Quantifying Data Locality in Dynamic Parallelism in GPUs

XULONG TANG, Pennsylvania State University, USA
ASHUTOSH PATTNAIK, Pennsylvania State University, USA
ONUR KAYIRAN, Advanced Micro Devices, Inc., USA
ADWAIT JOG, College of William & Mary, USA
MAHMUT TAYLAN KANDEMIR, Pennsylvania State University, USA
CHITA DAS, Pennsylvania State University, USA

GPUs are becoming prevalent in various domains of computing and are widely used for streaming (regular) applications. However, they are highly inefficient when executing irregular applications with unstructured inputs due to load imbalance. Dynamic parallelism (DP) is a new feature of emerging GPUs that allows new kernels to be generated and scheduled from the device-side (GPU) without the host-side (CPU) intervention to increase parallelism. To efficiently support DP, one of the major challenges is to saturate the GPU processing elements and provide them with the required data in a timely fashion. There have been considerable efforts focusing on exploiting data locality in GPUs. However, there is a lack of quantitative analysis of how irregular applications using dynamic parallelism behave in terms of data reuse.

In this paper, we quantitatively analyze the data reuse of dynamic applications in three different granularities of schedulable units: kernel, work-group, and wavefront. We observe that, for DP applications, data reuse is highly irregular and is heavily dependent on the application and its input. Thus, existing techniques cannot exploit data reuse effectively for DP applications. To this end, we first conduct a limit study on the performance improvements that can be achieved by hardware schedulers that are provided with accurate data reuse information. This limit study shows that, on an average, the performance improves by 19.4% over the baseline scheduler. Based on the key observations from the quantitative analysis of our DP applications, we next propose LASER, a Locality-Aware SchedulER, where the hardware schedulers employ data reuse monitors to help make scheduling decisions to improve data locality at runtime. Our experimental results on 16 benchmarks show that LASER, on an average, can improve performance by 11.3%.

CCS Concepts: • **Computing methodologies → Massively parallel and high-performance simulations**; • **Computer systems organization → Single instruction, multiple data**;

Additional Key Words and Phrases: General purpose graphics processing units, data reuse, cache locality, computation scheduling, performance evaluation

ACM Reference Format:
Xulong Tang, Ashutosh Pattnaik, Onur Kayiran, Adwait Jog, Mahmut Taylan Kandemir, and Chita Das. 2018. Quantifying Data Locality in Dynamic Parallelism in GPUs. *Proc. ACM Meas. Anal. Comput. Syst.* 2, 3, Article 39 (December 2018), 24 pages. https://doi.org/10.1145/3287318

Authors' addresses: Xulong Tang, Pennsylvania State University, State College, PA, 16801, USA, xzt102@psu.edu; Ashutosh Pattnaik, Pennsylvania State University, State College, PA, 16801, USA, ashutosh@psu.edu; Onur Kayiran, Advanced Micro Devices, Inc. Santa Clara, CA, 95054, USA, onur.kayiran@amd.com; Adwait Jog, College of William & Mary, Williamsburg, VA, 23185, USA, ajog@wm.edu; Mahmut Taylan Kandemir, Pennsylvania State University, State College, PA, 16801, USA, mtk2@psu.edu; Chita Das, Pennsylvania State University, State College, PA, 16801, USA, cxd12@psu.edu.

2476-1249/2018/1-ART39 $15.00
https://doi.org/10.1145/3287318

1 INTRODUCTION

Graphics Processing Units (GPUs) provide massive computational throughput for a wide spectrum of applications from various domains such as computer vision [42], finance [37, 49], machine learning [1, 18], and bioinformatics [55]. As progressively more applications get ported to GPUs for parallelization, the shortcomings of the traditional GPU execution model become evident. Particularly, execution of irregular applications with unstructured inputs on GPUs leads to severe bottlenecks such as imbalanced computational load across the GPU Compute Units (CUs). This inefficiency is widely observed in adaptive meshes and graph applications which are becoming important classes of applications due to their increasing popularity. Therefore, it is becoming more and more difficult to effectively utilize GPUs for such applications [58].

Dynamic Parallelism (DP) is a feature supported by CUDA [40] and OpenCL™ [3]. It allows the generation and scheduling (launching) of kernels dynamically on GPUs without the involvement of a host (CPU). This model of computation is quite useful for irregular applications with unstructured and irregular inputs, as it essentially increases parallelism *on-the-fly*. Specifically, if there are threads which are more compute-intensive than other threads, then these threads (*parent threads*) can parallelize their work by launching more threads (*child kernels*). This would allow for better (and on-demand) load balancing as there are higher number of threads to distribute across the GPU cores. However, by increasing the parallelism and redistributing the threads, the data reuse and access pattern can change dramatically. For example, original intra-thread temporal data reuse (i.e., the data blocks that are reused within a thread) can translate to inter-thread temporal reuse, due to the fact that parent thread offloads computations to its child threads. Moreover, this intra-thread temporal data reuse can even translate to inter-thread spatial locality, as the child kernel has multiple threads, and each child thread can work on a small portion of data.

Prior techniques on GPU cache optimization involved throttling the available parallelism [25, 30], bypassing the cache for certain memory requests to reduce contention [32, 60], and building efficient cache management policies tuned towards GPU applications [27, 41]. These techniques cannot be ported to improve cache performance for DP applications, as they are agnostic to the on-demand kernel launch behavior in DP applications. Furthermore, it is extremely difficult to control the temporal data reuse in applications via cache optimizations as they lack mechanisms to control the scheduling of instructions for execution. To be able to efficiently control the temporal data reuse of applications, we need intelligent scheduling strategies. Prior efforts efficiently scheduled applications based on their reuse behavior using runtime statistics, or via compiler based approaches, *but* they do not consider DP applications [30, 31, 44]. Wang *et al.* [57] proposed a work-group (WG) scheduling mechanism that maps child WGs together with its parent WG to the same compute unit (CU) during execution. This strategy does not differentiate between the children of a parent or take into account the data reuse behavior of the children among themselves. No prior work quantitatively studied the data reuse nor explored the potential performance benefits by *fully* exploiting the data reuse opportunities in DP applications.

Our goal in this paper is three-fold: (1) to quantitatively analyze the intrinsic data reuse opportunities in DP applications; (2) to reveal the best achievable performance improvements through a limit study; and (3) to propose a practical scheduling mechanism that improves data locality. To this end, we first conduct an in-depth data reuse analysis. We define a new "metric" called *reuse ratio* which captures the "intensity" of data reuse. The reuse ratio is computed for each pair of "schedulable units" where a schedulable unit is either a *kernel*, a *work-group*, or a *wavefront*. We then perform a limit study that exploits the reuse ratios for all possible permutations of the schedulable units at each level of the scheduling hierarchy and optimizes the placement (temporally and spatially) of the schedulable units to maximize data locality. Finally, we modify the hardware

schedulers and propose a practical scheduling mechanism that schedules the schedulable units in a locality-aware fashion. This paper makes the following major contributions:

- It provides an in-depth *data reuse* quantification and analysis of GPU dynamic parallelism applications. The analysis is at multiple granularities (kernel, work-group, and wavefront) with respect to suitability for improving cache locality via scheduling techniques.
- It defines a new metric called *reuse ratio*, which captures the intensity of data reuse among schedulable units. We discuss the merits of this metric and demonstrate how to use it to guide scheduling strategies for DP applications.
- It performs a limit study by proposing an *optimal* scheduling mechanism that optimizes compute placement for cache locality by exploiting reuse ratios at each level of scheduling, viz. kernel, work-group and wavefront. This is achieved by providing accurate "reuse ratio" information to the hardware schedulers. The *optimal* scheduler provides, on an average, 19.4% performance improvement over the baseline scheduler.
- Based on the key observations from the data reuse analysis, it proposes LASER, a Locality-Aware SchedulER, that monitors the reuse ratio metric and makes scheduling decisions based on it. Our experimental results show that, on an average, LASER provides 11.3% performance improvement over the baseline scheduler.

2 RELATED WORK

Work-group and wavefront scheduling: There has been a substantial body of work on building efficient work-group and wavefront scheduling mechanisms for GPUs to improve cache performance, memory bandwidth utilization, DRAM performance, system performance, and energy efficiency [19–22, 29–31, 33, 43, 44, 46, 57]. Lee *et al.* [30] proposed work-group and wavefront scheduling techniques which optimize the locality for applications with neighboring work-group data reuse by scheduling work-groups contiguously to CUs rather than in a round-robin fashion. Li *et al.*. [31] developed software-based techniques to improve the inter-work-group locality of an application. Jog *et al.* [21] investigated multiple scheduling techniques to reduce cache contention and improve memory-side prefetching and bank-level parallelism. Lai *et al.* [29] developed a three-stage methodology for mapping threads to cores using a formal model that captures thread characteristics as well as cache sharing behavior. Wang *et al.* [57] proposed work-group scheduling for DP applications, where they bind child WGs to its parent WG in order to exploit parent-child reuse.

Cache management in GPUs: There are several prior efforts focusing on improving data access performance on CPUs and GPUs [13, 23, 26, 47, 51–53]. Improved cache performance in GPUs is mainly achieved via efficient cache management policies ([2, 5, 10, 14, 16, 17, 27, 41]), throttling the amount of parallelism ([25, 30, 31, 54]), and cache bypassing ([10, 32, 59, 60]). Oh *et al.* [41] proposed an adaptive prefetching and scheduling mechanism to improve the GPU cache efficiency. They achieve this by grouping together work-groups and monitoring the data access patterns of the wavefronts in a work-group. Koo *et al.* [27] developed an access pattern-aware cache management technique which dynamically detects the type of locality of each load instruction by monitoring a representative wavefront. Chen *et al.* [10] adaptively bypassed memory requests to the cache based on reuse distances to protect against cache contention.

To our knowledge, this is the first work that systematically investigates the data reuse and access patterns of DP applications at various granularities of schedulable units. Our work is most closely related to the work of Wang *et al.* [57]. However, their approach does not differentiate between the children of a given parent kernel, nor does it consider the intra-kernel data reuse. Therefore, we quantify the memory behavior of DP applications in detail and analyze their data reuse patterns to

answer the following key questions. **Question 1:** *How prevalent are intra-kernel and inter-kernel data reuses in DP applications?* **Question 2:** *What is the effect of launch overhead on data reuse patterns?* **Question 3:** *Do neighboring work-groups have more of temporal or spatial reuse?* and **Question 4:** *Is it necessary to always prioritize the child work-groups?*

3 BACKGROUND

3.1 Dynamic Parallelism

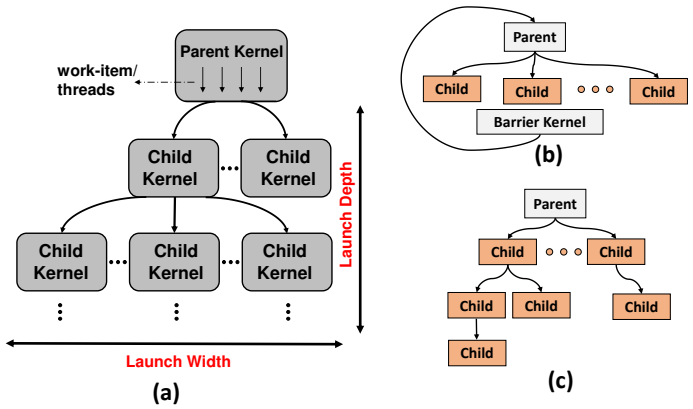

Fig. 1. (a) DP programming model. (b) Type I DP applications. (c) Type II DP applications.

In conventional GPU programming model, computations in an application are mapped to work-items (threads in NVIDIA terminology), work-groups (thread-blocks in NVIDIA terminology), and kernels. A kernel consists of multiple work-groups (WGs) and a WG consists of multiple work-items. At runtime, work-items are scheduled in groups, called "wavefronts". All threads in a wavefront execute the same instruction in a lock-step fashion. Unlike conventional applications, a DP application can launch nested device kernels (child kernels), as shown in Figure 1(a). Each work-item in a kernel has the capability to launch kernels. This feature is particularly useful for irregular applications, as threads with heavy computations can launch a child kernel and *offload* some of its computations to that child kernel. Therefore, one can potentially achieve both parallelism and better resource utilization. Note that, it is the programmer's responsibility to decide whether to launch a child kernel or not from within a parent kernel. Specifically, a *threshold* is set by the programmer and used in DP applications to make kernel launch decisions [38, 54]. In Figure 1(a), we call the depth of the nested launched kernels as *Launch Depth*. It represents the number of nested levels (depth) at which child kernels are launched. Note that the GPU hardware needs to reserve memory space for the child kernels [38, 58]. As a result, the maximum launch depth is limited to 24 by the hardware [40]. At each launch depth, there can be multiple kernels being launched. We refer to the number of kernels launched at a given depth as *Launch Width*.

3.2 Baseline Architecture

Figure 2 shows our *baseline* GPU architecture with support for DP. A DP application (similar to a regular GPU application) starts running on a host CPU and the very first kernel (parent) is launched to the GPU from the host (❶). This kernel is also labeled with a software queue ID (e.g., CUDA stream ID in CUDA terminology), which is used to provide execution ordering among kernels[1] There are 32 hardware work queues (HWQs) located in the Grid Management Unit (GMU). Kernels

[1]In DP, the parent thread can choose two ways to assign software queues (SQs) to child kernels. First, it can create a new SQ before launching a child kernel. Second, each parent WG has a default SQ. Parent threads in the same WG launch child kernels to the default SQ, if no new SQs are explicitly created for the child kernels.

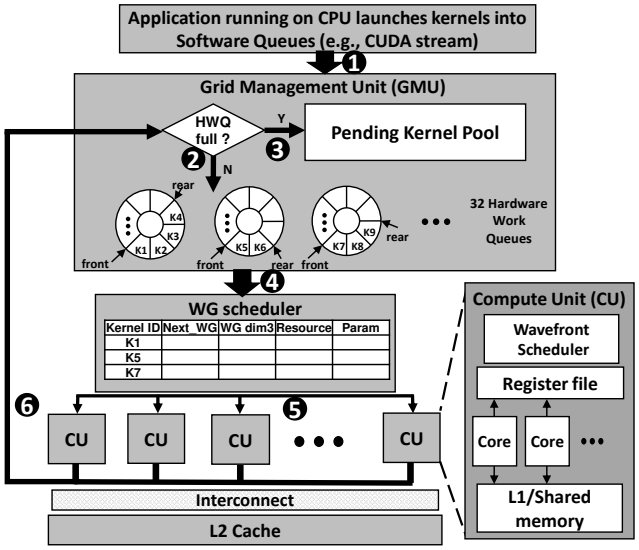

Fig. 2. Baseline GPU architecture.

with the same software queue ID are mapped to the same HWQ, and all kernels in the same HWQ are executed sequentially. If there is an available slot in a HWQ, the launched kernel is pushed into that HWQ (❷). Otherwise, the kernel is suspended in the pending kernel pool (❸), waiting for a free slot in HWQ. Kernels are scheduled for execution in a first-come-first-serve (FCFS) order (❹), with respect to kernel dependencies. The kernels at the head of HWQs can potentially execute concurrently. That is, the WG scheduler can schedule any work-groups (WGs) from the "head-of-queue" kernels, and map them onto CUs, if there are enough available resources (❺). In the baseline execution, the WG scheduler picks up WGs from the "head-of-queue" kernel and distributes them across all CUs in a round-robin fashion. On a CU, wavefronts from WGs are executed on cores. At any time during wavefront execution, multiple wavefronts may be standing by and waiting for execution in order to hide long latency operations (e.g., memory accesses, expensive math operations, etc.). Once a wavefront encounters a long latency operation, that wavefront is switched out, and another "ready" wavefront is scheduled to overlap with the long latency operation. The ready wavefront is chosen by the *wavefront scheduler* which uses a greedy-then-oldest policy (GTO) [46] to select candidate wavefronts. Specifically, in GTO, the same wavefront is always issued to execute if it does not encounter any long latency operations. Whenever a long latency operation is observed, the wavefront is replaced with the "oldest" pending wavefront.

With DP, any thread running on a CU can launch device kernels by calling the driver API. The newly-launched kernels are pushed into GMU by device driver (❻). Note that launching child kernels incurs extra latencies, (called *Launch Overhead*) [58]. Due to this overhead, a child kernel cannot start its execution immediately after launching. This overhead is accurately modeled in our simulation framework, and multiple approaches have been proposed in the literature to mitigate it [9, 15, 54, 56].

3.3 Evaluation Methodology

Infrastructure: We use a cycle-level simulator, GPGPU-Sim [6], that enables DP [56] as our evaluation framework. Table 1 provides the detailed configuration of the baseline GPU architecture. The maximum number of HWQs is 32, and the maximum number of concurrent WGs per CU is 16.

Table 1. GPU configuration parameters.

CU	13 CUs, 1400MHz, 5-Stage Pipeline
Resources per CU	32KB Shared Memory, 64KB Register File, Max.2048 threads (64 wavefronts, 32 threads/wavefront)
cache per CU	32KB 8-way L1 D-cache, 12KB 24-way Texture Cache, 8KB 2-way Constant cache, 2KB 4-way L1 I-cache, 128B cacheline
L2 Unified cache	128KB/Memory Partition, 1536KB Total Size, 128B cacheline, 8-way associativity
Scheduler	Greedy-Then-Oldest (GTO) [46] dual wavefront scheduler, Round-Robin (RR) WG scheduler
Concurrency	16 WGs/CU, 32 HWQs across all CUs
Interconnect	1 crossbar/direction (13 CUs, 6 MCs), 1.4GHz, islip VC and switch allocators
DRAM Model	2 Memory Partitions/MC, 6 MCs, FR-FCFS (128 Request Queue Size/MC)
Child Kernel Launch Overhead	$Latency = Ax + b$ where A is 1721 cycles, b is 20210 cycles, x is number of child kernels launched per wavefront [56]

Table 2. Benchmark characteristics: Number of kernels, type, depth (refers to the number of stages (parent-child-barrier) for Type-I applications and Launch Depth for Type-II applications), and high-reuse chain length (kernels/WGs).

Application	Input Sets	Benchmark	Total # of Kernels	Type	Depth	High-Reuse Chain of Kernels (Max, Avg)	High-Reuse Chain of WGs (Max, Avg)
Adaptive Mesh Refinement [58]	Combustion Simulation [28]	AMR	1261	II	24	(6, 1.5)	(6, 1.5)
Breadth-First Search [58]	Small Graph	BFS-small	1030	I	1	(1, 1)	(240, 11)
	Citation Network [48]	BFS-citation	126	I	23	(1,1)	(240, 37)
	Graph 500 [48]	BFS-graph500	6899	I	6	(27, 3)	(240, 17)
Graph Coloring [34]	Citation Network [48]	GC-citation	221	I	87	(1, 1)	(443, 2)
	Graph 500 [48]	GC-graph500	924	I	139	(10, 6)	(18, 4)
Relation Join [58]	Gaussian Distribution	JOIN-Gaussian	159	I	1	(21, 4)	(227, 4)
	Uniform Distribution	JOIN-uniform	6725	I	1	(721, 361)	(7, 2)
Mandelbrot Set [54]	N/A	Mandel	1025	II	6	(1, 1)	(1, 1)
Sparse Matrix Multiplication [54]	Small Matrices	SPMM-small	1025	I	1	(1024, 28)	(256, 23)
	Large Matrices	SPMM-large	5121	I	1	(824, 39)	(256, 72)
Quick Sort [39]	N/A	Quicksort	56	II	24	(48, 28)	(48, 28)
Radix Sort	N/A	Radixsort	4765	II	24	(176, 21)	(176, 21)
Sequence Alignment [11]	Arabidopsis Thaliana [36]	SA	24	I	1	(22, 19)	(289, 16)
Single Source Shortest Path [58]	Citation Network [48]	SSSP-citation	6524	I	23	(2612, 121)	(165, 7)
	Graph 500 [48]	SSSP-graph500	16087	I	6	(5830, 219)	(189, 28)

Table 3. Input characteristics.

Name	Size
Combustion Simulation [28]	Cell count: 150,000,000
Small Graph	Vertices: 1024 Edges: 1,047,552
Citation Network [48]	Vertices: 227,320 Edges: 814,134
Graph 500 [48]	Vertices: 65,536 Edges: 2,456,071
Gaussian Distribution	Array 1: 300,000 Array 2: 300,000
Uniform Distribution	Array 1: 204,800 Array 2: 204,800
Small Matrices	Matrix 1: 512*512 Matrix 2: 512*512
Large Matrices	Matrix 1: 5,120*512 Matrix 2: 512*5,120

Benchmarks: We use ten GPU DP applications from various benchmark suites. For applications whose data access patterns are highly influenced by input data, we also provide various inputs. As shown in Table 2, we call an <application, input> pair as one "benchmark", and there are a total of 16 benchmarks. We also provide the total number of launched kernels in Table 2. As one can see, these DP applications launch many more kernels compared to conventional (static) GPU applications [58], which motivates us to explore the kernel-level data reuse.

For a DP application, there can be two types of kernel launch patterns: Type I in Figure 1(b), and Type II in Figure 1(c). In type I, the launch depth for each parent kernel is 1. After the parent kernel and all its child kernels finish their executions, the host launches another barrier kernel to check some application related criteria in order to decide whether to launch another parent kernel or not. Let us consider BFS, where each thread in a parent kernel is responsible to traverse the edges of a node from the frontier node set that contains the nodes visited in last level of search. Based on the number of edges connected to a node, the parent thread may launch a child kernel to help traverse those edges. Before the next level of search starts, the visited edges and the frontier node set should be updated. Therefore, the host launches a barrier kernel to perform the update and also to check whether the search is over. If there are nodes not visited yet, another parent kernel will be launched from the host to perform the next level of search. Applications that have Type I feature include BFS, `Graph Coloring`, `Relation Join`, `Sparse Matrix Multiplication`, `Sequence Alignment`, and `Single Source Shortest Path`. In type II, the launch depth can be any number up to the hardware limit (24 levels). For example, in `Quick Sort`, the left-pivot array elements are sorted by a child kernel, whereas the right-pivot array elements are sorted by another child kernel. As a result, the kernel launch pattern is similar to a binary tree. If the maximum depth is reached or there are very few elements for processing (i.e., below the *threshold*), bubble sort is used to avoid any potential overheads involved in launching additional child kernels. Applications that have type II feature include AMR, `Mandelbrot Set`, `Quick Sort`, and `Radix Sort`. We list the types of each benchmark in Table 2. In the table, the number next to the type indicates either the number of barrier kernels in Type I, or the launch depth in Type II. We also provide the inputs used to execute our applications in Table 3.

3.4 Data Locality in DP

Prior works have shown that irregular applications exhibit data locality [7, 8, 45]. However, this data locality is hard to capture due to the dynamic, unpredictable, and divergent memory access patterns that generate it. As a result, they are not exploited well in current GPU architectures. By using DP, some of the data reuse is exposed between the boundary of kernels, WGs, and wavefronts due to child kernel launches. Specifically, a parent thread generally prepares or pre-processes the data before launching its child kernel. The child kernel operates on this data and returns the result back to the parent kernel. This "producer-consumer" relationship introduces *temporal* data locality between the parent-child kernels, WGs, and wavefronts. Meanwhile, child kernels (WGs or wavefronts) operate on neighboring data elements in the data layout. Since GPUs generally have large cachelines, *spatial* data locality among sibling-sibling kernels (WGs or wavefronts) is also exposed.

4 REUSE CHARACTERIZATION

In this section, we conduct an in-depth characterization of data reuses at different schedulable units for our benchmarks listed in Table 2. We profile each benchmark using GPGPU-Sim (discussed in Section 3.3) to get the memory footprint traces and analyze data reuse by parsing the memory traces. The memory footprints are extracted at a data block (128 Byte cache line) granularity after memory coalescing.

4.1 Kernel-Level Reuse

We explore three types of kernel-level data reuse based on the kernel relationships: *self-kernel*, *parent-child*, and *sibling-sibling*. To help explain these three different types of data reuses, let us consider a representative kernel launch sequence, shown in Figure 3(a). The parent kernel A launches child kernels B, C and E. Child kernels B and C further launch grandchild kernels D

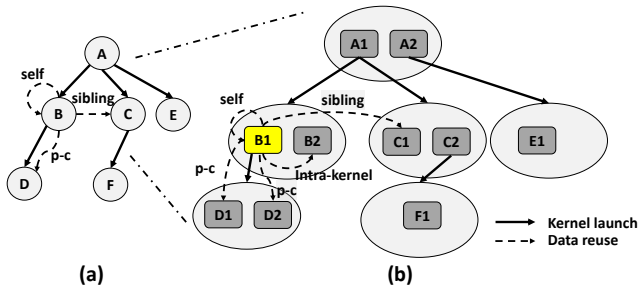

Fig. 3. Launch sequence and data reuse. (a) kernel-level. (b) work-group-level.

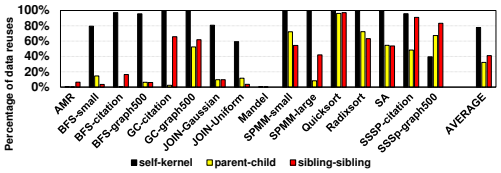

Fig. 4. Quantifying kernel-level data reuse.

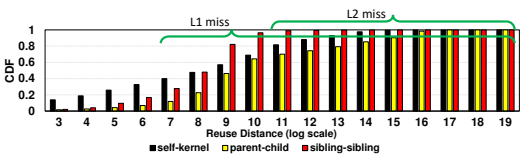

Fig. 5. Reuse distances for kernel-level data reuse.

and F, respectively. A solid line in the figure denotes the kernel launch sequence, while a broken line denotes a potential data reuse. From a particular kernel's perspective (kernel B for example), a *self-kernel* data reuse happens when a data block is referenced multiple times by itself during execution. In comparison, a *parent-child* data reuse happens when a data block is accessed at least once by both the parent kernel and child kernel (e.g., parent B and child D). Similarly, a *sibling-sibling* data reuse is said to occur when a data block is accessed at least once by both the sibling kernels (e.g., B and C). Note that sibling kernels are the child kernels launched from the *same* parent WG. For example, child kernels B and C are considered as sibling kernels; however, child kernels B and E (likewise, D and F) are not. We use this definition due to the fact that the child kernels launched by different parent WGs rarely share any data blocks and are unlikely to work on the same portion of the input data [58].

We count the number of memory accesses to the same data block made by self-kernel, parent-child kernels, and sibling-sibling kernels. Figure 4 plots the percentage of the shared memory footprints over the total memory footprints. Note that, a particular memory access can be counted multiple times toward different types of reuses. For example, a data block accessed by a kernel can also be accessed by its parent kernel and its siblings. As a result, the access to that data block is counted as both parent-child data reuse and sibling-sibling data reuse. This is the reason why the total sum of the three types of reuses can exceed 100% in Figure 4. From this figure, we make the following *critical observations*:

Observation 1: There exists significant data reuse in DP applications at a kernel granularity. Data blocks are heavily reused across all three types of kernel relationships. Specifically, on an average, across all 16 benchmarks, self-kernel, parent-child kernel, and sibling-sibling kernel account for 77.8%, 32.2%, and 41.1% of the total data reuse with respect to the total memory footprint, respectively. Recall the **Question 1** from Section 2, our characterization results show that data blocks are frequently reused among different kernels in these DP applications.

Observation 2: The amount of data reuse is different across different applications. For instance, applications such as `AMR` and `Mandel` do not have much data reuse for any of the three types of kernel relationships. This is because these two applications are compute-intensive with few data reuse.

All the other applications have significant data reuses in different types of kernel relationships. This is due to the significant data reuse exposed by the algorithms implemented in the applications.
Observation 3: For a given application, different inputs can lead to different data reuse patterns. This can be observed in GC, SPMM, and SSSP. For example, SSSP-citation shows significant self-kernel and sibling-sibling reuses, whereas SSSP-graph500 exhibits more data reuse in parent-child and sibling-sibling kernels. This is because the irregularity of input (e.g., a graph) in such applications can cause different number of child kernels to be launched with different kernel dimensions, eventually leading to different data access and reuse patterns. For example, the average number of neighboring nodes in citation graph is relatively small when compared to the graph500 graph. As the threshold which determines child kernel launch is fixed to a smaller number in SSSP by the programmer, SSSP-citation launches fewer as well as smaller child kernels compared to SSSP-graph500. On the other hand, the child kernels in SSSP-graph500 process more neighboring nodes than the child kernels in SSSP-citation. Consequently, SSSP-graph500 has more parent-child data reuse, but less self-kernel data reuse than SSSP-citation.

Next, we perform a study on characterizing the reuse distances of the applications. We define "reuse distance" as the number of *unique* data blocks between two references to the same data block. Generally, references to the same data block with short reuse distances are expected to hit in the caches. Figure 5 gives the CDF of the average reuse distances (in log_2 scale) between the references to the same data block for three types of data reuses, across all the benchmarks. From the figure, we make the following observations. First, in DP applications, the reuse distances of all three reuse types are generally larger than the GPU caches can take advantage of. For example, in self-kernel reuse, more than 60% and 25% of the data blocks referenced by the kernels are evicted from L1 cache and L2 cache, respectively. Second, the distances exhibited by the parent-child data reuses are longer when compared to the distances exhibited by the self-kernel and sibling-sibling data reuses. This is because the child launch overhead (discussed in Section 3.2) delays the child kernel execution, leading to long distances in parent-child data reuses. This provides an answer to **Question 2** from Section 2. As these child kernels are generally launched in bursts, they execute concurrently, leading to shorter reuse distance for the sibling-sibling relationship.
Takeaway: Unlike traditional GPU applications (i.e., those with static/compile-time parallelism), DP applications launch massive numbers of kernels (children). In addition to the data blocks being reused within a kernel, these light-weight child kernels exhibit high data reuses with their parent kernels as well as among themselves. However, long reuse distances prevent the underlying GPU caches from taking advantage of most reused data blocks.

This motivates us to explore a locality-aware kernel scheduling strategy which can schedule kernels with high degrees of data reuse among themselves close to each other during execution. However, we first need to quantify the "degree of data reuse" between the two kernels. In other words, we need to determine the "strength" of the kernel-to-kernel relation (in terms of the "intensity" of data reuse). Kernels having intensive data reuse among themselves should have a higher priority to be scheduled together when compared to kernels rarely having any data reuse. To this end, we define *Reuse Ratio* as a measure of the degree of data reuse among kernels.
Self-Kernel Reuse Ratio (R_k): Given a kernel k, the memory footprint of kernel k is denoted as M_k. Each entry in M_k is a memory access to a data block. We define the self-kernel reuse ratio, R_k, as:

$$R_k = 1 - \frac{uniq(M_k)}{size(M_k)}, \tag{1}$$

where $uniq(M_k)$ is the number of *unique* data blocks referenced by kernel k, and $size(M_k)$ is the total number of data block accesses. Therefore, $1 - uniq(M_k)/size(M_k)$ captures the fraction of "repeated accesses" to the same data block.

Parent-Child Reuse Ratio (R_{p-c}): Given a parent kernel p and its child kernel c, the traces of memory footprints from parent kernel p and child kernel c are denoted respectively as M_p and M_c. The parent-child data reuse ratio, R_{p-c}, is defined as:

$$R_{p-c} = \frac{size(x, x \in M_c \mid x \in uniq(M_p) \cap uniq(M_c))}{size(M_c)}. \tag{2}$$

The numerator is the total number of data blocks referenced by child kernel c, where each data block is also referenced by parent kernel p at least once. The denominator is the total number of data blocks accessed by the child kernel. Note that a data block can be referenced multiple times by either the parent kernel or the child kernel. Our definition of parent-child reuse ratio captures the "intensity" of data reuses in a child kernel with respect to its parent. Note that, we do not count child-parent kernel reuse ratio as child kernels need to be launched by their parent kernels. For this to happen, the parent kernels already need to be scheduled. Therefore, child-parent kernel reuse ratio does not help in dynamic kernel scheduling. Since a DP application can launch multiple levels of child kernels, a child in level l is considered as a parent in level $l + 1$.

Sibling-Sibling Reuse Ratio (R_{cx-cy}): Given two child kernels c_1 and c_2 launched by the *same* parent WG p_{wg}, the memory footprints of these two child kernels are denoted as M_{c1} and M_{c2}, respectively. We define the sibling-sibling data reuse ratio, R_{cx-cy}, as:

$$R_{cx-cy} = \frac{size(x, x \in M_{cx} \mid x \in uniq(M_{cx}) \cap uniq(M_{cy}))}{size(M_{cx})}, \tag{3}$$

where (cx, cy) can be either $(c1, c2)$ or $(c2, c1)$ which represent the reuse ratio in terms of child kernel $c1$ or $c2$, respectively. It is important to note that, we separate the reuse ratios for the two child kernels in the child kernel pair because doing so allows us to capture the scenario where two child kernels have different memory footprint intensity. For example, let us consider the scenario in Figure 6. Suppose child $c1$ has 100 accesses, child $c2$ has 50 accesses, and child $c3$ has 200 accesses. Let us assume that the data blocks referenced by 30 accesses from $c2$ are also referenced by $c1$. In this case, the reuse ratio R_{c2-c1} is $30/50 = 0.6$. However, there can be 40 accesses from $c1$ that reference the same set of data blocks. This is due to the fact that a data block can be referenced multiple times by a kernel. As a result, the reuse ratio R_{c1-c2} is calculated as $40/100 = 0.4$.

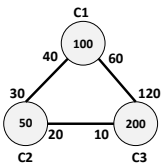

Fig. 6. Sibling-sibling data reuse.

It is important to emphasize that the reuse ratio does *not* capture the absolute number of data blocks being shared between the involved schedulable units. For example, both R_{c2-c1} and R_{c3-c1} are 0.6. However, there are 120 data blocks accessed by $c3$ that are also accessed by $c1$, whereas only 30 data blocks accessed by $c2$ that are also accessed by $c1$. Although the absolute value would be more accurate to represent the quantity of data reuses, it is less effective in managing caches. For instance, a kernel pair having one million total accesses with a 10% reuse ratio will have many more data blocks being reused when compared to a kernel pair having one thousand total accesses with 90% reuse ratio. However, the first kernel pair is not friendly to caches: there are 90% of data blocks that are not being reused, and therefore, it may lead to severe cache contention and poor cache performance.

Results: Figure 7 shows the data reuse ratio at the kernel-level for all 16 benchmarks. We divide the data reuse ratio into 10 ratio bins ($b0 - -b9$), with the stride size of 0.1. All 10 ratio bins are labeled on the X-axis. The Y-axis plots the CDF of the kernel pairs, which captures the number of kernel pairs that fall into the different ratio bins. Specifically, for two kernels in a kernel pair, we first calculate the three types of data reuse ratios using Equations (1)–(3) given above. Then, for each ratio bin, we count the number of kernel pairs whose reuse ratio falls into that bin. For example, if two kernels in a kernel pair have, say, 0.46 parent-child reuse ratio, we count this kernel

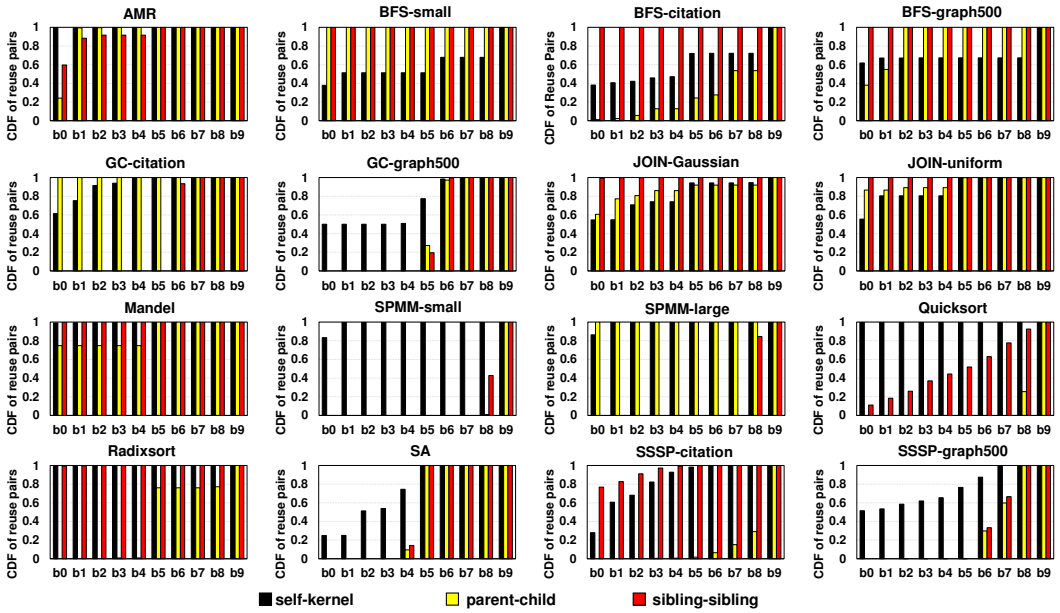

Fig. 7. Kernel-level data reuse ratios for all benchmarks. Each plot includes the results of one benchmark. The X-axis represents the data reuse ratio. The reuse ratio is divided into 10 bins (b0, b1, ..., b9), using 0.1 as stride size. If the reuse ratio of a kernel pair is between $(x, x + 0.1)$, that kernel pair is counted in bin bx. The Y-axis represents the CDF of reuse pairs. The black bar in each plot represents the CDF of self-kernel reuse, whereas the yellow and red bars represent the CDFs of parent-child and sibling-sibling reuses, respectively.

pair in the ratio bin $b4$. From the cumulative results shown in Figure 7, we make the following important observations:

Observation 1: Different benchmarks show different kernel pair distributions in all three types of kernel reuse ratios. For benchmarks `AMR` and `Mandel`, most of the kernel pairs have low self-kernel reuse ratio (less than 0.1) as these two benchmarks have few data blocks being reused (Figure 4). For benchmarks such as `GC-citation`, `GC-graph500`, `SPMM-small`, `SPMM-large`, `Radixsort`, `SA`, `SSSP-citation` and `SSSP-graph500`, most of the parent-child kernel pairs and/or sibling-sibling kernel pairs have similar reuse ratios. For example, in `SA`, 85% of parent-child kernel pairs fall in to $b5$. Benchmarks such as `Quicksort`, `BFS-small`, `BFS-citation`, `BFS-graph500`, `JOIN-Gaussian` and `JOIN-Uniform`, have a uniform distribution in one or more of their kernel relationships. For example, in `Quicksort`, the sibling-sibling kernel pairs are uniformly distributed across the 10 reuse bins.

Observation 2: For applications such as `SPMM`, `BFS`, `GC`, and `SSSP`, different inputs can lead to different distributions of kernel pairs in terms of their reuse ratios. For example, when `SPMM` is used with small sparse matrices (`SPMM-small`) as inputs, we see that there is a significant fraction of parent-child and sibling-sibling kernel pairs having high reuse ratios, whereas, with large sparse matrices (`SPMM-large`) as input, the number of parent-child kernel pairs with high reuse ratio reduces significantly (i.e., most of the parent-child kernel pairs fall into bin $b0$). The main reason is that the *threshold* (discussed in Section 3) set by the programmer significantly affects the number of child kernels and their properties. Therefore, for large matrices, parent thread always opts to launch child kernels to perform the multiplications in child kernels (due to more elements per row and column), whereas for small matrices, the multiplications are usually performed by the parent thread itself in an iterative fashion as there are fewer elements in the rows and columns of the input matrices.

Observation 3: By comparing Figure 4 and Figure 7, we observe that a benchmark with a high "data reuse" does *not* necessarily have high "reuse ratio". This is because, data reuse is measured as an *aggregated* metric, whereas the reuse ratio is measured as a *pair-wise* metric. For example, `SPMM-small` has high self-kernel data reuse than parent-child and sibling-sibling kernel data reuses (Figure 4). However, all of the parent-child kernel pairs fall into $b9$ in Figure 7 and all sibling-sibling kernel pairs fall into $b8$ and $b9$, but most of the self kernel pairs fall into $b0$ and $b1$. This discrepancy is because the reuse ratio is normalized to the memory footprints of two kernels whereas data reuse is computed using the memory footprint of the entire application. As we discussed earlier, reuse ratio is a more effective metric in managing caches.
Takeaway: Our results show that DP applications introduce parent-child and sibling-sibling kernel relationships in addition to the self-kernel relationship. The reuse distance analysis indicates that GPU cache system is unable to take advantage of inherent data reuses in DP applications. We define and analyze reuse ratio to indicate the "intensity" of the data reuses among kernels, which is used later to design a locality-aware kernel scheduling mechanism.

4.2 Work-group-Level Reuse

Work-groups are the smallest granularity of scheduling at the CU level. Note that, while it may seem possible to extend the kernel level data reuse information (discussed above) to the WG level, it is not the case. This is due to the fact that the kernels from a given application might have different number of WGs, and some of these WGs contribute to significant data reuse whereas others may not. Conceptually, we need to understand the data reuses among WGs in order to map WGs to CUs in a "locality-aware" fashion, and schedule them to execute close to each other in time to take full advantage of the per CU L1 cache. Similar to kernel level data reuse, we explore data reuse along four types of WG relationships: *self-WG*, *intra-kernel-WG*, *parent-child-WG*, and *sibling-sibling-WG* (shown in Figure 3(b)). Self-WG reuse and intra-kernel-WG reuse are defined within a kernel boundary (i.e., intra-kernel), whereas parent-child-WG reuse and sibling-sibling-WG reuse are defined across different kernels (i.e., inter-kernel). Specifically, for a given WG ($B1$ from kernel B highlighted in Figure 3(b)), self-WG reuse captures the fraction of data blocks that are accessed multiple times by that same WG. Intra-kernel-WG reuse captures the data blocks reused between the WGs within the same kernel ($B1$ and $B2$). Parent-child-WG reuse is measured between the parent WG ($B1$ in the example) and all WGs from its launched child kernels ($D1$ and $D2$). Finally, sibling-sibling-WG reuse captures the data blocks reused among the WGs that belong to the sibling kernels (e.g., $B1$ and $C1$).
Self-WG Reuse Ratio: Similar to self-kernel reuse ratio calculation, self-WG reuse ratio can be calculated using Equation (1) by replacing kernel with WG. Specifically, for a particular WG wg, $R_{wg} = 1 - uniq(M_{wg})/size(M_{wg})$.
Intra-kernel-WG Reuse Ratio: Given two WGs wg_i and wg_j from a kernel k, we define intra-kernel-WG reuse ratio using Equation (3) by replacing (cx, cy) with (wg_i, wg_j). We want to emphasize that this equation calculates the reuse ratio in terms of WG wg_i. That is, the ratio $R_{wg_i-wg_j}$ may *not* be the same as $R_{wg_j-wg_i}$.
Parent-Child-WG Reuse Ratio: Given a parent WG wg_p and a WG wg_c from a child kernel launched by wg_p, we define the parent-child-WG reuse ratio using Equation (2) by replacing p and c with wg_p and wg_c, respectively. One may notice that this definition only captures the reuse with respect to child WG wg_c. Ideally, we should also calculate the reuse ratio with respect to that parent WG. However, this is not necessary in practice. This is because, in order to schedule a child WG with its parent WG, the child kernel should be first launched and be ready to execute. That is, for a child WG to be visible to the scheduler (i.e., for a child kernel to be launched by the parent WG), its parent WG must be already scheduled and running on the CU. Thus, we just need to focus on

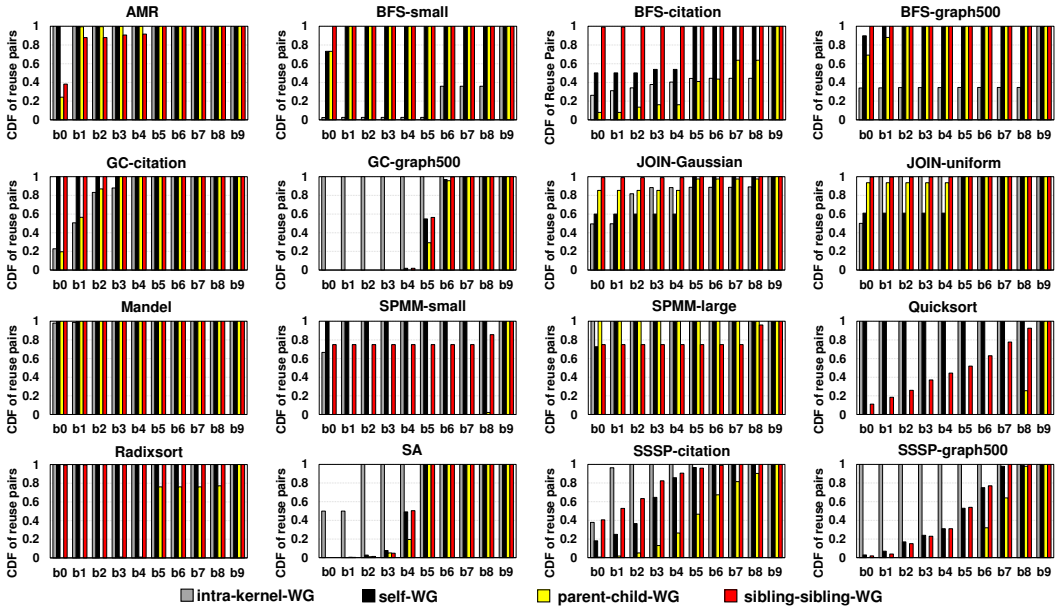

Fig. 8. WG-level data reuse ratio. The X-axis represents 10 bins (b0, b1, ..., b9) of data reuse ratio with 0.1 as stride size. The Y-axis represents the CDF of WG pairs. The gray bar captures intra-kernel-WG. The black bar represents the self-WG, and the yellow and red bars represent parent-child-WG and sibling-sibling-WG, respectively.

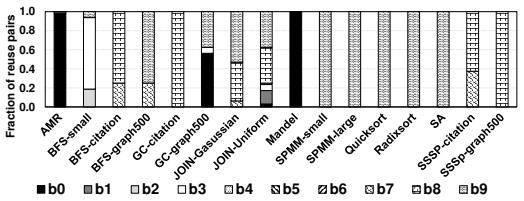

Fig. 9. Intra-wavefront data reuse.

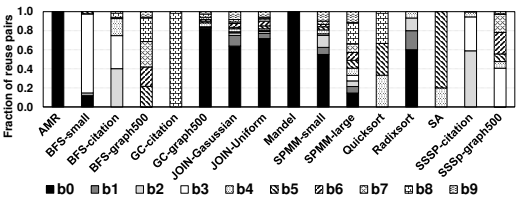

Fig. 10. Inter-wavefront data reuse.

where to schedule the child WG based on the child WG reuse ratio, not the parent WG, as it would be already running.

Sibling-Sibling-WG Reuse Ratio: Sibling-sibling-WG reuse ratio is measured among the WGs that belong to sibling kernels. The formal definition is similar to that of the sibling-sibling kernel reuse ratio and can be obtained by replacing the kernel information with WG information. It can be treated as a finer granularity of data reuse compared to the sibling-sibling kernels and can be calculated using Equation (3).

Results: Figure 8 shows the cumulative distribution of WG pairs for each benchmark. The four bars in each plot of this figure represent the four types of WG relationships.

We make the following observations from these results:

Observation 1: Similar to kernel level, the WG pair distributions of intra-kernel-WG, self-WG, parent-child-WG and sibling-sibling-WG are also application and input dependent.

Observation 2: By comparing Figure 7 and Figure 8, we observe that, in benchmarks `AMR`, `BFS-citation`, `BFS-graph500`, `GC-graph500`, `JOIN-Gaussian`, `JOIN-uniform`, `Quicksort`, `Radixsort`, `SA` and `SSSP-graph500`, the distribution of the WG pairs on a particular type of reuse ratio follows a similar trend to the distribution of kernel pairs on the same type of reuse ratio. In other words, the reuse ratio can translate from kernel level to WG level, due to the fact that a kernel consists

of multiple WGs. It is also interesting to observe that self-kernel reuse can translate to either intra-kernel-WG reuse, or self-WG reuse, or both. For example, in `BFS-graph500` (see Figure 7 and Figure 8), the self-kernel reuse translates to intra-kernel-WG reuse. However, in `BFS-citation`, self kernel reuse translates to both intra-kernel-WG and self-WG reuse. This is because the child kernels contain more WGs in `BFS-graph500` compared to the child kernels in `BFS-citation`. For benchmarks `BFS-small`, `GC-citation`, `Mandel`, `SPMM-small`, `SPMM-large` and `SSSP-citation`, the WG pair distribution is different from the kernel pair distribution. The is because that most of these applications contain branches in their kernel codes. Based on the inputs, the runtime branch conditions are different across WGs in a kernel. As a result, WGs may execute different paths, leading to different data reuse patterns at the WG level.

Observation 3: If two kernels do not reuse data blocks, all the WGs from these two kernels do not reuse data blocks either. For example, `Radixsort` and `SPMM-small` have very few kernels with high self-kernel reuse ratio. Consequently, these two benchmarks also have very few WGs with high self-WG and intra-kernel-WG reuse ratios[2]. However, this is not true in the inverse case. If two kernels have high data reuse between them, this does *not* guarantee that all the WGs from the two kernels will have high data reuse. For instance, more than 50% of the sibling-sibling kernels in `SPMM-small` have reuse ratios larger than 90%. However, at the WG granularity, 75% of the sibling-sibling WGs have reuse ratio less than 10% and the remaining 25% have reuse ratio more than 80%. This disparity arises due to the branch instructions in the kernel code, which lead to divergent paths across WGs, as discussed in Observation 2.

Takeaway: Our results indicate that not only neighboring WGs significantly share data blocks (**Question 3**), but also the parent-child WGs and sibling-sibling WGs (**Question 4**). This information is helpful in assisting WG-to-CU mapping. Specifically, WGs with high reuse ratios should be scheduled on the same CU and executed in close proximity in time, in order to take advantage of the per-CU L1 cache. On the other hand, WG with high self-WG reuse ratios should be scheduled to low-load CUs in order to reduce the L1 cache contention.

4.3 Wavefront-Level Reuse

Wavefront is the smallest schedulable unit and it is the granularity at which the GPU executes instructions. Therefore, wavefront scheduler can impact data locality significantly. To quantify the data reuse in wavefronts, we characterize the *intra-wavefront* and the *inter-wavefront* data reuses for all our benchmarks. For a wavefront w, the intra-wavefront reuse is quantified using Equation (1) by replacing kernel k with wavefront w. Similarly, given two wavefronts w_x and w_y, the inter-wavefront reuse is quantified using Equation (3) by replacing (cx, cy) with (w_x, w_y). Intuitively, one can still define self-wavefront, parent-child-wavefront, and sibling-sibling-wavefront relationships and analyze data reuses along those relationships. However, we choose to classify data reuses into inter-wavefront and intra-wavefront reuses, due to following two reasons. First, wavefronts are mapped to CUs at a WG granularity. In other words, all wavefronts from the same WG are mapped to a particular CU at the time that WG gets scheduled. Second, once mapped to a CU, wavefronts cannot migrate or be reassigned to other CUs, as WG migration is not supported. These two reasons limit the capability of scheduling "any" set of wavefronts together on the same CU. For example, suppose that two wavefronts from two sibling WGs have high inter-wavefront reuse between them. In order to convert the data reuses between these two wavefronts into data locality (cache hits), the two sibling WGs have to be scheduled first. As there are limited hardware resources, it is impossible to schedule all the WGs together to have all the wavefronts ready for the wavefront scheduler.

[2]Both self-WG and intra-kernel-WG are within a kernel boundary, and a low self-kernel reuse leads to a low self-WG and intra-kernel-WG reuses.

Algorithm 1 Locality-aware kernel scheduling

INPUT:
List_k : kernels launched in pending kernel pool
1: **while** HWQ q_i is empty **do**
2: **for** kernel k_x from the head of HWQ q_j, where $i \neq j$ **do**
3: Search for high reuse kernels with k_x
4: $high\_reuse(k_x) \leftarrow (k_{y1}, k_{y2}, \ldots, k_{yn})$
5: **if** k_{y1} is also in List_k **then**
6: schedule k_{y1} to q_i and remove k_{y1} from List_k

Once both the WGs are scheduled, it is possible for the wavefront scheduler to exploit the reuse between these two wavefronts. This is captured as inter-wavefront locality. We characterize the intra-wavefront and inter-wavefront reuse ratios in Figure 9 and Figure 10, respectively. We divide reuse ratio into 10 bins ($b0 - b9$), and for each benchmark, the Y-axis reports the percentage of wavefront pairs that fall into different bins.

Takeaway: Overall, DP applications have significant number of wavefronts exhibiting intensive intra-wavefront reuse. Moreover, there is also a sizable fraction of wavefronts showing intensive inter-wavefront reuse. Since a wavefront can offload its computations to other wavefronts by launching child kernels, some of the intra-wavefront data reuses are transferred to inter-wavefront data reuse between parent wavefront and child wavefronts [58].

5 OPTIMAL LOCALITY-AWARE SCHEDULER: A LIMIT STUDY

In this section, we propose an "optimal" scheduling mechanism to realize the maximum potential performance gains that can be achieved by leveraging data reuses in DP applications. Note however that, this optimal scheduler only has *a priori* knowledge about the data reuse patterns in the DP benchmarks and cannot change any application (data dependency, correctness, etc.) or hardware (occupancy limits, cache replacement policies, etc.) constraints to improve data locality. For this limit study, we profile all the benchmarks and analyze the reuse ratio among the schedulable units at kernel, WG and wavefront granularities. Then, we discuss scheduling policies that choose the appropriate schedulable units based on the reuse ratios. Note that, the scheduling policies in this limit study are *not* implementable in practice. Instead, they reveal the "optimal" benefits one can get from realizing data reuse in DP applications.

Challenges: There are several challenges involved in building an optimal scheduling mechanism. First, hardware constraints limit the effectiveness of the schedulers. For example, high reuse kernels should be launched to different HWQs for concurrent execution, but the number of HWQs limits the number of kernels that can execute concurrently. Second, in all the three granularities of reuses studied, high reuse schedulable units can form "reuse chains". For example, if kernel $k1$ has a high reuse ratio with kernel $k2$, and $k2$ also has high reuse with kernel $k3$, all three kernels collectively form a *high reuse chain*, $k1$-$k2$-$k3$. We quantify maximum as well as average high reuse chain length at kernel and WG granularities (shown in Table 2). For the limit study, we define "high reuse" as a reuse ratio of greater than 0.4. Since there are limited hardware resources (e.g., register file) and limited concurrency, it is impossible to always schedule the entire chain to hardware for concurrent execution. For example, in `Radixsort`, the maximum WG chain is 176, and as a result, it is not possible to find a CU and schedule all the 176 WGs in that CU. Third, scheduling an entire reuse chain of WGs into a CU can also lead to workload imbalance, causing some of the CUs to have more computations, while other CUs are idle. The WG scheduler should be able to dynamically *balance* the workload among CUs without compromising much on data locality.

Algorithm 1 provides the high-level pseudo-code for optimal kernel scheduler. Whenever there is an empty slot in any HWQ, the kernel scheduler tries to find a candidate kernel which has the highest reuse ratio with the already-running kernels, and schedules it into the head of that empty

Algorithm 2 Locality-aware WG scheduling

INPUT:
Q_{CU_i}: The CU queue of CU_i.
1: **for** each Q_{CU_i} **do** /**schedule reuse chains to CU queues*/
2: **if** size of Q_{CU_i} is minimum **then**
3: schedule reuse chain (wg_1, wg_2, …, wg_n) to Q_{CU_i}
4: **for** each CU_i **do** /**schedule WGs on CU*/
5: **if** CU_i can issue a WG **then**
6: **if** Q_{CU_i} is empty **then** Steal an WG from other CU queue.
7: **else** issue WG from Q_{CU_i}

HWQ. In other words, it tries to *maximize* the potential chances of running high reuse kernels concurrently. A significant benefit of our scheduling strategy is that it facilitates the subsequent locality-aware WG scheduling. More specifically, the WG scheduler is free and safe to choose any WGs from these high reuse kernels and schedule the selected WGs into CUs as long as there are enough hardware resources. It should also be noted that, dependencies among parent kernels and child kernels are rare in dynamic applications [38]. However, if there is a dependency, the dependent kernels are labeled with same software queue ID (e.g., CUDA stream), which in turn is mapped to the same HWQ for sequential execution. Our scheduler tries to co-locate parent and child kernels during scheduling such that the data reuse between dependent kernels are captured.

Algorithm 2 shows the WG scheduling policy used in our limit study. We define reuse ratio to be "high" if it is greater than 0.4. We form the high reuse chains of WGs based on the characterization results and import the high reuse chains to Algorithm 2 for scheduling. To take high reuse WG chains into consideration, we associate each CU with a CU queue. At runtime, we schedule high reuse WG chains into these CU queues. Note that the WGs in CU queues do not occupy CU resources (e.g., hardware threads, register file). Once a CU has sufficient available resources, it selects a candidate WG from its CU queue. Note that, scheduling an entire WG chain onto a single CU can lead to load imbalance (in terms of the number of WGs assigned) across different CUs. This is due to the different lengths of the WG reuse chains and can lead to significant GPU under-utilization and performance degradation. To avoid load imbalance, every time a WG chain is to be scheduled, all the CU queues lengths are checked. The WG chain is assigned to the CU queue with the least number of WGs in its queue. To tackle the issue of load imbalance, we enable *WG stealing* across the CU queues. Specifically, if a CU has available resources and its associated CU queue is empty, it steals a WG from another CU queue. Note that WGs that have been scheduled and are executing on a CU cannot migrate.

Once a WG is scheduled on a CU, the wavefronts in the WGs are mapped to the hardware wavefronts. There are a total of 64 hardware wavefronts in our baseline GPU, and the default wavefront scheduler is GTO [46]. GTO scheduling performs well in achieving intra-wavefront data locality. However, it is not as effective in exploiting inter-wavefront data locality. To address this, we enhance the two-level wavefront scheduler [35]. More specifically, if two WGs have high reuse ratio, the wavefronts from these two WGs are grouped together and executed in a round-robin fashion. To select between the groups, GTO is used. For example, let us assume four WGs: $wg1 = (w11, w12)$, $wg2 = (w21, w22)$, $wg3 = (w3)$, and $wg4 = (w41, w42)$. Suppose that work-groups $wg1$, $wg2$ and $wg3$ have high data reuse ratio among themselves, and $wg4$ has high self-WG reuse. Let us further assume all four WGs are running on the same CU. In this case, we make two groups of wavefronts based on the WG reuse ratios: $group1 = (w11, w12, w21, w22, w3)$ and $group2 = (w41, w42)$. First, GTO is used to select between the two groups, and once the group is selected, the wavefronts within the group are executed in a round-robin fashion.

Note that, although we use the profiled data reuse information to guide our scheduling, we still cannot achieve the optimal data locality along with the optimal performance. The reason

for this is two-fold. First, there is a tradeoff between parallelism (workload balance) and data locality. For instance, to guarantee CU occupancy, we have to schedule a WG whenever a CU has available resources. Consequently, if a high reuse WG from a child kernel is not available to the scheduler (e.g., due to launch overhead), we do not want to reserve the CU resources while waiting for a high reuse WG to be scheduled. This is because leaving a CU under-utilized reduces the effectiveness on tolerating long latency operations and leads to performance degradation. Second, GPUs usually have smaller caches compared to CPUs. Even with our locality-aware scheduling strategies, it is not guaranteed that all of the reused data blocks can remain in the cache when they are reused. To further improve data locality in GPUs, our scheduling strategies can co-exist with other locality-aware cache optimizations [17, 27, 41].

Implementation Issues: It is not feasible for the optimal scheduler to be implemented in practice. This is due to two main reasons. First, the data reuse information that is needed at runtime is not known *a priori*. Unlike regular applications, where profiling some *training* applications/inputs to build a prediction model can help predict the reuse information for new applications/inputs [12, 50], it is not possible to do the same for DP applications. This is due to the irregular and unstructured behavior of the applications and their inputs, as discussed in Section 4. Second, even if the data reuse information was known *a priori*, the bookkeeping and hardware overheads of implementing an optimal scheduler in practice would be too high due to the increase on area and power costs.

6 LASER – LOCALITY-AWARE SCHEDULER: A PRACTICAL APPROACH

In this section, we distill the observations from our data reuse characterization and limit study presented above, and propose LASER, a Locality-Aware SchedulER, that makes scheduling decisions based on the reuse ratios. The reuse ratios are computed dynamically at runtime with minimal hardware overheads and no profiling requirements. Figure 11 depicts the necessary architectural support required to implement LASER. We modify the baseline GPU architecture by extending/adding components in the GMU, WG scheduler, and CUs.

GMU: In the baseline GPU, the newly launched kernels (device kernels) are either directly launched into the HWQs in the GMU or temporarily "stored" in the pending kernel pool (a queue based structure) if there are no empty slots in HWQ. We partition the pending kernel pool to have two priority queues: High-Priority Queue (HPQ) and Low-Priority Queue (LPQ) Ⓑ. A kernel is queued to either HPQ or LPQ based on the priority flag associated in the kernel instance Ⓐ. The priority flags of child kernels are set at the time the parent thread launches the child kernels. The priority values are determined based on the outputs of reuse monitors (discussed later in this section) located in each CU. Once there is an empty slot in HWQ, the GMU selects the kernel at the head of HPQ and only selects a kernel from LPQ when HPQ is empty Ⓒ. Since we partition the pending kernel pool into HPQ and LPQ without increasing the pool capacity, the only incurred hardware overhead is adding one more read port and one more write port to the pending kernel pool.

WG scheduler: Recall that, in the baseline GPU scheduler, WGs from "head-of-queue" kernels in HWQs are scheduled to CUs in a round-robin fashion. The WG scheduler keeps tracks of the necessary WG scheduling information such as next WG to be scheduled and WG dimension for every kernel. In LASER, we extend the information table to include the parent information (i.e., parent kernel ID and WG ID Ⓓ). We also add a new table called Schedule Status Table (SST) in the WG scheduler to track the running WGs on each CU Ⓔ. Each entry in SST contains the information of the running WGs in the form of (k_id, WG_id) pair. Since each CU can have a maximum of 16 WGs resident [40], each table entry for a CU contains information from a maximum of 16 WGs.

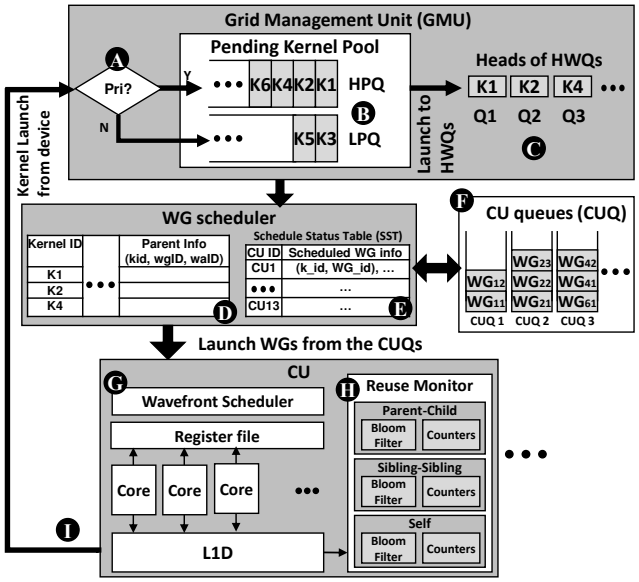

Fig. 11. Architectural support for LASER.

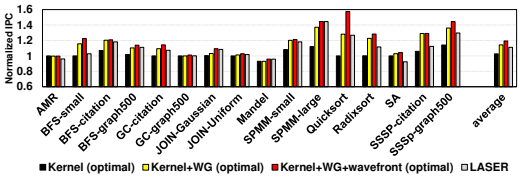

Fig. 12. IPC normalized to baseline scheduling.

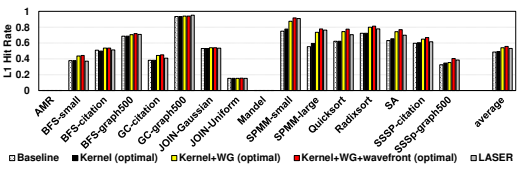

Fig. 13. L1 hit rates.

Therefore, we need 1664 bytes for the hardware SST [3]. A (k_id, WG_id) pair is inserted into the SST once the WG is scheduled to a CU and removed from the SST when the WG finishes execution and relinquishes its occupied resources. For scheduling a child WG, the WG scheduler relies on its parent's information and the information from SST, and schedules it on a CU where the parent WG is already running. As a result, parent-child WG reuse is captured. Recall that multiple WGs can have high data reuse among them and form reuse chains. To preserve the data reuse in reuse chains, we add a CU queue (CUQ) F for each CU. CUQs serve two purposes: (1) to ensure that the high reuse WGs are executed by the same CU as much as possible, and (2) to enable WG stealing across different CUs in order to avoid workload imbalance. The CUQs are mapped onto CUs in a one-to-one fashion. The WG scheduler always tries to schedule WGs to a CU from its CUQ (e.g., CUQ 1 to CU 1), and a CU only steals a WG from another CUQ if its CUQ is empty, and it has enough available resources for a new WG to be scheduled. We implement CUQs in the GPU's global memory such that the hardware overheads are minimized [57].

Compute Unit (CU): We modify/add two components in each CU: (1) the wavefront scheduler G, and (2) the reuse monitor H. As discussed in Section 5, we use a two-level wavefront scheduler to leverage wavefront-level data reuse. Specifically, if a child kernel is predicted to have high parent-child data reuse, all the child wavefronts are grouped with its parent wavefront (if it is not

[3] 1664 bytes is calculated by 13 CUs with each CU has maximum 16 WGs. For each WG, we track k_id and WG_id with each 4 bytes.

finished yet) in the same group and round-robin wavefront scheduler is applied within the group. Note that, as the majority of child kernels are light-weight [58] (i.e., they contain few WGs and each WG has few wavefronts), the total number of wavefronts in a child kernel is small, making the group size relatively small.

Reuse Monitor: Whenever there is a child kernel launch from a CU, we estimate the reuse type (i.e., parent-child, sibling-sibling, and self) of that particular child kernel. This is done through the information obtained from the reuse monitor Ⓗ in each CU. The reuse monitor consists of three Bloom filters, one for each type of data reuse: parent-child kernel, sibling-sibling kernel, and self kernel reuse. Each Bloom filter is associated with two counters: number of hits and number of misses. For each L1 cache read request, the accessed cacheline address is checked in the Bloom filter and the corresponding counters are updated. The address is then added to the Bloom filter based on the kernel type (e.g., only parent kernel adds to the parent-child Bloom filter and only child kernel adds to the sibling-sibling Bloom filter). The reuse monitor predicts the child kernel to be of parent-child reuse type if the "hit rate" of the parent-child Bloom filter is greater than a predefined threshold, and is larger than the hit rates of sibling-sibling and self-kernel. We empirically set the threshold to be 0.5. Similarly, for sibling-sibling and self-kernel, the reuse monitor predicts the reuse type based on the corresponding hit rates of Bloom filters. If a child kernel is predicted to be parent-child type or sibling-sibling type, the kernel is labeled a high-priority. Otherwise, if it is predicted self-reuse type or no reuse, it is labeled a low-priority. We use MurmurHash2 [4] as the hash function in the Bloom filter.

At the beginning of application execution, the very first parent kernel is launched by the host CPU and pushed into HPQ in the GMU. Due to the lack of reuse information at the initial stage of execution, the WGs in that parent kernel are scheduled to CU queues in a round-robin fashion for load balance. In order for the reuse monitor to capture parent-child WG reuses, the first couple of child kernels are launched to the HPQ and child WGs are scheduled to the same CUs where the parent WGs execute. After this stage, further kernel launches are attached with estimated priorities based on information from reuse monitor Ⓘ. The kernel's priority is checked at Ⓐ and is either pushed into the HPQ or the LPQ Ⓑ. When a parent kernel finishes its execution, we reset the Bloom filters but leave the counter values unchanged. Later, when the next parent kernel starts launching child kernels, it uses the counter information to perform the priority prediction as well as update the counters and Bloom filters. The major hardware overheads come from the structure of Bloom filters which are space-efficient data structures. A Bloom filter does not need to account for a lot of entries, especially in applications that exhibit frequent data reuse, since only the misses are added to Bloom filter and hits just update the counters. In LASER, each Bloom filter needs 605 bytes (with 1000 entries and 0.1 false positive probability). As a result, the total hardware overhead is 2 KB for the 3 Bloom filters per CU.

7 EXPERIMENTAL EVALUATION

In this section, we evaluate the effectiveness of our limit study and our proposed scheduling mechanism, LASER. We also compare against two prior efforts targeting data locality on GPUs.

Results of the limit study: The first three bars in Figure 12 show the normalized IPC across all 16 benchmarks from our limit study. The results are *normalized* to the baseline scheduler which uses FCFS kernel scheduling, round-robin WG scheduling, and GTO wavefront scheduling. The first three bars in Our limit study is a three-part study which involves an optimal locality-aware kernel scheduling policy(**kernel**), an optimal locality-aware kernel+WG scheduling policy (**kernel+WG**) and finally, an optimal locality-aware kernel+WG+wavefront scheduling (**kernel+WG+wavefront**). On an average, the three policies achieve performance improvements of 2.6%, 14.3%, and 19.4%, respectively with respect to the baseline scheduling policy. From the results, we make the following

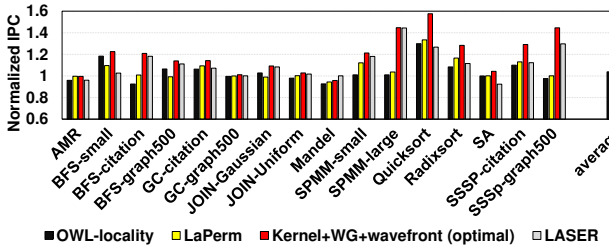

Fig. 14. Performance comparison against prior schemes.

observations. First, for most benchmarks having high data reuses in Figure 4, such as three inputs of BFS and both inputs of SSSP, the performance improvements are also high compared to other benchmarks. This is because we schedule the units (i.e., kernels, WGs, and wavefronts) with high reuse ratios close to each other during execution. Second, for applications AMR and Mandel, the results are similar to baseline since these two applications do not have intrinsic data reuse properties. Third, for benchmarks such as JOIN-Uniform and SA, even though we use accurate reuse information to guide scheduling, some of the data reuse opportunities may not be achievable, especially data reuse along the relationship of parent-child. This is primarily due to the launch overhead. Specifically, child kernels are not available in pending kernel pool immediately after launch. Since our approach can only choose the kernels from the pending kernel pool which contains only ready-to-execute kernels, we fail to exploit some of the parent-child data reuses.

Note that, we do not enable throttling in any of our experimental evaluation. Throttling has been proved to be very beneficial in reducing the cache contention [24, 31]. With throttling enabled, we expect both, the limit study and LASER to provide better performance improvements as it reduces the cache contention.

Evaluation of LASER: The last bar in Figure 12 shows the overall performance of LASER, *normalized* with respect to the baseline scheduler. LASER, on an average, achieves 11.3% performance improvement across the 16 GPU benchmarks tested. We make two observations from the results. First, in benchmarks BFS-citation, SPMM-small, and SPMM-large, LASER performs very well (close to the "optimal"). This is because most of the parent-child kernels pairs and/or sibling-sibling kernel pairs have similar high reuse ratios (i.e., fall into the same bin as shown in Figure 7). As a result, the reuse type prediction in LASER is very accurate. Second, there is still a sizable gap between LASER and the optimal scheduler for few applications. This is due to two reasons. First, for benchmarks such as Quicksort, SSSP-citation and SSSP-graph500, LASER fails to accurately predict the reuse type as the kernel pairs have diverse reuse ratios in these benchmarks. For example, the sibling-sibling pairs in Quicksort are uniformly distributed among reuse ratio bins (see Figure 7). Second, for benchmarks such as BFS-small, Radixsort and SA, the reuse information collection overheads (i.e., the parallelism is compromised at the initial stages of execution where the child kernels are bound to the same CU to collect reuse information) outweigh the performance improvements that we get with the improved data locality.

Figure 13 plots the L1 hit rates with our limit study and LASER. As can be seen, the L1 hit rate increases by enabling hardware schedulers to be locality-aware. The L1 hit rate significantly improves for **kernel+WG** scheduling. This is because that WGs with high reuses are now mapped to the same CU to take advantage of the L1 cache unlike the scenario in **kernel**, where the WGs are scheduled in round-robin. By using LASER, the L1 hit rate is within 3% of the optimal **kernel+WG+wavefront** scheduler.

We next compare LASER with two prior efforts that target data locality in GPUs: *OWL-locality* [21] and *LaPerm* [57]. OWL-locality implements a locality-aware wavefront scheduler to reduce cache contention and improve the latency hiding capability. It enhances the two-level wavefront scheduler

with the knowledge of the data layout to WG mapping. LaPerm, on the other hand, binds child WGs with their direct parent WG, and schedules them on the same CU in a load balanced fashion. Figure 14 shows the comparison of OWL-locality, LaPerm, and LASER along with the optimal scheduling (**kernel+WG+wavefront**) policy. The results are normalized to baseline scheduling. On average, OWL-locality and LaPerm improve performance by 3.8% and 5.7%, respectively, over the baseline, whereas LASER provides 11.3% performance improvement. In summary, DP applications are generally complicated in data access pattern and have data reuses along different types of kernel/WG relationships. As a result, simply co-locating neighboring WGs (as in OWL-locality) or binding parent-child WGs (as in LaPerm) does not fully exploit the data reuse.

8 CONCLUSIONS

Dynamic parallelism is an effective approach for improving GPU performance and resource utilization when executing irregular applications. While there have been prior efforts focusing on resource management and overhead tolerance for dynamic parallelism, the data access patterns and data reuse remain unclear. In this paper, we systematically characterize the data reuse and data locality opportunities that exist in dynamic parallel GPU applications. Based on our observations, we conduct a limit study to show the performance benefits of an "optimal" scheduler that realizes as much data reuse as possible. Furthermore, we propose a practical locality-aware scheduler, called LASER, which makes the GPU hardware schedulers locality-aware, and thus improves data reuse. Our experimental evaluations show that, on an average, 19.4% and 11.3% performance improvements can be achieved with an optimal scheduler and LASER, respectively.

ACKNOWLEDGMENT

We thank Ganesh Ananthanarayanan for shepherding our paper. We also thank the anonymous reviewers for their constructive feedback. This research is supported in part by NSF grants #1526750, #1763681, #1439057, #1439021, #1629129, #1409095, #1626251, #1629915, #1657336 and #1750667. AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies. OpenCL is a trademark of Apple Inc. used by permission by Khronos Group, Inc.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*.

[2] Vignesh Adhinarayanan, Indrani Paul, Joseph Greathouse, Wei N. Huang, Ashutosh Pattnaik, and Wu chun Feng. 2016. Measuring and Modeling On-Chip Interconnect Power on Real Hardware. In *IISWC*.

[3] AMD. 2013. AMD APP SDK OpenCL User Guide. (2013).

[4] Austin Appleby. 2016. Murmur Hash 2. https://github.com/aappleby/smhasher/blob/master/src/MurmurHash2.cpp. (2016).

[5] Sara S Baghsorkhi, Isaac Gelado, Matthieu Delahaye, and Wen-mei W Hwu. 2012. Efficient performance evaluation of memory hierarchy for highly multithreaded graphics processors. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 23–34.

[6] A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt. 2009. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *ISPASS*.

[7] S. Beamer, K. Asanovic, and D. Patterson. 2015. Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server. In *2015 IEEE International Symposium on Workload Characterization*.

[8] Shuai Che, Jeremy W. Sheaffer, Michael Boyer, Lukasz G. Szafaryn, Liang Wang, and Kevin Skadron. 2010. A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*.

[9] Guoyang Chen and Xipeng Shen. 2015. Free Launch: Optimizing GPU Dynamic Kernel Launches Through Thread Reuse. In *MICRO*.

[10] Xuhao Chen, Shengzhao Wu, Li-Wen Chang, Wei-Sheng Huang, Carl Pearson, Zhiying Wang, and Wen-Mei W Hwu. 2014. Adaptive cache bypass and insertion for many-core accelerators. In *Proceedings of International Workshop on Manycore Embedded Systems*. ACM, 1.

[11] Haoyu Cheng, Huaipan Jiang, Jiaoyun Yang, Yun Xu, and Yi Shang. 2015. BitMapper: an efficient all-mapper based on bit-vector computing. In *BMC Bioinformatics*.

[12] Chen Ding and Yutao Zhong. 2003. Predicting Whole-program Locality Through Reuse Distance Analysis. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*.

[13] Wei Ding, Xulong Tang, Mahmut Kandemir, Yuanrui Zhang, and Emre Kultursay. 2015. Optimizing Off-chip Accesses in Multicores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[14] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V Veidenbaum. 2012. Improving cache management policies using dynamic reuse distances. In *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*. IEEE, 389–400.

[15] I. E. Hajj, J. Gomez-Luna, C. Li, L. W. Chang, D. Milojicic, and W. m. Hwu. 2016. KLAP: Kernel launch aggregation and promotion for optimizing dynamic parallelism. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[16] Wenhao Jia, Kelly A Shaw, and Margaret Martonosi. 2012. Characterizing and improving the use of demand-fetched caches in GPUs. In *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 15–24.

[17] Wenhao Jia, Kelly A Shaw, and Margaret Martonosi. 2014. MRPB: Memory request prioritization for massively parallel processors. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE, 272–283.

[18] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia (MM '14)*.

[19] Adwait Jog, Onur Kayiran, Tuba Kesten, Ashutosh Pattnaik, Evgeny Bolotin, Nilardish Chatterjee, Steve Keckler, Mahmut T. Kandemir, and Chita R. Das. 2015. Anatomy of GPU Memory System for Multi-Application Execution. In *MEMSYS*.

[20] Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. 2013. Orchestrated Scheduling and Prefetching for GPGPUs. In *ISCA*.

[21] Adwait Jog, Onur Kayiran, Nachiappan C. Nachiappan, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. 2013. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance. In *ASPLOS*.

[22] Adwait Jog, Onur Kayiran, Ashutosh Pattnaik, Mahmut T. Kandemir, Onur Mutlu, Ravi Iyer, and Chita R. Das. 2016. Exploiting Core Criticality for Enhanced Performance in GPUs. In *SIGMETRICS*.

[23] Mahmut Kandemir, Hui Zhao, Xulong Tang, and Mustafa Karakoy. 2015. Memory Row Reuse Distance and Its Role in Optimizing Application Performance. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*.

[24] Onur Kayiran, Adwait Jog, Mahmut T. Kandemir, and Chita R. Das. 2013. Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs. In *PACT*.

[25] Onur Kayiran, Adwait Jog, Ashutosh Pattnaik, Rachata Ausavarungnirun, Xulong Tang, Mahmut T. Kandemir, Gabriel H. Loh, Onur Mutlu, and Chita R. Das. 2016. uC-States: Fine-grained GPU Datapath Power Management. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT)*.

[26] Orhan Kislal, Jagadish Kotra, Xulong Tang, Mahmut Taylan Kandemir, and Myoungsoo Jung. 2018. Enhancing Computation-to-core Assignment with Physical Location Information. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[27] Gunjae Koo, Yunho Oh, Won Woo Ro, and Murali Annavaram. 2017. Access pattern-aware cache management for improving data utilization in gpu. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 307–319.

[28] AL Kuhl. 2010. Thermodynamic States in Explosion Fields. In *IDS*.

[29] B. C. C. Lai, H. K. Kuo, and J. Y. Jou. 2015. A Cache Hierarchy Aware Thread Mapping Methodology for GPGPUs. *IEEE Trans. Comput.* 64, 4 (April 2015), 884–898. https://doi.org/10.1109/TC.2014.2308179

[30] Minseok Lee, Seokwoo Song, Joosik Moon, John Kim, Woong Seo, Yeongon Cho, and Soojung Ryu. 2014. Improving GPGPU resource utilization through alternative thread block scheduling. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE, 260–271.

[31] Ang Li, Shuaiwen Leon Song, Weifeng Liu, Xu Liu, Akash Kumar, and Henk Corporaal. 2017. Locality-Aware CTA Clustering for Modern GPUs. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 297–311. https://doi.org/10.1145/3037697.3037709

[32] Chao Li, Shuaiwen Leon Song, Hongwen Dai, Albert Sidelnik, Siva Kumar Sastry Hari, and Huiyang Zhou. 2015. Locality-driven dynamic GPU cache bypassing. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 67–77.

[33] Gu Liu, Hong An, Wenting Han, Xiaoqiang Li, Tao Sun, Wei Zhou, Xuechao Wei, and Xulong Tang. 2012. FlexBFS: A Parallelism-aware Implementation of Breadth-first Search on GPU. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*.

[34] Lifeng Nai, Yinglong Xia, Ilie G. Tanase, Hyesoon Kim, and Ching-Yung Lin. 2015. GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions. In *SC*.

[35] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. 2011. Improving GPU Performance via Large Warps and Two-level Warp Scheduling. In *MICRO*.

[36] NCBI. 2016. National Center for Biotechnology Information. http://www.ncbi.nlm.nih.gov. (2016).

[37] NVIDIA. 2011. JP Morgan Speeds Risk Calculations with NVIDIA GPUs. (2011).

[38] NVIDIA. 2012. Dynamic Parallelism in CUDA. (2012).

[39] NVIDIA. 2015. CUDA C/C++ SDK Code Samples. (2015).

[40] NVIDIA. 2018. CUDA Programming Guide. (2018).

[41] Yunho Oh, Keunsoo Kim, Myung Kuk Yoon, Jong Hyun Park, Yongjun Park, Won Woo Ro, and Murali Annavaram. 2016. APRES: improving cache efficiency by exploiting load characteristics on GPUs. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 191–203.

[42] S. I. Park, S. P. Ponce, J. Huang, Y. Cao, and F. Quek. 2008. Low-Cost, High-Speed Computer Vision using NVIDIA's CUDA Architecture. In *AIPR*.

[43] Ashutosh Pattnaik, Xulong Tang, Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, and Chita R. Das. 2016. Scheduling Techniques for GPU Architectures with Processing-In-Memory Capabilities. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT)*.

[44] Sooraj Puthoor, Xulong Tang, Joseph Gross, and Bradford M. Beckmann. 2018. Oversubscribed Command Queues in GPUs. In *Proceedings of the 11th Workshop on General Purpose GPUs (GPGPU collocated with PPoPP)*.

[45] Minsoo Rhu, Michael Sullivan, Jingwen Leng, and Mattan Erez. 2013. A locality-aware memory hierarchy for energy-efficient GPU architectures. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 86–98.

[46] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2012. Cache-Conscious Wavefront Scheduling. In *MICRO*.

[47] Jihyun Ryoo, Orhan Kislal, Xulong Tang, and Mahmut Taylan Kandemir. 2018. Quantifying and Optimizing Data Access Parallelism on Manycores. In *Proceedings of the 26th IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*.

[48] Peter Sanders and Christian Schulz. 2012. 10th Dimacs Implementation Challenge-Graph Partitioning and Graph Clustering. (2012).

[49] Ivy Schmerken. 2009. Wall Street Accelerates Options Analysis with GPU Technology. (2009).

[50] Xipeng Shen, Yutao Zhong, and Chen Ding. 2004. Locality Phase Prediction. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*.

[51] Xulong Tang, Mahmut Kandemir, Praveen Yedlapalli, and Jagadish Kotra. 2016. Improving Bank-Level Parallelism for Irregular Applications. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[52] Xulong Tang, Mahmut Taylan Kandemir, Hui Zhao, Myoungsoo Jung, and Mustafa Karakoy. 2019. Computing with Near Data. In *Proceedings of the 2019 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*.

[53] Xulong Tang, Orhan Kislal, Mahmut Kandemir, and Mustafa Karakoy. 2017. Data Movement Aware Computation Partitioning. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[54] Xulong Tang, Ashutosh Pattnaik, Huaipan Jiang, Onur Kayiran, Adwait Jog, Sreepathi Pai, Mohamed Ibrahim, Mahmut Kandemir, and Chita Das. 2017. Controlled Kernel Launch for Dynamic Parallelism in GPUs. In *Proceedings of the 23rd International Symposium on High-Performance Computer Architecture (HPCA)*.

[55] Panagiotis D. Vouzis and Nikolaos V. Sahinidis. 2011. GPU-BLAST: using graphics processors to accelerate protein sequence alignment. *Bioinformatics* 27, 2 (2011), 182–188.

[56] Jin Wang, Norm Rubin, Albert Sidelnik, and Sudhakar Yalamanchili. 2015. Dynamic Thread Block Launch: A Lightweight Execution Mechanism to Support Irregular Applications on GPUs. In *ISCA*.

[57] Jin Wang, Norm Rubin, Albert Sidelnik, and Sudhakar Yalamanchili. 2016. LaPerm: Locality Aware Scheduler for Dynamic Parallelism on GPUs. In *ISCA*.
[58] Jin Wang and Yalamanchili Sudhakar. 2014. Characterization and Analysis of Dynamic Parallelism in Unstructured GPU Applications. In *IISWC*.
[59] Xiaolong Xie, Yun Liang, Guangyu Sun, and Deming Chen. 2013. An efficient compiler framework for cache bypassing on GPUs. In *Computer-Aided Design (ICCAD), 2013 IEEE/ACM International Conference on*. IEEE, 516–523.
[60] Xiaolong Xie, Yun Liang, Yu Wang, Guangyu Sun, and Tao Wang. 2015. Coordinated static and dynamic cache bypassing for GPUs. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE, 76–88.

Received August 2018; revised October 2018; accepted December 2018