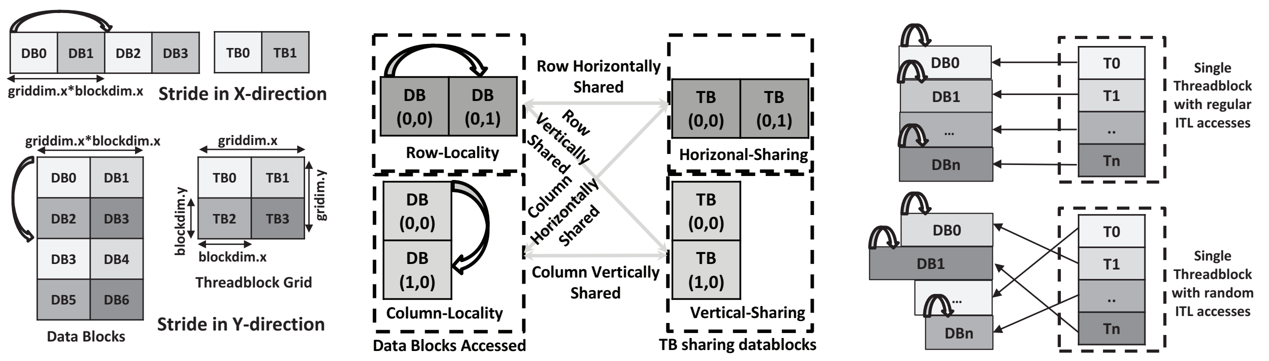
Figure 5 depicts an end-to-end overview of our proposed LADM mechanism. First, we perform a symbolic off-line index analysis on CUDA source code during the compilation process, detailed in Sections III-B and III-C. Our analysis generates a *locality table*, which is embedded in the executable. There is one row in the table for every access to a global data pointer passed to every *__global__* CUDA function. The compiler fills the locality table with the detected locality type, data type and the MallocPC of the associated *cudaMallocManaged* call from the CPU that allocated this data structure. The MallocPC is used to connect the symbolic compile-time information with dynamic runtime parameters. At runtime, each *cudaMallocManaged* call inserts the number of allocated



```
for (int m = 0; m < Width/TILE_WIDTH; ++m) {
  /*Original Code A: Row Horizontally-shared*/
    As[ty][tx] = A [  Row * WIDTH + m*TILE_WIDTH+tx  ];
  // Prime comp.
  // As[ty][tx] = A [( by * TILE_WIDTH + ty )
                 * (blockDim.x*gridDim.x) + tx + m*TILE_WIDTH ];
                   loop-invariant(by,...)      loop-variant(m,...)
  /*Original Code B: Column vertically-Shared*/
    Bs[ty][tx] = B [(m*TILE_WIDTH+ty)*WIDTH +Col];
  // Prime comp.  B[ (m*TILE_WIDTH*blockDim.x*gridDim.x) +
                     loop-variant(m,gx,...)
  // Prime comp.   (ty*(blockDim.x*gridDim.x)+(bx*TILE_WIDTH+tx)) ]
                     loop-invariant(bx,...)
  ...
}
/*Original Code C: No Locality*/
C[Row * Width+Col] = Pvalue;
 // Prime comp.
 // C[(by * TILE_WIDTH + ty ) * (blockDim.x*gridDim.x)
     + ( bx * TILE_WIDTH + tx) ]                    = Pvalue;
          loop-invariant(bx,by,...)
```

Fig. 6: Matrix multiplication indices analysis

pages and address information into the kernel/argument tuples associated with this call. The mapping between *cudaMallocManaged* calls and kernel launch arguments is provided by the CPU compiler. Fortunately, the way GPU programs are written today, *cudaMallocManaged(ptr);* followed by *kernel_launch(ptr);* almost always occurs. This allows us to statically determine which *cudaMallocManaged* is associated with which kernel argument. We use traditional pointer aliasing analysis to determine the safety of this argument binding. If the static analysis is not successful, then LADM has no choice but to use a default policy for that particular call operation. Finally, on every kernel launch, the Locality-Aware Scheduling and Placement (LASP) component, described in Section III-D, reads the locality table and decides the proper scheduling policy, data placement and cache strategy to reduce off-chip traffic and mitigate NUMA impact.



(a) No datablock-locality with stride in x and y direction.

(b) Row or column datablock-locality with horizontal or vertical threadblock sharing.

(c) Intra-thread spatial locaity with regular-sized datablocks (top image) and irregular-sized random accesses (bottom image).

Fig. 7: Common locality types found in GPU workloads. Arrows indicate threadblock motion and datablocks are shaded based on the shade of the threadblock (TB) that accesses them.

B. Threadblock-centric Locality Patterns

When work is launched to a transparent NUMA-GPU system, threads are assigned to GPUs at the threadblock granularity [51]. To create a 1:1 mapping between data placement and threadblock scheduling, we define a *datablock* as the region of data accessed by a threadblock on each iteration of the kernel's outermost loop. For example, consider the simplified kernel code for a dense $A \times B = C$ matrix-matrix multiply listed in Figure 6. Each thread computes one output element of the C matrix, striding through a row of A and a column of B on each loop iteration. Across an entire threadblock, each iteration of the loop will access a square region of both matrix A and B. The data accessed by the threadblock on this loop iteration is what we call a datablock. The datablock's size is directly related to the size of the data type being accessed by each thread, the dimensions of the threadblock and the components that make up the array index. Using this taxonomy, it is possible to classify the way threadblocks access data structures into one of three categories: No datablock-locality, row/column-locality, and intra-thread locality. Figure 7 plots a visual representation of our datablock-locality definitions.

Figure 7a shows the No datablock-Locality (NL) case, where threadblocks do not access the same datablocks. A simple example of an application with no datablock-locality is $C = A + B$ vector addition, where each threadblock accesses a contiguous region of A and B with no reuse or sharing. Stencil applications are another example where there is no locality among threadblocks, except among the adjacent elements. Applications that have no datablock-locality come in two forms. In the first, the kernel does not contain any loops, each datablock is computed on and then discarded. In the second, the kernel has loops and on each iteration of the loop, the threadblock strides across the data structure to another non-shared datablock. We call this movement among datablocks, *threadblock motion*. As shown in Figure 7a, threadblocks can access exclusive datablocks with a stride in either the x or y direction. Strided accesses frequently exist in GPGPU workloads when kernels increase the work in each thread by launching fewer threads than elements in the input data structures. Increasing work granularity per thread is a widely used optimization in CUDA programs to reduce thread initialization overhead and redundant computation [40].

It is also common for groups of threadblocks to share groups of datablocks. Figure 7b illustrates a sharing pattern where datablocks are accessed in either the row or column directions, by either a row or a column of threadblocks from the thread grid. For example, consider the A matrix in $A \times B = C$ matrix-matrix multiplication. A row of datablocks is shared among horizontal threadblocks. The accesses to the B matrix in matrix-matrix multiply demonstrate a different pattern. Here, a column of datablocks will be shared among vertical threadblocks. Two other possible combinations occur when rows are vertically shared and when columns are horizontally shared. We call workloads that have Row and/or Column Locality RCL workloads.

Figure 7c demonstrates the last type of common locality present in GPU workloads: Intra-Thread Locality (ITL). For these data structures, individual threads exhibit spatial locality across strided, regularly-sized datablocks or data-dependent, irregularly-sized datablocks. A number of prior works have shown that these applications can have significant intra-thread locality [29], [30], [34], [67], [68], making shared-cache interference a significant problem.

C. Static Locality and Sharing Detection

We make the observation that static compiler analysis can make reasonable predictions about which of these three common locality patterns exist in GPU programs. We show that each locality and sharing pattern can be predicted based on an index analysis of accesses to each global data structure. The core idea is to extend traditional CPU index analysis [3] to be aware of threadblock-level definitions of parallelism. This index analysis is performed on the CUDA source code.

For regular kernels, there are two key elements we seek to determine from the static analysis: (1) the direction the

TABLE II: Index analysis and taken actions. *bx = blockIdx.x, by = blockIdx.y, gDimx = gridDim.x*, m is an induction variable. For the $loopInvariant$ function, if one of bx or by is not listed, then none of the terms in the equation contain that variable. For the $loopVariant$ function, if $gDimx$ is not listed, then none of the terms in the equation contain $gDimx$.

Locality Types	Index Equation	Fig	Dims	Threadblock Scheduling	Data Placement	Cache Policy
1: No datablock-locality	$loopInvariant(bx, by, ...) + stride \times m \ \forall \ stride \neq 1$	7a	1D/2D	Align-aware	Stride-aware	RTWICE
2: Row-locality, horizontally shared	$loopInvariant(by, ...) + loopVariant(m, ...)$	7b	2D	Row-binding	Row-based	RTWICE
3: Column-locality, horizontally shared	$loopInvariant(bx, ...) + loopVariant(m, ...)$	7b	2D	Col-binding	Row-based	RTWICE
4: Row-locality, vertically shared	$loopInvariant(by, ...) + loopVariant(m, gDimx, ...)$	7b	2D	Row-binding	Col-based	RTWICE
5: Column locality, vertically shared	$loopInvariant(bx, ...) + loopVariant(m, gDimx, ...)$	7b	2D	Col-binding	Col-based	RTWICE
6: Intra-thread locality	$loopVariant(m) = m$	7c	1D	Kernel-wide	Kernel-wide	RONCE
7: Unclassified	none of the above	N/A	1D/2D	Kernel-wide	Kernel-wide	RTWICE

threadblock moves on each loop iteration (i.e., threadblock motion), and (2) which threadblocks in the grid share the same datablocks. To determine these two variables, our source analysis begins by identifying global array accesses and expanding their index equations such that they are composed of only prime variables. We consider the following variables prime: thread IDs, block IDs, grid dims, block dims, induction variables (i.e., the loop counter) and constants. Using these variables, we then perform the analysis detailed in Algorithm 1 to classify the access, if possible.

Table II details the general index equations that are matched by our static analysis to determine which type of locality is predicted for each global array access. The compiler will attempt to match each array access to one of these 6 mutually exclusive types using Algorithm 1. The basic idea behind our index analysis is to break the index in two groups of terms. One group contains all the terms dependent on an induction variable, which we call the *loop-variant group*. The second group is composed of all the terms that are not dependent on the induction variable, which we call the *loop-invariant group*. That is, all the terms that are multiplied by an induction variable are combined in the loop-variant group, and all the remaining terms are collected in loop-invariant group. The loop-variant group determines the threadblock motion of the access, i.e., do threadblocks move horizontally or vertically through the data structure and how far do they move. Conversely, the loop-invariant terms do not change on each loop iteration and are used to determine which datablock each threadblock starts at.

To illustrate how global array-based data structures are typically accessed in GPU programs, we refer to the matrix multiplication example in Figure 6. The comments below the accesses to matrix A, B and C decompose the Row, Col and $WIDTH$ variables into the prime components using backward substitution and algebraic simplification. Once the access has been broken down into invariant and variant components, the compiler determines which key variables the groups are dependent on and detects the locality type using Algorithm 1.

The classification in Algorithm 1 begins by testing the special-case that the only term in the loop-variant group is the induction variable multiplied by 1. If this is the case, then we assume the access has intra-thread locality and classify the access as ITL (row 6 in Table II). If that test fails, the algorithm tests if the access has no locality, by checking if the loop-invariant terms are dependent on both bx and by (for 2D threadblocks) or just bx (for 1D threadblocks). If so, we predict the access has no locality and then derive the stride by dividing the loop-variant term by m, classifying the access as row 1 of Table II. The access to C in Figure 6 is an example of a no locality access. If neither of these first checks are true, then we search for the 4 sharing patterns in Figure 7b. If the loop-invariant term depends on by and not bx, then the starting datablock of all the threadblocks with the same by (i.e., all threadblocks in the same row) will be the same. The same is true for a dependence on bx only, except threadblocks in the same grid column start in the same place.

After the sharing pattern is determined, the loop-variant terms are checked to determine the threadblock motion direction. If the loop-variant terms depend on $gDimx$, then we predict a whole row is being skipped on each iteration and that the threadblock motion is in the column direction, otherwise we predict that threadblocks move across a row of the data structure so long as a loop-variant term exists. Based on which combination of sharing and motion is detected, one of rows 2 through 5 in Table II is selected for accesses in 2D threadblocks. The A access in Figure 6 is an example of row threadblock motion, shared across threadblocks in a grid row and the B access illustrates column threadblock motion, shared across columns of threadblocks. If the array index does not match one of the locality types in Table II, for

Algorithm 1 Access classification algorithm.

```
 1: if $loopVariant(m, ...) = m$ then
 2:     access = ITL;
 3: else if $loopInvariant(bx, by, ...)$ then
 4:     access = NoLocality;
 5:     stride = $loopVariant(m, ...)/m$;
 6: else if 2D Blocks then
 7:     if $loopInvariant(by, ...)$ then
 8:         access = ThreadblockRowShares;
 9:     else if $loopInvariant(bx, ...)$ then
10:         access = ThreadblockColsShares;
11:     if $loopVariant(m, gDimx, ...)$ then
12:         access $|=$ ColumnThreadblockMotion;
13:         stride = $loopVariant(m, gDimx, ...)/m$;
14:     else if $loopVariant(m, ...)$ then
15:         access $|=$ RowThreadblockMotion;
```

example the array index contains a data-dependent component with no intra-thread locality (i.e., $X[Y[tid]]$), we leave it as unclassified (row 7 in Table II) and the default placement policy is used.

After classifying each of the global array accesses in a kernel to one of the rows in Table II, the compiler's work is done. The final classification of each symbol is embedded into the binary and used by the runtime system, described in the next section, to determine appropriate placement and threadblock scheduling.

D. Locality-Aware Scheduling and Page Placement

LASP is LADM's runtime system that implements page placement and threadblock scheduling based on locality patterns identified by the compiler.

1) LASP Data Placement: Based on the locality pattern detected for each data structure, LASP places data using the following methods.

Stride-aware placement (Row 1 in Table II): To avoid off-chip traffic from strided accesses, LASP must ensure that all the datablocks accessed by a particular threadblock map to the same node. Using the stride information provided by the compiler analysis, we determine which pages need to be co-located on a given node. We interleave the pages in a round robin fashion using the page granularity given by Equation 1. Note that, in order to determine which threadblock maps to the next node we need to know what decision the threadblock scheduler will make. Here we assume that the aligned scheduler described in Section III-D2 will be used.

$$InterleavingGranularity = \left\lceil \frac{strideSize}{\#nodes} \right\rceil^{pageSize} \tag{1}$$

Row- and column-based placement (Rows 2-5): LASP uses row- or column-based page placement to put a whole row or column of data on the same node. For example, when rows are horizontally shared, row-based placement is used along with the row-binding scheduler (Section III-D2). When column-based locality is horizontally shared, column-based placement is employed with row-binding scheduler. In column-based placement, we interleave data over nodes in a round-robin fashion using Equation 1 where stride size is the data structure's row width.

Kernel-wide data partitioning (Rows 6 and 7): If a data structure has intra-thread locality or unclassified irregular accesses, such as graph traversal workloads. In this case, we fall back to the default data placement strategy of kernel-wide partitioning that has experimentally shown good performance for workloads that use CSR data or perform stencil operations. In these difficult to predict workloads, LADM relies on our caching mechanism described in Section III-E to further mitigate off-chip accesses by improving the L2 hit rate.

Timing of page placement and prefetching opportunities: LASP works with UVM, relieving the programmer from the burden of manually copying memory to the device. However, unlike traditional first-touch page placement, LASP makes a prediction about where every page should be placed. The pages for the data structure can be copied to the correct node as soon as the first kernel that uses a data structure is launched. We must wait until kernel launch time in order to determine the threadblock and grid sizes, which are required to compute the datablock size and strides. However, if the compiler can statically determine what the size of the first kernel launch will be, copying could potentially be started before kernel launch. It is possible that the placement derived from the first kernel launch is sub-optimal for subsequent kernel launches. Despite this potential disagreement, we find that the access pattern from the first kernel launch is often consistent with subsequent kernel launches. We leave the exploration of inter-kernel data transformations as future work.

2) LASP Threadblock Scheduling: Based on the locality pattern detected for each data structure, LASP schedules threadblocks using the following methods.

Alignment-aware and kernel-wide scheduler (Rows 1, 6 and 7): In the absence of any strong row or column data affinity, the scheduler attempts to load balance the work in a page-aligned fashion. To avoid the issue of page-misalignment suffered by Batch+FT [5], we can predict what the minimum threadblock batch size by using Equation 2, where dividing the page size by the datablock size tells us the minimum number of consecutive threadblocks (*MinTBBatch*) that should be assigned to each node to avoid misaligning datablocks and threadblocks.

$$MinTBBatch = \frac{pageSize}{datablockSize} \tag{2}$$

The minimum batch size will change depending on the page size and kernel arguments, since the datablock size will vary between kernels. As a result, the static batch size used in [5] will suffer when the datablocks are mis-aligned. In workloads with no locality, we have found that the datablock size is often equal to $bx \times primitiveSize$, where primitive size is 4 or 8 bytes (i.e., float versus double). Unlike CODA [36], which changes the physical page interleaving granularity and proposes fine-grained sub-page interleaving to ensure alignment, LASP keeps the page interleaving as-is and applies dynamic batch sizing using Equation 2 to maintain data alignment. The scheduler interleaving granularity can be any multiple of the batch size (i.e., $n \times MinTBBatch, n \geq 1$). In kernel-wide scheduling, n is the maximum possible value, in which we partition the threadblock grid into N contiguous chunks of threadblocks, where N is the number of GPU nodes.

Row- and Column-binding scheduler (Rows 2-5): The row-binding scheduler will place all threadblocks from the same row on the same node such that row-level datablock-locality is exploited. For a grid with more rows than GPU nodes, we place contiguous rows of threadblocks on each node. Similar to the row-binding scheduler, the column-binding scheduler assigns all threadblocks from the same column of the grid to the same node in order to exploit column-level datablock-locality.

Hierarchical-aware Scheduling: To exploit the fact that chiplets on the same discrete GPU will have greater bandwidth than chiplets that reside on different GPUs, the hardware and runtime system must coordinate to expose the hierarchically clustered locality domains of the underlying hardware to LASP. This allows LASP to assign adjacent threadblocks to the physically co-located chiplets on the same GPU, before moving to the next GPU. LASP employs a hierarchical affinity round-robin scheduler wherein we assign a chuck of contiguous rows or columns of threadblocks to a discrete GPU, then the assigned threadblocks are scheduled in a round-robin fashion among the chiplets within the GPU.

Data structure Locality Disagreements: Some kernels will access multiple data structures in different ways. When this happens, each structure will be placed in the way we predict is optimal, but there is only one threadblock scheduler we can select for a particular kernel. For example, in the matrix multiply example in Figure 6, the placement of the A matrix favors a row-binding threadblock scheduler, whereas the placement of B favors column-binding scheduling. Since it is not possible to give each data structure the scheduler that suits it best, we must pick a winner. To break the tie, we favor the scheduling policy that is associated with the larger data structure, because it will intuitively have a bigger effect on off-chip accesses, whereas smaller, frequently accessed data structures have a much greater chance of residing and hitting in the requesting node's L2. So, in our matrix multiply example, if matrix A is larger than B, we opt for a row-binding scheduling and rely on the L2 cache to reduce the off-chip traffic of the smaller matrix B. Unequal matrix sizes are commonly found in deep learning applications where a small matrix of images is multiplied by a large matrix of neuron weights.

E. Compiler-assisted Remote Request Bypassing

LASP is an efficient solution for regular workloads. However, there are additional opportunities presented in NUMA-GPU when the workloads are irregular and have intra-thread locality. Predicting the data-dependent access patterns of these irregular applications is not possible at compile time. Therefore, these irregular workloads, shown in Figure 7c, rely heavily on L2 caches to reduce off-chip traffic and mitigate NUMA issues [51]. We seek to improve these workloads via an intelligent cache management technique we call *cache-remote-once* that makes better use of cache in NUMA-GPUs.

Figure 8 illustrates the key idea of *cache-remote-once* (RONCE). In our baseline, the L2 cache is shared between local and remote traffic, similar to the dynamic shared L2 cache proposed in [51]. That is, the remote request checks the local L2 first, and if it is a miss, the request is redirected to the correct home node through the inter-chip connection. In this scenario, remote read requests are cached twice, once at the L2 cache of the home GPU and another time at the L2 cache of the GPU that sends the request. In fact, *cache-remote-twice* (RTWICE) can be beneficial in RCL workloads that count on the remote cache to minimize the NUMA effects on the victim data structure. In these workloads, remote requests are accessed by multiple SMs across the GPUs (i.e. inter-GPU locality), as shown by the solid line in Figure 8. However, workloads with intra-thread locality, caching requests twice is a waste of cache resources if the line is only accessed by one warp and one SM in the requesting GPU, as depicted by the dashed line in Figure 8. Therefore, there is no need to cache the request at the home GPU, since it may interfere with local traffic. To this end, we propose compiler-assisted remote request bypassing (CRB). In CRB, we use our compiler index analysis to determine the locality type found in the program (i.e., RCL vs ITL) and enable the RONCE bypassing policy only in ITL workloads, since our experiments show that applying RONCE for RCL may hurt the performance.

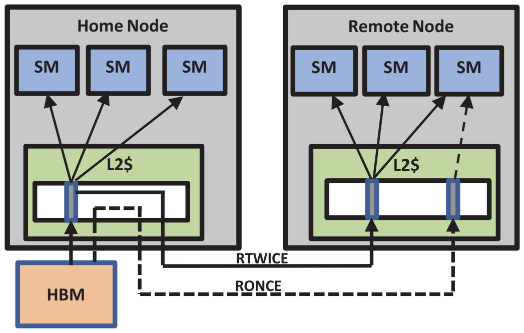


Fig. 8: llustration of existing NUMA caching policy *cache-remote-twice* (the solid line) and our proposed *cache-remote-once* cache management strategy (the dashed line)

IV. Experimental Methodology

A. Simulation Methodology

To evaluate LADM we use GPGPU-Sim version 4.0 with the recent memory system improvements from Accel-Sim simulation framework [35]. We have modified the simulator in order to model a hierarchical multi-GPU design with four GPUs connected via a switch, where each GPU is composed of four chiplets as depicted in Figure 1. The configuration parameters used in our system are listed in Table III and are similar to prior works [5], [51], [66]. We have implemented the dynamically shared L2 multi-GPU cache coherence proposal from Milic et al. [51] with cache insertion policy changes that have been described in Section III-E.

We have implemented the NUMA-GPU analysis proposed in the CODA system [36] and have also extended it to be aware of the GPU's hierarchical nature (H-CODA). We consider the offline profiling proposed in CODA to be an orthogonal approach to static analysis, thus we did not apply it to any evaluated technique. In all results, H-CODA is operating on top of the baseline cache coherence system. The original

TABLE III: Multi-GPU Configuration

#GPUs	4 GPUs, 4 chiplets per GPU
#SMs	256 SMs (64 SMs per GPU, 16 SMs per chiplet)
SM configuration	Volta-like SM [35], 64 warps, 4 warp scheds, 64KB shared memory, 64KB L1 cache, 1.4 GHZ
L2 cache	16MB (1MB per GPU chiplet), 256 banks, Dynamic shared L2 with remote caching [51]
Intra-Chiplet Connect	16x16 crossbar, total BW=720 GB/s
Inter-Chiplet Connect	bi-directional ring, 720 GB/s per GPU
Inter-GPU Connect	4x4 crossbar, 180 GB/s per link, bi-directional
Monolithic Interconnect	256x256 crossbar, total BW=11.2 TB/s
Memory BW	180 GB/s per chiplet, 720 GB/s per GPU

CODA work did not utilize any remote caching capability in hardware, but as shown in [51], utilizing remote caching in NUMA-GPUs significantly improves performance scalability on a wide range of workloads. In particular, our experiments show that enabling remote caching improves performance of general matrix multiplication (GEMM) operations by $4.8\times$ on average, reducing off-chip traffic by $4\times$.

B. Workload Selection and Characterization

We run LADM on a selection of 53 scalable workloads from Rodinia 3.1 [17], CUDA SDK [57], Parboil [75], Lonestar [60] and Pannotia [16]. In addition, we include a variety of deep learning matrix math operations in which we exploit intra-layer model parallelism by running GEMM operation on multiple GPU nodes as practiced in large model training frameworks [72]. We used the optimized *sgemm* from [57], [75] as our reference implementation of GEMM and we extract layer and matrices dimensions from several popular DL networks [4], [25], [77]. Like prior work [5], [51], we initially pare a broader set of 53 workloads from all the benchmarks suites listed above and select only those workloads that have enough parallelism to scale-up on our simulated multi-GPU system. Of these 27 scalable benchmarks, LADM's locality detector places 24 into identifiable patterns and places 3 into the unclassified category. Table IV lists the workloads used in this study, along with their detected locality types, scheduler decision, number of launched threadblocks, input size and L2 sector misses per kilo warp instructions (MPKI). It is worth noting that a workload can contain more than one locality type and kernel. In the table, we list the dominant locality type found in the dominant kernel.

C. Hardware Validation of LASP Principles

Like prior work, the LADM system relies on co-designed hardware and software features to maximize locality and performance. Features like remote caching, inter-GPU cache coherence, programmatically available hierarchical locality cluster information, and the capability to perform fine grained data placement among chiplets in GPUs are not present in GPUs that are available to researchers today. However, the compiler analysis provided by LASP allows us to test the software based placement of thread and data blocks on real GPUs today. We hand implemented LASP for the RCL machine learning workloads listed in Table IV when running on a 4-GPU cluster within an NVIDIA DGX-1 system [74].

TABLE IV: Workloads used to evaluate LADM in simulation.

Workload	Locality Type	Scheduler Decision	TB Dim	Input Size	Launched TBs	L2 MPKI
VecAdd [57]	NL	Align-aware	(128,1)	60 MB	10240	570
SRAD [17]	NL	Align-aware	(16,16)	96 MB	16384	290
HS [17]	NL	Align-aware	(16,16)	16 MB	7396	58
ScalarProd [57]	NL-Xstride	Align-aware	(256,1)	120 MB	2048	329
BLK [57]	NL-Xstride	Align-aware	(128,1)	80 MB	1920	291
Histo-final [75]	NL-Xstride	Align-aware	(512,1)	36 MB	1530	268
Reduction-k6 [57]	NL-Xstride	Align-aware	(256,1)	32 MB	2048	1056
Hotspot3D [17]	NL-Ystride	Align-aware	(64,4)	128 MB	1024	87
CONV [57]	RCL	Row-sched	(16,4)	120 MB	18432	66
Histo-main [75]	RCL	Col-sched	(16,16)	36 MB	1743	201
FWT-k2 [57]	RCL	Col-sched	(256,1)	64 MB	4096	102
SQ-GEMM [57]	RCL	Row-sched	(16,16)	128 MB	2048	61
Alexnet-FC-2 [57], [77]	RCL	Col-sched	(32,4)	400 MB	2048	8
VGGnet-FC-2 [57], [77]	RCL	Col-sched	(32,4)	76 MB	8192	8
Resnet-50-FC [57], [77]	RCL	Col-sched	(32,4)	99 MB	16384	17
LSTM-1 [4], [57]	RCL	Col-sched	(32,4)	64 MB	4096	6
LSTM-2 [4], [57]	RCL	Col-sched	(32,4)	32 MB	2048	27
TRA [57]	RCL	Row-sched	(16,16)	32 MB	16384	291
PageRank [16]	ITL	Kernel-wide	(128,1)	18 MB	23365	85
BFS-relax [60]	ITL	Kernel-wide	(256,1)	220 MB	2048	508
SSSP [16]	ITL	Kernel-wide	(64,1)	57 MB	4131	585
Random-loc [84]	ITL	Kernel-wide	(256,1)	64 MB	41013	4128
Kmeans-noTex [67]	ITL	Kernel-wide	(256,1)	60 MB	1936	158
SpMV-jds [75]	ITL	Kernel-wide	(32,1)	30 MB	4585	640
B+tree [17]	unclassified	Kernel-wide	(256,1)	16 MB	6000	112
LBM [75]	unclassified	Kernel-wide	(120,1)	370 MB	18000	784
StreamCluster [75]	unclassified	Kernel-wide	(512,1)	56 MB	1024	89

We use the *cudaMemAdvise* API to place the data in the correct node, assuming a 4k page. For threadblock scheduling, we used multi-kernel execution where we launch each kernel on a different GPU using CUDA streams. The kernel code was not changed and we did not employ any data replication or reactive solutions as practiced in optimized multi-GPU libraries [58], [72]. If we had access to the GPU driver, we could provide these features to the user transparently. When applying LASP's input aware scheduler and placement on real hardware, we observed $1.9\times$ and $1.4\times$ performance improvement compared to CODA and kernel-wide partitioning respectively. This performance improvement is achieved by preserving row- and column-locality and favoring column-binding scheduling over the row-binding scheduling when matrix B is larger than matrix A. Although this speedup required hand application coding to implement the LASP placement functionality, it is an existence proof that static analysis based locality management can lead to significant changes in performance on real systems today and into the future.

V. Experimental Results

A. Simulation Results of LADM

Figure 9 and 10 show the normalized performance and off-chip memory traffic for LADM, H-CODA [36] and a hypothetical monolithic GPU, when running on our simulated multi-GPU system described in Section IV-A. Compared to H-CODA, LADM improves the performance by $1.8\times$ and decrease inter-GPU memory traffic by $4\times$ on average. H-CODA and LADM are both aware of page-alignment issues. Thus, for the *VecAdd*, they both achieve the same performance. However, LADM achieves better performance in the remaining no-locality workloads due to its stride-aware placement. H-CODA fails to exploit the strided accesses found in the no-locality workloads, which causes more than 50% of memory accesses to go off-chip. Moreover, in stencil workloads, *SRAD*, *HS* and *HotSpot3D*, LADM outperforms H-CODA by $4\times$ on average by launching contiguous threadblocks and exploiting adjacent locality of stencil workloads.

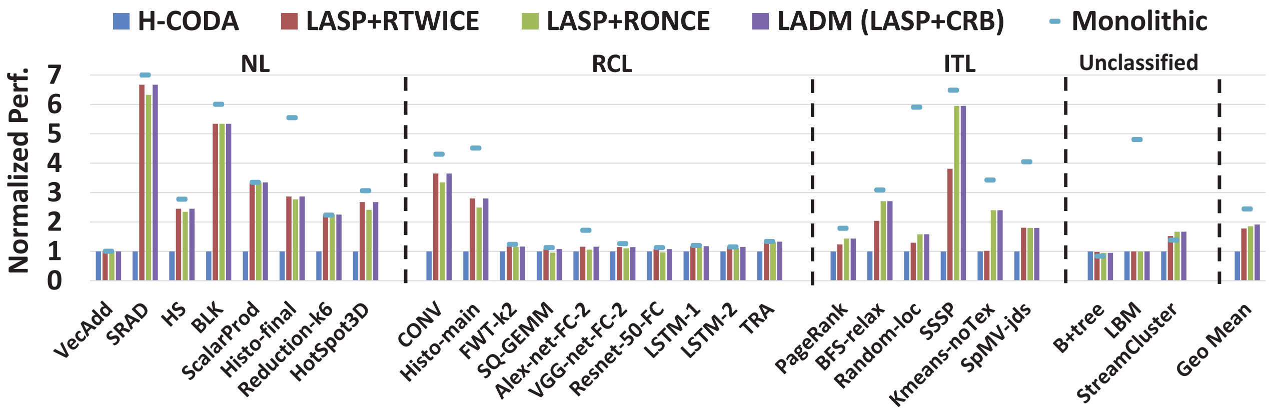


Fig. 9: Performance of H-CODA, LASP with RTWICE and RONCE, LADM and hypothetical monolithic GPU. The data are normalized to H-CODA performance.

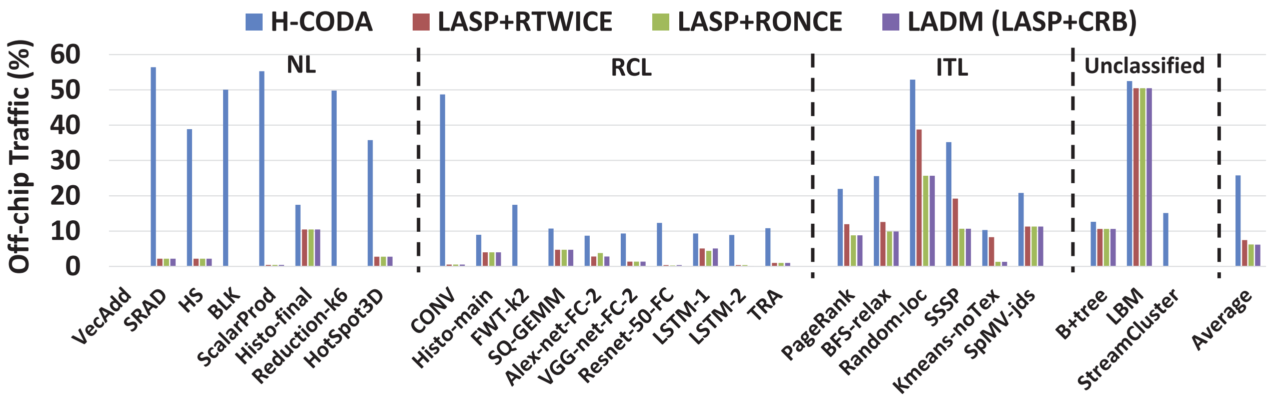
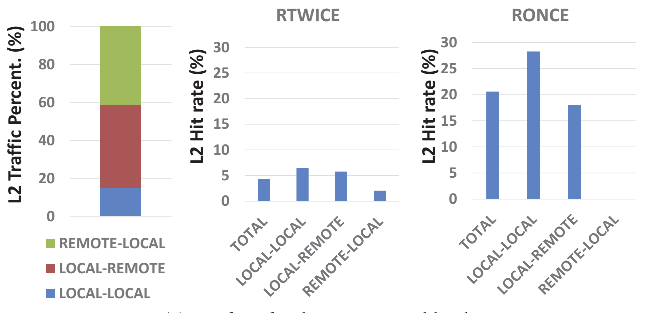


Fig. 10: Percentage of total memory traffic that goes off-node for H-CODA vs LASP vs LADM.

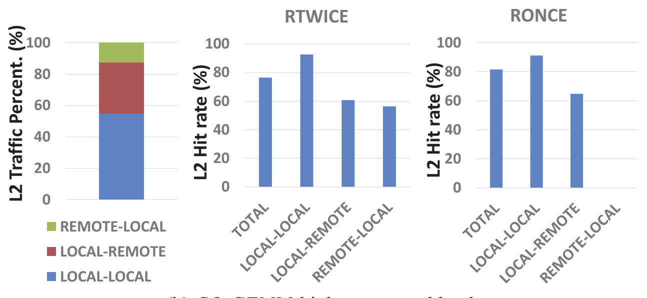
In column-locality and row-locality workloads, LADM outperforms H-CODA by 2.25×. Exploiting the column and row locality efficiently and launching the same threadblock row or column to the same chip has a substantial effect on performance. However, due to the round-robin page and threadblock interleaving of H-CODA, it fails to exploit row- and column-locality. In the machine-learning workloads, L2 remote-caching filters out off-chip traffic significantly with only 8% remaining in H-CODA. However, because of its row and column schedulers, along with its input size awareness, LADM reduces off-chip traffic further, and outperforms H-CODA by 17% on average. Although H-CODA's static analysis is agnostic to column sharing among threadblocks, it performs well when column placement is preferable. The matrix sizes in these machine-learning layers are aligned such that H-CODA's static page interleaving happens to place shared pages on the same node.

In the ITL workloads, H-CODA fails to exploit the locality between adjacent edges in graphs represented in CSR format. In contrast, LASP preserves locality by partitioning the data into large chunks of consecutive pages, improving performance by 1.7× on average. Furthermore, after applying our RONCE policy, LASP+RONCE outperforms RTWICE by an average of 38%. However, applying RTWICE outperforms RONCE by 8% on average for RCL and stencil workloads. Thus, CRB takes the best of both policies by enabling RONCE in ITL workloads and RTWICE in other locality patterns. In the unclassified workloads, LADM does not improve either performance or off-chip data accesses, except for *streamcluster*. Some workloads, like *b+tree* and *streamcluster* achieve higher performance than the monolithic GPU due to reducing bank conflicts and higher cache hit rate in the distributed L2 cache of the multi-GPU configuration. Similar trends were also observed in prior work [84].

Overall, LADM outperforms H-CODA by 1.8× on average and capturing 82% of monolithic chip performance. The reasons behind the remaining 18% performance gap between LADM and monolithic chip are three-fold. First, complex indices are used, as in *lbm* and *histo*, and LADM fails to exploit their locality. Second, irregular data-dependent accesses with no intra-thread locality are frequently generated in many ITL graph workloads, and L2 remote-caching has limited impact to reduce off-chip traffic. Third, the L2 cache coherence overhead, that invalidates L2 caches between kernel boundaries, combined with global synchronization, destroys the inter-kernel locality that was exploited in the large L2 cache of the monolithic chip. Recent work [66] on hardware-



(a) *random_loc* low reuse workload.



(b) *SQ-GEMM* high reuse workload.

Fig. 11: Case study of RONCE cache policy effectiveness on high and low reuse workloads.

supported L2 cache coherence is orthogonal to LADM and can be integrated to reduce the L2 coherence overhead.

B. Remote Request Bypassing Analysis

To better understand the remote request bypassing technique, we classify incoming L2 traffic into one of three categories: (1) LOCAL-LOCAL: A memory request generated from a local (in-node) core and serviced by local DRAM. (2) LOCAL-REMOTE: A memory request generated from a local (in-node) core. On a miss, the DRAM for the memory request is on a remote node. (3) REMOTE-LOCAL: A memory request generated from a remote node. On a miss, the DRAM for the memory request is on the local DRAM node. The total number of misses in LOCAL-REMOTE traffic is equal to the total number of REMOTE-LOCAL accesses.

Figure 11a presents a case study of the *random_loc* workload, where RONCE improves the performance. In *random_loc*, REMOTE-LOCAL traffic has a low hit-rate when applying RTWICE. Additionally, REMOTE-LOCAL represents 45% of the L2 traffic and causes severe contention with local accesses. Applying RONCE to bypass the REMOTE-LOCAL accesses gives more cache resources to the other traffic types and improves total L2 hit-rate by $4\times$. Improving the LOCAL-REMOTE hit-rate leads to fewer off-chip accesses, resulting in better performance. In contrast, Figure 11b plots the results when RONCE hurts the performance in *SQ-GEMM* workload. As shown in figure, REMOTE-LOCAL represents 12% of the traffic and has a relatively high hit-rate from the inter-GPU data sharing of the shared matrix. Thus, bypassing REMOTE-LOCAL leads to a performance degradation.

VI. Related Work

A number of researchers [28], [32], [48] have explored disintegrating multi-core CPUs into smaller chips in order to improve manufacturing yield. In a multi-GPU context, past work [36], [51], [84] investigated similar multi-socket and MCM NUMA GPU designs to scale GPU performance beyond a single socket. We have discussed their approaches in details throughout this paper and compare their results with LADM. Baruah et al. [7] propose hardware-software support for page migration in multi-GPU shared-memory systems. Milic et al. [51] propose dynamic, phase-aware interconnect bandwidth partitioning. They also dynamically adapt L2 caching policy to minimize NUMA effects. These works employ reactive runtime solutions whereas we apply a low-overhead proactive approach.

Young et al. [84] propose a DRAM-cache with optimized hardware coherence for multi-GPU systems. Xiaowei et al. [66] propose a customized L2 cache coherence protocol for hierarchical multi-chiplet multi-GPU systems. These cache coherence protocols are orthogonal to our work and can be applied on top of LADM for further performance improvement.

While significant work has been done to optimize weak-scaling performance using MPI + GPUs (where each rank controls a GPU operating on a relatively isolated partition of data [2], [39]) or via the OpenCL runtime driver [38], [41]. However, transparently achieving *strong scaling* on NUMA-GPU systems with diverse sharing patterns is still an open problem, which we aim to address in this work.

Prior work on locality-aware threadblock scheduling in single GPU contexts has either not used static analysis [29], [42], [82] or performed a subset of the analysis done by LADM [18], [43] simply because the placement of data has not been an objective. Handling page alignment, the effect of remote caching, and matching competing access patterns to data structures are all issues that arise in the NUMA context that are not addressed in prior work on threadblock scheduling for cache locality. It is difficult to provide a fair quantitative comparison to these works, as it requires us to fill-in-the-blanks on how the techniques would be applied to NUMA-GPUs.

Several works [1], [37], [44], [85] have provided batching and reactive prefetching to improve UVM performance in single GPU systems. LASP can be extended to efficiently support oversubscribed memory by proactively placing the next page where it is predicted to be accessed, avoiding page-faulting overheads. Using the locality table information, the pages that are already accessed by finished threadblocks and will not be used again, can be evicted and replaced with the new pages *proactively*.

Compiler-assisted index analysis has been used in CPUs and GPUs to perform affine loops transformation in order to: (1) improve locality via data tiling within a single-GPU machine [8], [71], [83], and (2) automatically parallelize serial code on parallel machines [9], [31], [46], [61]. However, these works perform source-to-source transformation and do not provide any runtime decisions on *how* to efficiently schedule the threads. Furthermore, prior work on GPU static analysis does not exploit all the locality patterns identified by LADM. In this work, we extend single thread index analysis to be threadblock-centric for the NUMA-GPU domain.

It is worth mentioning that, with modifications to account for threadblock motion and inter-thread sharing, a polyhedral framework [10], [24], [71] could be used in place of LADM's index analysis. However, we believe that LADM‘s simpler and effective index-based analysis increases the likelihood it will be adopted in contemporary GPU compilers (e.g. NVCC [54]). Either way, the choice of compiler infrastructure used is orthogonal to the datablock analysis proposed in this paper.

Data placement has been a focus of CPU research in OpenMP NUMA systems. Solutions include adding new OpenMP language primitives which are explicitly used by the programmer [14], [21], [49], [50], compiler-assited page migration [47], [64] or reactively changing the virtual page size [23]. Although thread scheduling is a concern in CPU-NUMA systems, the focus is largely on workload balancing via advanced work stealing algorithms [59] or avoiding cache thrashing [52], but not to ensure memory page locality. In this work, we coordinate both data placement and thread scheduling to exploit various locality patterns of massively multithreaded multi-GPU systems.

VII. Conclusion

Thanks to high levels of inherent parallelism, many GPU workloads will be able to strongly scale performance, if large enough GPUs can be built. However, due to the physical limitations of chip and interconnect technologies, GPUs built with enough resources to leverage this abundant parallelism will have to overcome significant NUMA effects. This work describes a locality-aware data management system designed to transparently overcome the NUMA effects of future hierarchical GPUs. By combining static analysis with hardware data placement, thread scheduling, and cache insertion policies LADM decreases inter-GPU memory traffic by $4\times$, improving system performance by $1.8\times$ across a range of workloads with varying locality. LADM demonstrates that intelligent coordination of threadblock scheduling and data placement can offset the need for expensive GPU interconnect technologies in the future.

Acknowledgments

This work was supported, in part, by NSF CCF #1910924 and Sandia National Labs [2].

References

[1] N. Agarwal, D. Nellans, M. O'Connor, S. W. Keckler, and T. F. Wenisch, "Unlocking Bandwidth for GPUs in CC-NUMA Systems," in *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 354–365.

[2]Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

[2] A. M. Aji, L. S. Panwar, F. Ji, M. Chabbi, K. Murthy, P. Balaji, K. R. Bisset, J. Dinan, W.-c. Feng, J. Mellor-Crummey *et al.*, "On the Efficacy of GPU-Integrated MPI for Scientific Applications," in *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, 2013, pp. 191–202.

[3] R. Allen and K. Kennedy, *Optimizing compilers for modern architectures: a dependence-based approach.* Morgan Kaufmann San Francisco, 2002, vol. 289.

[4] J. Appleyard, T. Kocisky, and P. Blunsom, "Optimizing Performance of Recurrent Neural Networks on GPUs," *arXiv preprint arXiv:1604.01946*, 2016.

[5] A. Arunkumar, E. Bolotin, B. Cho, U. Milic, E. Ebrahimi, O. Villa, A. Jaleel, C.-J. Wu, and D. Nellans, "MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 320–332.

[6] A. Arunkumar, E. Bolotin, D. Nellans, and C.-J. Wu, "Understanding the Future of Energy Efficiency in Multi-Module GPUs," in *IEEE 25th International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 519–532.

[7] T. Baruah, Y. Sun, A. T. Diner, S. A. Mojumder, J. L. Abelln, Y. Ukidave, A. Joshi, N. Rubin, J. Kim, and D. Kaeli, "Griffin: Hardware-Software Support for Efficient Page Migration in Multi-GPU Systems," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 596–609.

[8] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs," in *Proceedings of the 22nd annual international conference on Supercomputing*, 2008, pp. 225–234.

[9] M. M. Baskaran, J. Ramanujam, and P. Sadayappan, "Automatic C-to-CUDA Code Generation for Affine Programs," in *International Conference on Compiler Construction*, 2010, pp. 244–263.

[10] C. Bastoul, "Code Generation in the Polyhedral Model Is Easier Than You Think," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2004, pp. 7–16.

[11] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali, "A Case for NUMA-aware Contention Management on Multicore Systems," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques (PACT)*, 2010, pp. 557–558.

[12] W. Bolosky, R. Fitzgerald, and M. Scott, "Simple But Effective Techniques for NUMA Memory Management," *ACM SIGOPS Operating Systems Review*, pp. 19–31, 1989.

[13] P. Bright, "Moore's law really is dead this time," https://arstechnica.com/information-technology/2016/02/moores-law-really-is-dead-this-time/, 2016.

[14] F. Broquedis, N. Furmento, B. Goglin, P.-A. Wacrenier, and R. Namyst, "ForestGOMP: an Efficient OpenMP Environment for NUMA Architectures," *International Journal of Parallel Programming*, pp. 418–439, 2010.

[15] J. Cabezas, L. Vilanova, I. Gelado, T. B. Jablin, N. Navarro, and W.-m. W. Hwu, "Automatic Parallelization of Kernels in Shared-Memory Multi-GPU Nodes," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015, pp. 3–13.

[16] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding Irregular GPGPU Graph Applications," in *IEEE International Symposium on Workload Characterization (IISWC)*, 2013, pp. 185–195.

[17] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.

[18] L.-J. Chen, H.-Y. Cheng, P.-H. Wang, and C.-L. Yang, "Improving GPGPU Performance via Cache Locality Aware Thread Block Scheduling," *IEEE Computer Architecture Letters*, pp. 127–131, 2017.

[19] R. Das, R. Ausavarungnirun, O. Mutlu, A. Kumar, and M. Azimi, "Application-to-Core Mapping Policies to Reduce Memory Interference in Multi-Core Systems," in *IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013, pp. 107–118.

[20] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, V. Quema, and M. Roth, "Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems," in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013, p. 381394.

[21] M. Diener, E. H. Cruz, M. A. Alves, P. O. Navaux, and I. Koren, “Affinity-Based Thread and Data Mapping in Shared Memory Systems,” *ACM Computing Surveys (CSUR)*, 2017.

[22] B. Falsafi and D. A. Wood, “Reactive NUMA: A Design for Unifying S-COMA and CC-MAMA,” in *Proceedings of the 24th annual international symposium on Computer architecture (ISCA)*, 1997, pp. 229–240.

[23] F. Gaud, B. Lepers, J. Decouchant, J. Funston, A. Fedorova, and V. Quéma, “Large Pages May Be Harmful on NUMA Systems,” in *Proceedings of 2014 USENIX Annual Technical Conference (USENIX ATC)*, 2014, pp. 231–242.

[24] T. Grosser, A. Groesslinger, and C. Lengauer, “Polly: Performing Polyhedral Optimizations on a Low-Level Intermediate Representation,” *Parallel Processing Letters*, pp. 1–27, 2012.

[25] S. Han, H. Mao, and W. J. Dally, “Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding,” *arXiv preprint arXiv:1510.00149*, 2015.

[26] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, “Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches,” in *Proceedings of the 36th annual international symposium on Computer architecture (ISCA)*, 2009, pp. 184–195.

[27] Intel, “Intel EMIB,” https://www.intel.com/content/www/us/en/foundry/emib.html/, 2016.

[28] N. E. Jerger, A. Kannan, Z. Li, and G. H. Loh, “NoC Architectures for Silicon Interposer Systems: Why Pay for more Wires when you Can Get them (from your interposer) for Free?” in *Proceedings of the 47th Annual International Symposium on Microarchitecture (MICRO)*, 2014, pp. 458–470.

[29] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, “OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance,” in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013, pp. 395–406.

[30] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, “Orchestrated Scheduling and Prefetching for GPGPUs,” in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013, pp. 332–343.

[31] J. Juega, J. Gomez, C. Tenllado, S. Verdoolaege, A. Cohen, and F. Catthoor, “Evaluation of state-of-the-art polyhedral tools for automatic code generation on GPUs,” *XXIII Jornadas de Paralelismo, Univ. Complutense de Madrid*, 2012.

[32] A. Kannan, N. E. Jerger, and G. H. Loh, “Enabling Interposer-based Disintegration of Multi-core Processors,” in *48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 546–558.

[33] ——, “Exploiting Interposer Technologies to Disintegrate and Reintegrate Multicore Processors,” *IEEE Micro*, pp. 84–93, 2016.

[34] O. Kayran, A. Jog, M. T. Kandemir, and C. R. Das, “Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs,” in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013, pp. 157–166.

[35] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, “Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling,” in *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 473–486.

[36] H. Kim, R. Hadidi, L. Nai, H. Kim, N. Jayasena, Y. Eckert, O. Kayiran, and G. Loh, “CODA: Enabling Co-location of Computation and Data for Multiple GPU Systems,” *ACM Transactions on Architecture and Code Optimization (TACO)*, pp. 1–23, 2018.

[37] H. Kim, J. Sim, P. Gera, R. Hadidi, and H. Kim, “Batch-Aware Unified Memory Management in GPUs for Irregular Workloads,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, pp. 1357–1370.

[38] J. Kim, H. Kim, J. H. Lee, and J. Lee, “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)*, 2011, pp. 277–288.

[39] V. V. Kindratenko, J. J. Enos, G. Shi, M. T. Showerman, G. W. Arnold, J. E. Stone, J. C. Phillips, and W.-m. Hwu, “GPU Clusters for High-Performance Computing,” in *2009 IEEE International Conference on Cluster Computing and Workshops*, 2009, pp. 1–8.

[40] D. Kirk and W. Wen-mei, *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010.

[41] J. Lee, M. Samadi, Y. Park, and S. Mahlke, “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)*, 2013, pp. 245–256.

[42] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu, “Improving GPGPU Resource Utilization Through Alternative Thread Block Scheduling,” in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014, pp. 260–271.

[43] A. Li, S. L. Song, W. Liu, X. Liu, A. Kumar, and H. Corporaal, “Locality-Aware CTA Clustering for Modern GPUs,” in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017, pp. 297–311.

[44] C. Li, R. Ausavarungnirun, C. J. Rossbach, Y. Zhang, O. Mutlu, Y. Guo, and J. Yang, “A Framework for Memory Oversubscription Management in Graphics Processing Units,” in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 49–63.

[45] H. Li, S. Tandri, M. Stumm, and K. C. Sevcik, “Locality and Loop Scheduling on NUMA Multiprocessors,” in *International Conference on Parallel Processing (ICPP)*, 1993, pp. 140–147.

[46] W. Li, “Compiling for NUMA Parallel Machines,” Cornell University, Tech. Rep., 1994.

[47] Y. Li, R. Melhem, A. Abousamra, and A. K. Jones, “Compiler-assisted Data Distribution for Chip Multiprocessors,” in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010, pp. 501–512.

[48] G. H. Loh, N. E. Jerger, A. Kannan, and Y. Eckert, “Interconnect-Memory Challenges for Multi-chip, Silicon Interposer Systems,” in *Proceedings of the 2015 International Symposium on Memory Systems (MEMSYS)*, 2015, pp. 3–10.

[49] Z. Majo and T. R. Gross, “Matching Memory Access Patterns and Data Placement for NUMA Systems,” in *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO)*, 2012, pp. 230–241.

[50] C. McCurdy and J. Vetter, “Memphis: Finding and Fixing NUMA-related Performance Problems on Multi-core Platforms,” in *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2010, pp. 87–96.

[51] U. Milic, O. Villa, E. Bolotin, A. Arunkumar, E. Ebrahimi, A. Jaleel, A. Ramirez, and D. Nellans, “Beyond the Socket: NUMA-aware GPUs,” in *Proceedings of the 50th Annual International Symposium on Microarchitecture (MICRO)*, 2017, pp. 123–135.

[52] A. Muddukrishna, P. A. Jonsson, and M. Brorsson, “Locality-Aware Task Scheduling and Data Distribution for OpenMP Programs on NUMA Systems and Manycore Processors,” *Scientific Programming*, 2015.

[53] S. Naffziger, K. Lepak, M. Paraschou, and M. Subramony, “AMD Chiplet Architecture for High-Performance Server and Desktop Products,” in *IEEE International Solid- State Circuits Conference - (ISSCC)*, 2020, pp. 44–45.

[54] NVIDIA, “NVCC,” https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html.

[55] ——, “NVIDIA NVLink: High Speed GPU Interconnect,” https://www.nvidia.com/en-us/design-visualization/nvlink-bridges/.

[56] ——, “NVIDIA NVSWITCH,” https://images.nvidia.com/content/pdf/nvswitch-technical-overview.pdf.

[57] ——, “CUDA C/C++ SDK Code Samples,” http://developer.nvidia.com/cuda-cc-sdk-code-samples, 2011.

[58] ——, “cuBLASXt,” https://docs.nvidia.com/cuda/cublas/index.html#using-the-cublasXt-api, 2020.

[59] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, M. Spiegel, and J. F. Prins, “OpenMP task scheduling strategies for multicore NUMA systems,” *The International Journal of High Performance Computing Applications*, pp. 110–124, 2012.

[60] M. A. O’Neil and M. Burtscher, “Microarchitectural Performance Characterization of Irregular GPU Kernels,” in *IEEE International Symposium on Workload Characterization (IISWC)*, 2014, pp. 130–139.

[61] Y. Paek and D. A. Padua, “Experimental Study of Compiler Techniques for NUMA Machines,” in *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, 1998, pp. 187–193.

[62] S. Pal, D. Petrisko, A. A. Bajwa, P. Gupta, S. S. Iyer, and R. Kumar, “A Case for Packageless Processors,” in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 466–479.

[63] S. Pal, D. Petrisko, M. Tomei, P. Gupta, S. S. Iyer, and R. Kumar, “Architecting Waferscale Processors-A GPU Case Study,” in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 250–263.

[64] G. Piccoli, H. N. Santos, R. E. Rodrigues, C. Pousa, E. Borin, and F. M. Quintão Pereira, “Compiler Support for Selective Page Migration in NUMA Architectures,” in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2014, pp. 369–380.

[65] J. W. Poulton, W. J. Dally, X. Chen, J. G. Eyles, T. H. Greer, S. G. Tell, J. M. Wilson, and C. T. Gray, “A 0.54 pJ/b 20 Gb/s Ground-Referenced Single-Ended Short-Reach Serial Link in 28 nm CMOS for Advanced Packaging Applications,” *IEEE Journal of Solid-State Circuits*, pp. 3206–3218, 2013.

[66] X. Ren, D. Lustig, E. Bolotin, A. Jaleel, O. Villa, and D. Nellans, “HMG: Extending Cache Coherence Protocols Across Modern Hierarchical Multi-GPU Systems,” in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 582–595.

[67] T. G. Rogers, M. O’Connor, and T. M. Aamodt, “Cache-Conscious Wavefront Scheduling,” in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012, pp. 72–83.

[68] ——, “Divergence-aware Warp Scheduling,” in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 99–110.

[69] A. Saulsbury, T. Wilkinson, J. Carter, and A. Landin, “An Argument for Simple COMA,” in *First IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 1995, pp. 276–285.

[70] Y. S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina *et al.*, “Simba: Scaling Deep-Learning Inference with Multi-Chip-Module-Based Architecture,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019, pp. 14–27.

[71] J. Shirako, A. Hayashi, and V. Sarkar, “Optimized Two-Level Parallelization for GPU Accelerators using the Polyhedral Model,” in *Proceedings of the 26th International Conference on Compiler Construction*, 2017, pp. 22–33.

[72] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, “Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism,” *arXiv preprint arXiv:1909.08053*, 2019.

[73] T. Simonite, “Moores Law Is Dead. Now What?” https://www.technologyreview.com/s/601441/moores-law-is-dead-now-what/, 2016.

[74] B. Solca, “NVIDIA DGX-2 is the world largest gpu.” https://www.notebookcheck.net/Nvidia-DGX-2-is-the-world-s-largest-GPU.292930.0.html, 2018.

[75] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, “Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing,” *Center for Reliable and High-Performance Computing*, vol. 127, 2012.

[76] Y. Sun, T. Baruah, S. A. Mojumder, S. Dong, X. Gong, S. Treadway, Y. Bao, S. Hance, C. McCardwell, V. Zhao *et al.*, “MGPUSim: Enabling Multi-GPU Performance Modeling and Optimization,” in *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*, 2019, pp. 197–209.

[77] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, “Efficient Processing of Deep Neural Networks: A Tutorial and Survey,” *Proceedings of the IEEE*, pp. 2295–2329, 2017.

[78] D. Tam, R. Azimi, and M. Stumm, “Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors,” in *ACM SIGOPS Operating Systems Review*, 2007, pp. 47–58.

[79] T. Vijayaraghavany, Y. Eckert, G. H. Loh, M. J. Schulte, M. Ignatowski, B. M. Beckmann, W. C. Brantley, J. L. Greathouse, W. Huang, A. Karunanithi *et al.*, “Design and Analysis of an APU for Exascale Computing,” in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 85–96.

[80] N. Vijaykumar, E. Ebrahimi, K. Hsieh, P. B. Gibbons, and O. Mutlu, “The Locality Descriptor: A Holistic Cross-Layer Abstraction to Express Data Locality In GPUs,” in *45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 829–842.

[81] O. Villa, D. R. Johnson, M. Oconnor, E. Bolotin, D. Nellans, J. Luitjens, N. Sakharnykh, P. Wang, P. Micikevicius, A. Scudiero *et al.*, “Scaling the Power Wall: A Path to Exascale,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2014, pp. 830–841.

[82] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang, “Coordinated Static and Dynamic Cache Bypassing for GPUs,” in *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 76–88.

[83] Y. Yang, P. Xiang, J. Kong, and H. Zhou, “A GPGPU Compiler for Memory Optimization and Parallelism Management,” in *InProceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2010, pp. 86–97.

[84] V. Young, A. Jaleel, E. Bolotin, E. Ebrahimi, D. Nellans, and O. Villa, “Combining HW/SW Mechanisms to Improve NUMA Performance of Multi-GPU Systems,” in *Proceedings of the 51th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 339–351.

[85] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler, “Towards High Performance Paged Memory for GPUs,” in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 345–357.