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.

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

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.

$$MinTBBatch = \frac{pageSize}{datablockSize} \quad (2)$$

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.

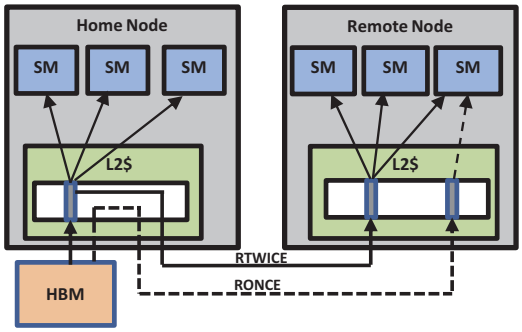


Fig. 8: llustration of existing NUMA caching policy *cache-remote-twice* (the solid line) and our proposed *cache-remote-once* cache management strategy (the dashed line)

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.

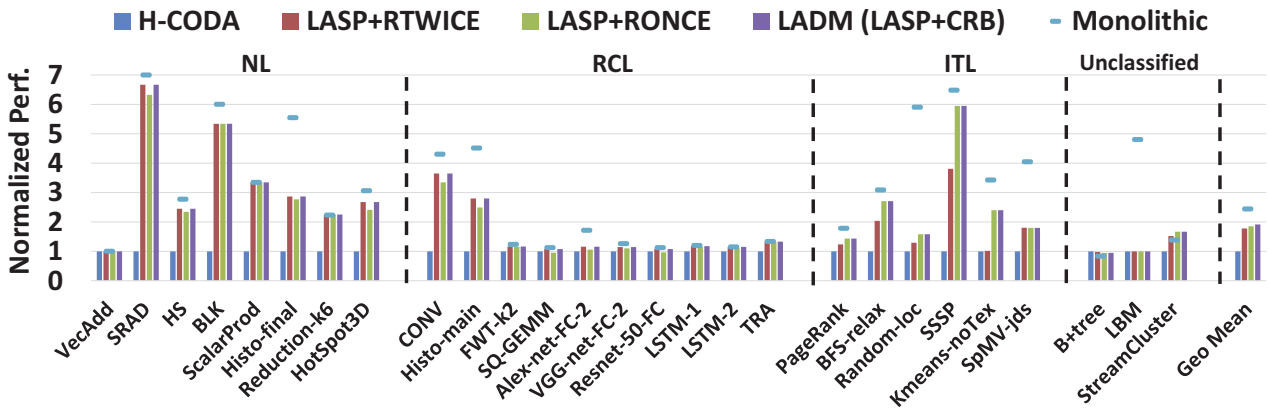


Fig. 9: Performance of H-CODA, LASP with RTWICE and RONCE, LADM and hypothetical monolithic GPU. The data are normalized to H-CODA performance.

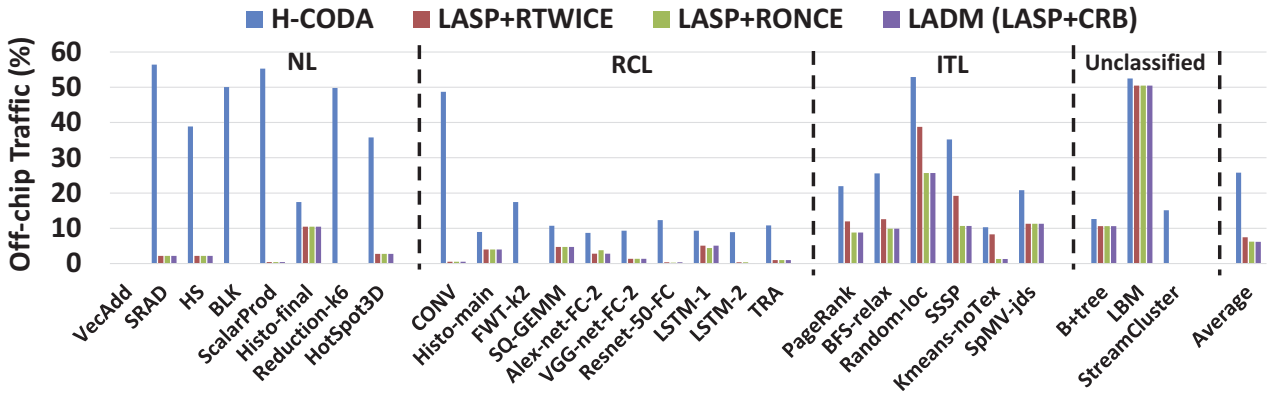
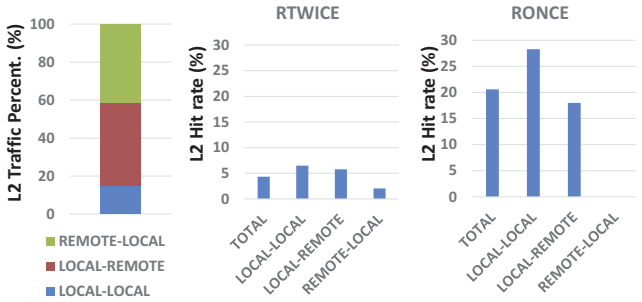


Fig. 10: Percentage of total memory traffic that goes off-node for H-CODA vs LASP vs LADM.

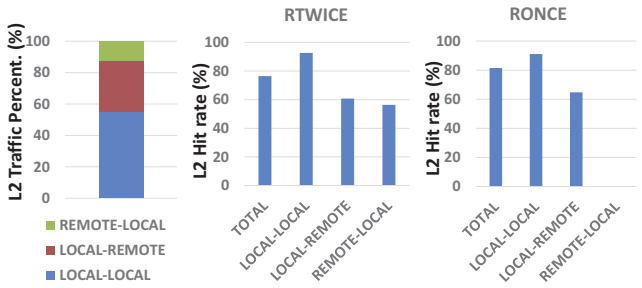
In column-locality and row-locality workloads, LADM outperforms H-CODA by 2.25×. Exploiting the column and row locality efficiently and launching the same threadblock row or column to the same chip has a substantial effect on performance. However, due to the round-robin page and threadblock interleaving 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.