2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture

LaPerm: Locality Aware Scheduler for Dynamic Parallelism on GPUs

Jin Wang* Norm Rubin† Albert Sidelnik† Sudhakar Yalamanchili*
*Georgia Institute of Technology †NVIDIA Research
Email: *{jin.wang,sudha}@gatech.edu, †{nrubin,asidelnik}@nvidia.com

***Abstract*—Recent developments in GPU execution models and architectures have introduced dynamic parallelism to facilitate the execution of irregular applications where control flow and memory behavior can be unstructured, time-varying, and hierarchical. The changes brought about by this extension to the traditional bulk synchronous parallel (BSP) model also creates new challenges in exploiting the current GPU memory hierarchy. One of the major challenges is that the reference locality that exists between the parent and child thread blocks (TBs) created during dynamic nested kernel and thread block launches cannot be fully leveraged using the current TB scheduling strategies. These strategies were designed for the current implementations of the BSP model but fall short when dynamic parallelism is introduced since they are oblivious to the hierarchical reference locality.**

We propose *LaPerm*, a new locality-aware TB scheduler that exploits such parent-child locality, both spatial and temporal. LaPerm adopts three different scheduling decisions to i) prioritize the execution of the child TBs, ii) bind them to the stream multiprocessors (SMXs) occupied by their parents TBs, and iii) maintain workload balance across compute units. Experiments with a set of irregular CUDA applications executed on a cycle-level simulator employing dynamic parallelism demonstrate that LaPerm is able to achieve an average of 27% performance improvement over the baseline round-robin TB scheduler commonly used in modern GPUs.

***Keywords*-GPU; dynamic parallelism; irregular applications; thread block scheduler; memory locality**

I. Introduction

General purpose graphics processing units (GPUs) have emerged as an effective vehicle for accelerating a diverse range of applications - especially traditional scientific and engineering computations [1][2][3][4]. These GPU architectures execute bulk synchronous parallel (BSP) programs organized as 1D-3D arrays of *thread blocks* (TBs) in the 32-thread unit of warps which can effectively map the applications with structured control and data flows with large data sets. However, emerging data intensive applications in data & graph analytics, retail forecasting, and recommendation systems to name a few, are dominated by algorithms with a time-varying, nested, parallel structure that exhibits irregular memory and control flows challenging the ability to effectively harness the performance of these GPUs. Responding in part to this need, new programming models have emerged to capture time-varying nested parallelism characterized by these applications - referred to as *dynamic parallelism*. The CUDA Dynamic Parallelism (CDP) [5] model extends the base CUDA programming model with device-side nested kernel launch capabilities to enable programmers to exploit this dynamic evolution of parallelism in applications. OpenCL provides similar capabilities with the OpenCL device-side kernel enqueue [6]. Such dynamic parallelism in BSP applications presents a very different profile of spatial and temporal memory reference locality than that presented in traditional CUDA and OpenCL applications. This paper proposes, describes, and evaluates a TB scheduler for GPUs designed to exploit reference locality in GPU applications that exploit dynamic parallelism.

Considerable attention has been focused on the impact of reference locality in GPU applications and the development of performance enhancement techniques that exploit this reference locality in warp scheduling techniques, cache management, and resource management [7], [8], [9], [10], [11], [12], [13], [14]. While many of the insights that motivate these works are applicable, none of them address the domain of dynamic parallelism in GPUs. Dynamic parallelism involves device-side nested launches of kernels or TBs (equivalently workgroups in OpenCL). Consequently it introduces new types of locality behaviors, for example, spatial and temporal reference locality between parent kernels and child kernels, or between child kernels launched from the same parent kernel thread (sibling kernels). Modern GPU microarchitecture schedulers are designed for non-dynamic parallelism settings and are unaware of this new type of locality relationships. Existing locality-aware TB schedulers do not work across the kernel launching boundary and therefore only utilize the TB locality information within a single kernel, either parent or child.

We seeks to exploit the new data reference locality relationship in dynamic parallelism. Since dynamic parallelism occurs at the level of kernels or TBs, our work focuses on an effective TB scheduling policy to improve memory behavior by exploiting data reference locality between parent and child TBs and between sibling TBs. The choice of *TB* granularity is also a result of *fine-grained* dynamic parallelism that exists in irregular applications. Towards this end we first provide an analysis of parent-child and child-sibling reference locality in a set of benchmark applications. This analysis motivates a scheduler for TBs across the Stream Multiprocessors (SMXs). Consequently prioritization schemes are proposed to ensure that child kernels can exploit temporal locality with parent kernels, and spatial and temporal locality with sibling kernels. We also observe a tradeoff between parallelism and locality - improved utilization of all SMXs can be at expense of

1063-6897/16 $31.00 © 2016 IEEE
DOI 10.1109/ISCA.2016.57

reduced locality in individual SMX L1 caches. The proposed TB scheduler seeks to balance overall SMX utilization with effective utilization of the local SMX L1 caches resulting in overall improved system performance. Our proposal is designed to be orthogonal to warp scheduling techniques and can be combined with existing warp scheduling disciplines. On average the proposed approach achieves 27% IPC improvement over the baseline TB scheduler.

II. Background

This section provides a brief introduction to the GPU execution model and the baseline GPU architecture. These details, especially the dynamic parallelism execution model and the TB scheduling strategy adopted by the current baseline GPU architecture, are both relevant and important to the understanding of the proposed TB scheduler.

A. GPU Execution Model

Modern GPUs employ the BSP execution model where the program to be executed on the GPU is expressed as a kernel function with 1D-3D arrays of TBs. Each TB is further specified as 1D-3D arrays of threads, all executing the same kernel code subject to user-defined synchronization barriers. Multiple memory spaces can be accessed by the GPU threads during their execution, including the global memory which is visible to all the threads of a kernel, the shared memory which is visible only to the threads of a TB, and the local memory that is private to each individual thread.

B. GPU Architecture and TB Scheduling

In this paper, the baseline GPU architecture is modeled after the NVIDIA Kepler K20c GPU with the GK110 architecture. Without loss of generality, the proposed ideas along with the experiments and analysis also apply to other general purpose GPU architectures designed to implement the BSP execution model. Figure 1 illustrates the architecture of the baseline GPU that is composed of several major functional units: the kernel management unit (KMU), the kernel distributor unit (KDU), the computation units referred as Stream Multiprocessors (SMXs) and the SMX scheduler which dispatches TBs to the SMXs. Each SMX features multiple computation cores (SIMD unit), special function units, register files and on-chip scratch memory that can be configured as an L1 cache or shared memory. An L2 cache is shared across SMXs and connects through one or more memory controllers to the off-chip DRAM which is used as the global memory.

Kernels are launched from the host CPU to the GPU and are managed by KMU. The KMU selects independent kernels and dispatches them to fill the KDU (currently with a maximum of 32 entries). Kernels in the KDU are then executed in a first-come-first-serve (FCFS) order. The TBs of each kernel are dispatched to the SMXs and occupy necessary resources on the SMXs such as registers, shared memory, and texture memory in order to be executed. Each TB is divided into groups of 32-threads called warps which are executed on the SIMD units. If all SMXs are not fully occupied by TBs from one kernel, the SMX scheduler can dispatch TBs from the next kernel in the KDU to the SMX, which results in concurrent kernel execution. In today's GPU, kernels can be executed concurrently to the limit of 32 (number of entries in KDU).

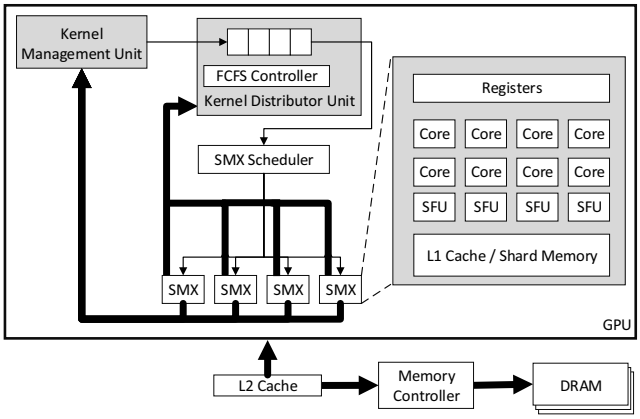


Fig. 1: Baseline GPU Architecture

After a kernel is dispatched to KDU, the SMX scheduler will schedule the TBs of this kernel to the SMX in a round-robin fashion. Each cycle it picks one TB using the increasing order of the TB ID and dispatches it to the next SMX that has enough available resources to execute this specific TB. When a kernel is launched, all SMXs are unoccupied and therefore can accommodate one TB at each cycle, resulting in TBs being evenly distributed across the SMXs. For example, scheduling 100 TBs on a 13-SMX K20 GPU would result in SMX0 being assigned TBs (0, 13, 26, ...), SMX1 being assigned TBs (1, 14, 27, ...) and so on. When all the SMXs are fully occupied by the TBs, the SMX scheduler is not able to dispatch new TBs to the SMXs until one of the older TBs finishes execution. To illustrate it with the same example, suppose each SMX will be fully occupied by 3 TBs. TB 39 cannot be dispatched to SMX0 immediately after TB 38 as there are not enough resources available. At some point, TB 17 on SMX4 becomes the first TB to finish execution so the SMX scheduler can schedule TB 39 to the SMX4 instead of SMX0. This TB scheduling strategy in the baseline GPU architecture is designed to ensure the fairness of occupancy across all the SMXs and thereby execution efficiency. It works well for most structured GPU applications.

To achieve better memory system performance on GPUs, kernels would generate more coalesced memory accesses where threads in each warp access consecutive memory addresses that results in a singe global memory transaction, utilize L2 and L1 caches to decrease memory access latency, and stage data in the shared memory for faster TB-wide data sharing.

C. Dynamic Parallelism on GPUs

Newer generations of GPUs have been designed to support device-side kernel launching functionality – CUDA Dynamic Parallelism (CDP) on NVIDIA GPUs or device-side enqueue

in OpenCL - which provides the capability of invoking kernels on demand from the GPU threads. Device-side kernel launches are useful and can be potentially beneficial when applied to dynamic parallelism in irregular applications on GPUs which occurs in a data dependent, nested, time-varying manner and the most straightforward implementations usually lead to poor workload balance, control flow divergence and memory irregularity. It has been demonstrated in [15] that there exist segments of computations within many irregular applications that locally exhibit structured control and memory access behaviors. The device launching capability extends the basic GPU programming model and enables programmers to exploit this dynamic evolution of local, structured parallelism in applications by generating new kernels on demand when such structured parallelism is recognized during execution.

While dynamic parallelism extensions such as CDP address the productivity and algorithmic issues, efficiently harnessing this functionality on GPUs has been difficult due to multiple factors such as low SMX utilization and high kernel launch latency. Meanwhile, related research [16][17][18][19] have been proposed using either new compiler or microarchitecture techniques to explore dynamic parallelism on GPUs. Specifically in [16], the authors propose the "Dynamic Thread Block Launch (DTBL)" extension to the existing GPU execution model that supports launching lightweight TBs in the form of TB groups from GPU threads on demand and coalescing these new TBs with existing kernels when the configurations of the new TBs match those of the native TBs of the existing kernels. With DTBL, the device kernels in the irregular applications that are originally implemented with CDP are replaced on-demand by light weight TB groups. This new extension increases the SMX execution efficiency, mitigates the overhead of CDP kernel launches while preserving the benefits of CDP in terms of control flow and memory behavior, and therefore is able to achieve both productivity and performance advantages for dynamic parallelism. However, that work does not consider the memory locality effects of hierarchical parallelism created by device side kernel or TB launches. Several works have established the importance of such memory locality effects and the need for TB and warp level scheduling techniques [9][10][12][13]. In this paper, we investigate and characterize the memory locality behavior of dynamic parallelism based on both the CDP and DTBL models and propose TB scheduling policies to exploit memory locality. We expect these techniques and insights also can be applied to other forms of dynamic parallelism that are bound to emerge in the future for BSP applications.

We refer to the TBs of a device kernel (CDP) or a TB group (DTBL) as *dynamic TBs*. TBs which launch new device kernels or TB groups are the *direct parent TB*. All the TBs that are in the same kernel or TB group as the direct parent TB are the *parent TBs*. The TBs in the newly launched device kernels or TB groups are the *child TBs*. Figure 4(a) shows an example of the parent-child launching using either the CDP or the DTBL model. In this example, there are eight parent TBs (P0-P7) in the parent kernel. TB P2 generates two child TBs (C0-C1) and is their direct parent. The direct parent of the four child TBs (C2-C5) is TB P4. The notations of parent and child TBs will be used in subsequent discussion and equally applicable to both CDP and DTBL models as well as potentially other dynamic parallelism models as long as they retain the TB-based BSP execution model.

The baseline GPU microarchitecture used in this paper reflects the launch paths for both CDP and DTBL. In Figure 1, each SMX is able to issue new kernel launches via the path from itself back to the KMU. Just as with host-launched kernels, device-launched kernels are dispatched from the KMU to the KDU, and then from KDU to SMXs. In addition, there is also a path from each SMX to the KDU where new TBs can be generated and coalesced to existing kernels in the KDU. The new TBs will be appended to the end of the TB pool of the existing kernel and dispatched to the SMXs after all the native TBs. In the baseline, dispatching dynamic TBs is no different than dispatching TBs from the host-launched kernels, i.e., TBs are distributed to the SMXs in a round-robin fashion subject to resource availability on the SMXs.

III. Memory Locality in Dynamic Parallelism

In this section, we examine the memory reference locality that exists between the parent and child TBs in the course of exploiting dynamic parallelism on a GPU. Parent-child locality provides an opportunity for optimizing performance that is not exploited by existing TB schedulers on current GPUs, and is the major motivation of the proposed LaPerm TB scheduler.

A. Spatial and Temporal Locality

Researches [20][21][22] have shown that while it is common to observe the existence of reference locality at certain time during the execution of irregular applications, it usually occurs in a way that is non-uniform, fine-grained, nested, and dynamic. In structured applications, (e.g., many scientific codes) inter-thread locality often leads to effective coalescing of memory references and consequent efficient use of memory bandwidth. In contrast, the non-uniform occurrences of locality in irregular applications makes it difficult to exploit the peak memory bandwidth. However it has been shown in [15] that the use of dynamic parallelism can convert intra-thread locality to uniform inter-thread locality which in turn can lead to increased coalescing of memory accesses and thereby effective use of memory bandwidth. For example, expanding the neighbors of a vertex in a graph problem is often done by a single thread leading to intra-thread locality across outgoing edges. With dynamic parallelism, a child TB can expand each vertex concurrently designated by the parent thread. Thus, intra-thread locality of the parent is converted to inter-thread locality of the child TB. The work in this paper focuses on the shared structures between parent and child which can lead to locality of references between the parent threads and the child threads.

We demonstrate the existence of such locality by examining the memory access patterns of the direct parent and child TBs in multiple benchmarks described in Table II. The examination

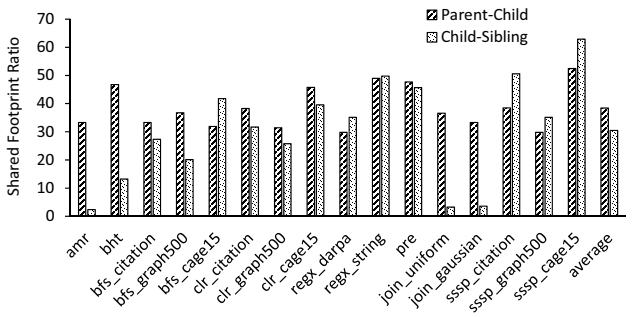


Fig. 2: Shared footprint ratio for parent-child and child-sibling TBs.

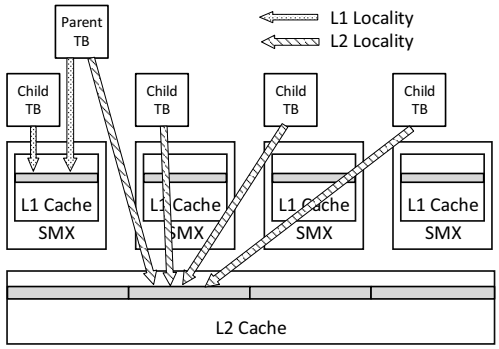


Fig. 3: Parent-Child Locality and Potential Impact on L1 and L2 Cache

process is performed between each direct parent TB and all of its child TBs, as well as between each child TB and all of its sibling TBs (the TBs that are launched by the same direct parent). The memory access patterns are application-dependent regardless of whether the CDP or the DTBL model is used. To quantify the memory access patterns and reveal the potential parent-child locality, we i) record the set of memory references that the direct parent and all of its child TBs make, and compute their respective sizes as p and c in units of a 128-byte cache block, ii) identify the memory references that are shared between the direct parent and all of its child TBs, and compute the total size as pc cache blocks, iii) compute the ratio pc/c as the *shared footprint ratio* for parent-child. Similarly, we record the memory references made by a single child TB and all of its sibling TBs respectively with size co and cs, identify the memory references shared by them with size cos and compute the ratio cos/cs as the *shared footprint ratio* for child-sibling. Figure 2 shows the results with an average shared footprint ratio of 38.4% for parent-child and 30.5% for child-sibling. It should be noted that data locality also exists among the parent TBs, but is significantly less than parent-child or child-sibling data reuse. Our analysis shows that the average shared footprint ratio for parent TBs is 9.3%. Therefore in this paper we focus on schedulers that can utilize the parent-child TB data reuse. Higher shared footprint ratio reveals better potential locality between the direct parent and the child TBs which may exist both spatially and temporally as described in the following:

Temporal Locality: A common practice in using the dynamic parallelism model for irregular applications is that the parent TB performs the necessary computation to generate the data, passes the data pointers (usually stored in the global memory) to the child and invokes the child to continue the computation. The reuse of the parent-generated data by the child TBs results in good temporal locality as long as the execution of the child TBs is "soon enough" after the parent. The parent-child shared footprint ratio shown in Figure 2 demonstrate the potential existence of such temporal locality.

Spatial Locality: Spatial locality may exist either between the direct parent TB and the child TBs or between different child TBs. This is usually because the computations of either the parent or the child can access memory locations that are relatively spatially close. For example, using a common data structure such as Compressed Sparse Row (CSR) for the graph problem where neighbor vertices are stored in consecutive addresses in the memory, different child TBs may explore subgraphs that are stored closely to each other in the memory. Compared with parent-child locality, the child-sibling locality can have higher variation as shown in Figure 2, depending on the benchmark characteristics or even the input data. For example, the input graphs *citation_network* and *cage15* exhibit more concentrated connectivity as vertices are more likely to connect to their (spatially) closer neighbors. Therefore, with the CSR data structure and its memory mapping, the child-sibling shared footprint ratio for the graph benchmarks that take these two input graphs are higher than *graph500* where vertices can connect to other vertices all over the graph, resulting in child TBs dealing with more distributed memory accesses. It is even more apparent in the benchmark *amr* and *join* that the child TBs are always working on its own memory region with virtually no data reference from other child TBs, causing the lowest shared footprint ratio among all the benchmarks.

While the intra TB locality of the child TBs with dynamic parallelism can result in more coalesced memory accesses that can leverage the global memory of the GPU memory hierarchy, the locality between the parent and child TBs provides an opportunity for improved memory performance in terms of L1 and L2 cache behavior as shown in Figure 3. L2 cache performance can be increased if locality exists among the TBs that are executed closer in time. Furthermore, execution of these TBs on the same SMX may even have a positive impact the L1 cache performance. However, exploiting such potential cache behaviors is by itself not straightforward and can largely depend on the GPU SMX scheduler.

B. Round-Robin TB Scheduler

The SMX scheduler on current GPUs adopts the round-robin (RR) TB scheduling policy which is designed for fairness and efficiency. This is the baseline scheduling policy we use. This policy works well for structured applications.

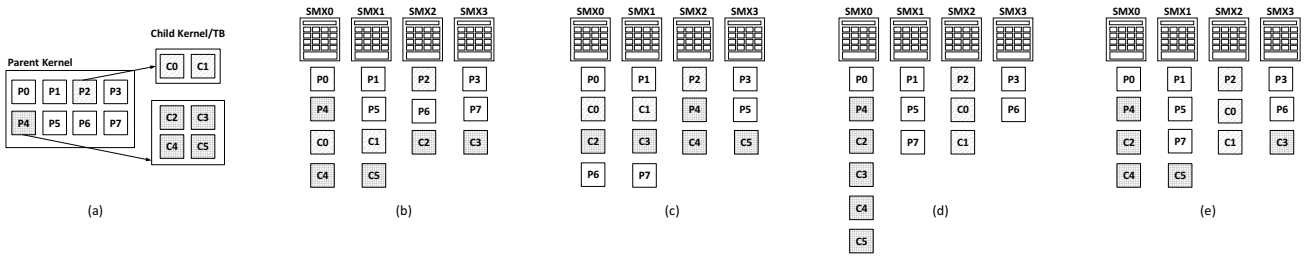


Fig. 4: An example of the parent-child kernel/TB launching (a), its TB scheduling results using the Round-Robin TB scheduler (b), TB Prioritizing (*TB-Pri*) (c), Prioritized SMX binding (*SMX-Bind*) (d) and Adaptive Prioritized SMX Binding (*Adaptive-Bind*) (e).

However, with the ability to dynamically launch kernels or TBs, this policy fails to exploit parent-child locality or child-sibling locality for dynamic TBs.

Figure 4(b) illustrates the effect of the RR policy for dispatching parent and child TBs to the SMXs for the example shown in Figure 4(a). The 8 parent TBs (P0-P7) and 6 child TBs (C0-C5) are executed on a GPU that has 4 SMXs (SMX0-SMX3). Each SMX is able to accommodate one TB. In the baseline GPU architecture, the KDU employs a FCFS kernel scheduler for all the parent and child kernels while the SMX scheduler only dispatches child TBs after the parent TBs. Therefore, the child TBs (C0-C5) will be scheduled after parent TBs (P0-P7). Furthermore, TBs are dispatched to SMXs in a round-robin fashion, so all the parent TBs and child TBs are distributed evenly across all the SMXs (assuming each parent TB and child TB is able to complete execution at the same pace) as shown in Figure 4(b). There are two major issues with the resulting TB distribution in terms of the impact on locality:

- Child TBs do not start execution soon after their direct parents. After TB P2 is executed, the SMXs are occupied by TBs (P4-P7) before P2's child TBs (C0-C1) can be dispatched. TBs (P4-P7) may pollute the L1/L2 cache and make it impossible for TBs (C0-C1) to reuse the data generated by TB P2 directly.
- Even if a child TB is scheduled soon enough after its direct parent TB, such as TB (C2-C3), they are not dispatched on the same SMX as its direct parent. Therefore, it is difficult to utilize the L1 cache of each SMX to exploit the parent-child or child-sibling locality.

The above two issues are exacerbated in real applications where the parent kernel generally is comprised of many TBs so that child TBs have to wait even longer before they can be dispatched and executed. The long wait may potentially destroy any opportunity to utilize the parent-child locality information.

Therefore, we propose *LaPerm*, a locality-aware TB scheduler which is specifically designed to improve locality behavior when employing dynamic parallelism on GPUs. As we will demonstrate, LaPerm leverages the spatial and temporal locality between and among parent and child TBs leading to better memory system performance, and therefore overall performance for irregular applications that employ dynamic parallelism.

IV. LaPerm Scheduler

In this section, we introduce the LaPerm TB scheduler which is comprised of three scheduling decisions: TB Prioritizing, Prioritized SMX Binding and Adaptive Prioritized SMX Binding. Each of the scheduling decisions differ in the specific forms of reference locality that they exploit and may showcase different performance benefits for applications with different characteristics as we will demonstrate in experiments and evaluations. LaPerm applies to the dynamically generated TBs both from device kernels in CDP as well as the TB groups in DTBL. We also propose architecture extensions to support LaPerm on GPUs.

A. TB Prioritizing

To address the issues with the RR TB scheduler for dynamic parallelism, we first propose a TB Prioritizing Scheduler (*TB-Pri*), where dynamic TBs are assigned a higher priority so that they can be dispatched to SMXs before the remaining TBs of the parent kernel or TB group. The parent TBs are given an initial priority and the launched child TBs are assigned a priority of one greater than that of the parent TBs. This priority assignment process can be nested to accommodate nested launches from the parent TBs to a maximum level L of the child TBs. Any nested launch level that exceeds L will be clamped to L. The goal of *TB-Pri* is to start the execution of dynamic TBs as soon as they are launched by the parents to facilitate the leverage of temporal locality.

Figure 4(c) shows an example of applying *TB-Pri* to Figure 4(a). For the purpose of illustration, we refer the process of scheduling four TBs to the four consecutive SMXs starting from SMX0 to SMX3 as *one round* of TB dispatching, one cycle for each TB on an SMX. Assume the parent TBs (P0-P7) are assigned with priority 0, then the child TBs (C0-C5) are assigned with priority 1. The first round of TB dispatching stays the same as the RR scheduler where TBs (P0-P3) are distributed to SMX0-SMX3. As the child TBs (C0-C1) are generated by TB P2 and assigned higher priority than TBs (P4-P7), they will be dispatched to the SMXs before (P4-P7) in the second round, resulting in C0 on SMX0, C1 on SMX1,

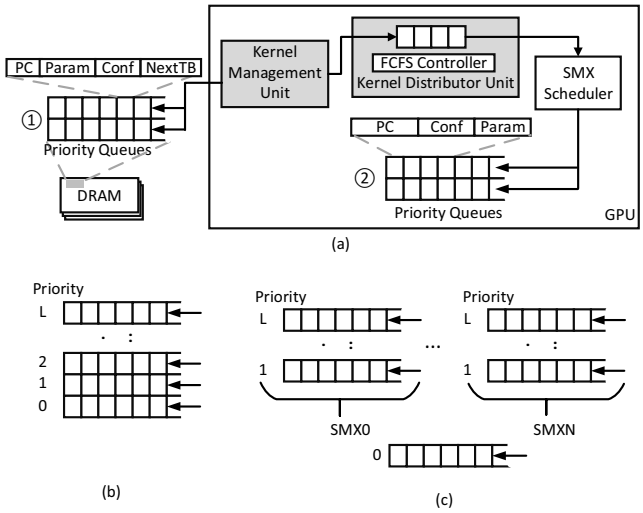


Fig. 5: Architecture Extension for LaPerm (a), the priority queues used by *TB-Pri* (b) and the SMX-bound priority queues used by *SMX-Bind* and *Adaptive-Bind*.

P4 on SMX2 and P5 on SMX3. Since TB P4 generates another four dynamic TBs (C2-C5), they will be scheduled on SMX0-SMX1 in the third round before the remaining two parent TBs P6 and P7 which are dispatched in the final round. Compared with the RR scheduler, child TBs (C0-C1) and (C4-C5) are scheduled earlier (in the second and third round instead of the third and the fourth round), which reduces the time gap from their direct parents and increases the possibility of better cache behavior because of the temporal locality. As child TBs can be scheduled to all the SMXs on the GPU, L2 cache performance increase can be the major benefit.

Architecture Support. To support *TB-Pri*, the kernel and TB scheduler are extended such that they can manage TBs with different priority values. The newer generation of NVIDIA GPUs support prioritized kernel launches [23], where kernels assigned with higher priority can be scheduled first and preempt the kernels with lower priority using the technique described in [24]. This is realized through multiple queues with different priority values, each of which contains the kernels with a specific priority value as shown in Figure 5(b). These priority queues are stored in the global memory and managed by KMU which dispatches kernels to KDU from the queues with higher priority followed by those with lower priority. Thus the SMX scheduler will also distribute TBs from kernels with higher priority to the SMX before those with lower priority. Preemption happens when higher-priority kernels are generated after lower-priority kernels start execution. In this case, when a TB from the lower-priority kernel finishes execution, the SMX scheduler will dispatch the waiting TBs from the higher-priority kernel to take up the freed capacity.

As shown in Figure 5(a), *TB-Pri* for both CDP and DTBL can also use the priority queues to manage the device kernels. Each entry of the priority queue contains information of the device kernel or TB groups including PC, parameter address, thread configuration and the next TB to be scheduled. The host-launched kernels stays in the lowest priority queue 0. In CDP, the priority queues are stored in the global memory①. The child kernels are assigned to the queue whose priority value is greater than its direct parent priority by one so that TBs from the child kernels are able to be dispatched before the remaining parent TBs. In the same priority queue, the newer kernels are appended to the tail so the priority queue itself is FCFS. The same priority queue structures are also used for DTBL to store the dynamic TB group information. As proposed in [16], DTBL uses both the on-chip SRAM and the global memory to store the dynamic TB group information when they are generated from the SMXs. These TB group tables are reused to store the priority queues (① for global memory and ② for on-chip SRAM) so that the on-chip SRAM ensures fast access to the TB group information from the SMX scheduler while the priority queues stored in the global memory serve as the overflow buffer.

Issues. Although *TB-Pri* leverages temporal locality and moves the child TB execution earlier, soon after the direct parent, the TB may be scheduled on any SMX. This can only increase the L2 cache performance. In the example Figure 4(c), TB (C0-C1) are still executed on different SMXs than its direct parent TB P2, therefore the L1 cache on SMX2 still cannot be utilized for parent-child data reuse. A similar observation applies to child TB (C2-C5) of the direct parent TB P4. TB C4 is now executed on SMX2 immediately after P4, which exhibits better locality than in Figure 4(b) and facilitates better L1 cache utilization, but the remaining child TBs (C2, C3, C5) are distributed across all the SMXs so that both the parent-child and child-sibling locality improvement is limited to the L2 cache behavior.

B. Prioritized SMX Binding

To utilize the entire GPU cache hierarchy more effectively, especially the L1 cache for data reuse, we extend *TB-Pri* so that the child TB should also be bound to the specific SMX that is used to execute its direct parent. This policy is referred to as Prioritized SMX Binding or *SMX-Bind*. The SMX binding directs the SMX scheduler to dispatch the child TBs such that they can use the same L1 cache on the SMX that is used by the direct parent.

Figure 4(d) shows the scheduling result using *SMX-Bind* for the parent-child launch structure in Figure 4(a). *SMX-Bind* identifies that child TBs (C0-C1) are launched by TB P2 from SMX2 so (C0-C1) are bound and dispatched on the same SMX2. Similarly, child TBs (C2-C5) are bound by their direct parent P4's executing SMX which is SMX4. The binding process ensures that C0 and C1 are scheduled in the second and third round to SMX2 and (C2-C5) are scheduled from the third to the sixth round to SMX0. All the remaining parent TBs are still dispatched using the original round-robin scheduling scheme. As the child TBs are now always scheduled on the same SMX as the direct parent, L1 cache can be well utilized to exploit the parent-child and child-sibling locality.

Architectural Support. The priority queues used for *TB-Pri* are extended to support *SMX-Bind* as shown in Figure 5(c) where the priority queues from 1 to L are used for each of the SMXs (SMX0-SMXN shown in Figure 5(c)). The priority queue 0 is shared by all the SMXs and reserved to store the information of the top-level parent kernels (host-launched kernels). A simple duplication of the original priority queues in Figure 5(a)) for the N SMXs would cost (N-1) times more hardware overhead, therefore, the extended architecture evenly divides the original priority queues in into N priority queues, each associated with one SMX with the expectation that TBs are evenly distributed across the SMXs. For each newly generated device kernel or TB group, the SMX scheduler will push its information to the priority queues that are associated with the SMX occupied by the direct parent. For each SMX, the TB dispatching process only fetches TBs from the associated priority queues until all the associated priority queues are empty so that new parent TBs can be fetched from priority queue 0. Note that in some GPUs, SMXs are divided into multiple clusters where each cluster possess more than one SMX and the L1 cache is shared by all the SMXs in a cluster [25]. In this case, *SMX-Bind* scheduling scheme associates the priority queues with the entire cluster and the newly generated TBs will be bound to any SMX in the cluster. Within each cluster, the round-robin dispatching strategy is employed for the TBs fetched from the priority queues.

Issues. In an ideal case, dynamic TBs can be evenly distributed across all the SMX to avoid any fairness issues and ensures that the evenly divided priority queues among SMXs are used in a balanced and efficient manner. However, as shown in Figure 4(d), it is possible that some parent TBs may have more nested launch levels or more child TBs (e.g. TB P4 and its four child TBs) than others. Restricting all these child TBs to a single SMX (or SMX cluster) may result in the idling of other SMXs and low overall execution efficiency. In Figure 4(d), SMX1 and SMX2 are idle after the third round of scheduling and SMX3 is idle after the second round, creating an unbalanced SMX workload. In irregular applications, it is very common that the launching patterns including nesting levels and child TB numbers vary from one parent TB to another, increasing the possibility that *SMX-Bind* could suffer from the SMX workload imbalance issue.

C. Adaptive Prioritized SMX Binding

To solve the load imbalance issues in *SMX-Bind* and increase the overall execution efficiency while preserving the cache performance benefits, we further optimize the *SMX-Bind* scheduling scheme to incorporate a more flexible TB dispatching strategy which is referred as Adaptive Prioritized SMX Binding (*Adaptive-Bind*). *Adaptive-Bind* still first dispatches prioritized child TBs to their bound SMX followed by other lower-priority parent TBs. At some point, both the prioritized child TBs bound to one SMX and all the parent TBs have been dispatched. *Adaptive-Bind* will then cross the SMX boundary and dispatch child TBs that are supposed to

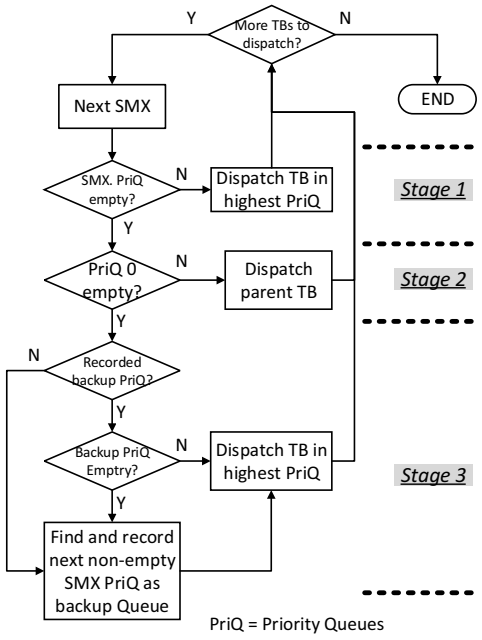


Fig. 6: LaPerm Scheduler Flow Chart

be bound to other SMXs to the current SMX if it has enough available resource to execute these child TBs. In this process, the dispatching scheme effectively put all child TBs bound to other SMXs as the backup TBs of the current SMX. The backup TBs can be viewed as TBs with the priority even lower than the top-level parent TBs which has priority 0. The goal here is to generate a more balanced TB distribution across all the SMXs to avoid any SMX idleness, which can result in balanced data reuse with increasing SMX utilization. This scheduling policy is balancing the tradeoff between exploiting reference locality within the cache hierarchy with utilization of the spatially distributed SMXs.

The scheduling results of *Adaptive-Bind* on Figure 4(a) is shown in Figure 4(e). Until the third round, the TB dispatching of *Adaptive-Bind* is the same as that of *SMX-Bind* as shown in Figure 4(d). The difference starts on SMX3 in the third round. As no child TBs are bound to SMX3 and all the parent TBs have been dispatched, *Adaptive-Bind* fetches the next TB from P4's child TBs – TB C3 which was originally bound to SMX0 – and executes it on SMX3. A similar procedure applies in the fourth round of TB scheduling on SMX1. The result shows that TB P2 and child TBs (C0-C1), and TB P4 and child TBs (C2, C4) are scheduled on the same SMX while the remaining child TBs are scheduled across all the SMXs. Compared with *SMX-Bind*, the performance of L1 may decrease due to less parent-child data reuse but it is compensated for by better SMX workload balance and thereby a potential positive impact on the overall execution efficiency.

Architectural Support. *Adaptive-Bind* still employs the same SMX-bound priority queues that are used by *SMX-Bind*. However, an extended SMX scheduler is designed to manage the priority queues and dispatch TBs. We show the complete

Adaptive-Bind TB scheduler operation flow in Figure 6 which is implemented as an extension to the SMX scheduler used by current GPUs. The LaPerm scheduler starts by following the normal routine of an SMX scheduler to check if there are more TBs from KDU to dispatch and execute. Then it checks all the SMXs, one for each cycle, and selects the candidate TB for the current SMX in three progressive stages: 1) highest-priority TB in the current SMX's priority queues, 2) parent TB in the global priority 0 queue and 3) highest-priority TB in the backup queues. Stage 2 only happens when the current SMX's priority queues are empty while stage 3 only happens when both the current SMX's priority queues and priority 0 queue are empty.

Note that in stage 3, when *Adaptive-Bind* selects priority queues of one SMX as the backup queues for the current SMX, it will focus on scheduling TBs from the chosen backup queues until they are also empty. As shown in Figure 6, the backup queues will be recorded each time they are selected and reused next time when stage 3 is invoked if they are not empty. The major reasons for this fixed backup scheme are that i) the TBs from the backup queues are also more likely to be scheduled on the same SMX which may help leverage their locality and ii) although the SMX scheduler is able to schedule TBs with different configurations on the same SMX, it may incur the overhead of resource initializing such as register and shared memory partitioning. Focusing on priority queues of one SMX can effectively minimize such overhead as each entry of a priority queue is either a kernel or a TB group that contains TBs using the same configuration.

For simplification, Figure 6 illustrates the LaPerm scheduler in DTBL model where TB groups are launched and pushed to the priority queues and directly scheduled by the SMX scheduler. In CDP model, new kernels are pushed to the SMX-bound priority queues stored in the global memory and dispatched by KMU to KDU and then to the on-chip SMX-bound priority queues used by the SMX scheduler. Therefore, The LaPerm scheduler also involves extension to the KMU kernel scheduler where it checks all the SMX-bound priority queues in a round-robin fashion (one SMX a time), dispatches the kernel with the highest priority if there is an available KDU entry and store its information in the corresponding priority queue of its bound SMX. The KDU entry number (currently 32 on GPUs that supports CDP) limits the dynamic kernels and thereby dynamic TBs that are available to be dispatched by the LaPerm SMX scheduler within a time frame. If the KDU is filled up with the 32 concurrent kernels, newly generated kernels cannot be dispatched from the KMU to the KDU even if they have higher priority. This is also a known limit in the kernel preemption context [24] where the kernels that are available to be preempted are limited to the ones that stay in KDU. In contrast, all the dynamic TBs in a DTBL model are coalesced to kernels in KDU so they are always visible to the LaPerm scheduler. As a result, TB dispatching with LaPerm on CDP may not always be able to find the highest-priority TB and achieve the optimized results in terms of locality and cache performance.

D. Impact of Launching Latency

LaPerm is built on the assumption that the child TBs can be executed early enough after the direct parent TBs to utilize the temporal locality and spatial locality. However, an important issue in the dynamic parallelism model is the launching latency of the child TBs especially in CDP [15], which can i) cause a long wait before the child TBs can actually be dispatched by the LaPerm scheduler, ii) introduce a lengthy time gap between the parent and child and iii) kill any potential parent-child locality. The DTBL model [16] along with any other future developments in dynamic parallelism models with better architectural, memory system, runtime, driver support may further reduce the launching latency and make full use of LaPerm. Section V analyzes the impact of launching latency on LaPerm scheduler performance.

E. Overhead Analysis

The major hardware overhead is caused by the priority queues used by LaPerm and the SMX scheduler extension shown in Figure 5. The SMX-bound priority queues that are stored in the global memory can have flexible size and be allocated during the runtime. They are indexed per SMX and per priority level. The on-chip SMX-bound priority queues are stored in a 3K bytes SRAM for each SMX (about 1% of the area cost by the register file and shared memory) and is able to store 128 entries (24 byte per entry). For an L-level priority queue, (L-1) index pointers are employed to separate these 128 entries to store TBs using the decreasing order of priority. The priority queue 0 shared by all the SMXs needs additional 768 bytes (32 24-byte entries) on-chip SRAM storage. Note that for CDP, the number of entries of the on-chip priority queues is limited to 32 per SMX the same as the KDU entry number.

The major timing overhead comes from pushing new dynamic TBs into priority queues and the LaPerm TB dispatching process. For CDP, generating new device kernels already incurs the overhead in storing the new kernel information in the global memory [23] which is the memory access latency. Pushing them to the priority queues stored in the global memory does not introduce additional overhead. Dispatching kernels by the KMU from the priority queues to the KDU may incur maximum extra L cycles where L is the maximum priority levels. The overhead is caused by the searching of the highest-priority kernel where in the worst case, all the L priority queues have to be searched, one cycle for each. For DTBL, inserting a new TB group to the on-chip priority queues introduces the searching overhead of the 128-entry queue to locate the insert position according to TB group's priority, which can be 128 cycles in the worst case. However, this searching overhead can be hidden by the setting up process of the TB groups such as allocating parameter buffer. If the on-chip priority queue is full and the new TB groups have to be stored in the overflow priority queues in the global memory, the overhead would be the global memory access latency which can also partly be hidden by the TB group setting up process. Finally, the dispatching process of LaPerm

TB is designed such that all the three stage searches can be finished within one cycle just as the baseline TB scheduler.

F. Discussion

The LaPerm scheduler is designed in a manner that is transparent to the warp scheduler, therefore it may be combined with any warp scheduler optimization such as [7][8]. Specifically, warp schedulers described in [10] also take into account the locality between different TBs and seek higher memory system utilization including bank-level parallelism, row buffer hit rate and cache hit rate by using a TB-aware warp grouping and prioritizing approach. Such warp schedulers can be leveraged by LaPerm to achieve even better memory system performance.

While many TB scheduling strategies are designed for the regular BSP model and may not apply by their own under the dynamic parallelism model, they can be certainly implemented as an optimization to LaPerm. For example, the TB scheduler introduced in [12] can dynamically adjust the dispatching TB number on each SMX to avoid too much memory contention. In LaPerm, the relatively small L1 cache (maximum 48 KB) may result in not fitting enough reusable data of the parent and child TBs, which can benefit from the incorporation of such contention-based TB control strategies.

This paper does not consider different data reuse patterns across child TBs, the impact of data reuse distance between the parent and the child TBs as well as that of the different hardware parameters such as cache size, thereby any scheduling optimization accordingly which could be implemented with commensurate runtime and hardware support. The LaPerm scheduler is a first step in scheduling approaches based on understanding data-reuse in dynamic parallelism that provides insights to help address these problems.

V. Experiments

A. Methodology

To evaluate and analyze the impact of the LaPerm scheduler on the memory system and the overall performance, We implement it on the cycle-level GPGPU-Sim simulator [26]. We first configure GPGPU-Sim to model our baseline architecture shown in Table I, which supports device kernel launches in CDP and dynamic TB group launches in DTBL using the same implementation described in [16] with the baseline RR TB scheduler. The configuration is designed to be compatible with CUDA compute capability 3.5. Also we adopt the same methodology used by [16] to simulate the launching latency of device kernels in CDP. The launching latency of the TB groups in DTBL is modeled directly in the simulator.

The benchmark applications used to evaluate the LaPerm scheduler are shown in Table II. They are irregular data intensive CUDA applications implemented both with CDP and DTBL as described in [15] and [16] using different input data set that features various characteristics. The implementations launch a new device kernel or a new TB group for any recognized dynamic parallelism in the applications. We trace all the parent kernels, child device kernels and dynamic TBs in all the benchmark applications to generate the performance results. All the applications are run entirely on the simulator including all the CUDA runtime APIs except for *regx* which we manually populate the memory data into GPGPU-Sim and run only the computation kernels to avoid extremely long simulation. The reported results include the overhead from both CDP/DTBL as well as the proposed LaPerm scheduler.

TABLE I: GPGPU-Sim Configuration Parameters

Clock Freq.	SMX: 706MHz, Memory: 2600MHz
Resources	13 SMXs, per SMX: 2048 Threads, 16 TBs, 65536 Registers, 32KB shared memory
L1 Cache	32 KB
L2 Cache	1536 KB
Cache line size	128 Bytes
Max # of Concurrent Kernels	32
Warp Scheduler	Greedy-Then-Oldest [7]

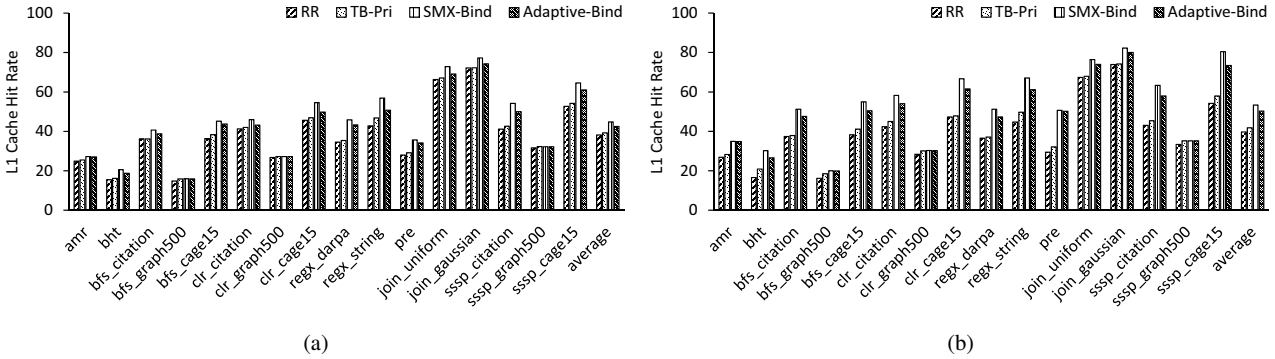
TABLE II: Benchmarks used in the experimental evaluation.

Application	**Input Data Set**
Adaptive Mesh Refinement (AMR)	Combustion Simulation[27]
Barnes Hut Tree (BHT) [28]	Random Data Points
Breadth-First Search (BFS) [29]	Citation Network[30] Graph 500 Logn20[30] Cage15 Sparse Matrix [30]
Graph Coloring (CLR) [31]	Citation Network[30] Graph 500 Logn20[30] Cage15 Sparser Matrix [30]
Regular Expression Match (REGX) [32]	DARPA Network Packets [33] Random String Collection
Product Recommendation (PRE) [34]	Movie Lens [35]
Relational Join (JOIN) [36]	Uniform Distributed Data Gaussian Distributed Data
Single Source Shortest Path (SSSP) [37]	Citation Network[30] Graph 500 Logn20[30] Cage15 Sparser Matrix[30]

B. Result and Analysis

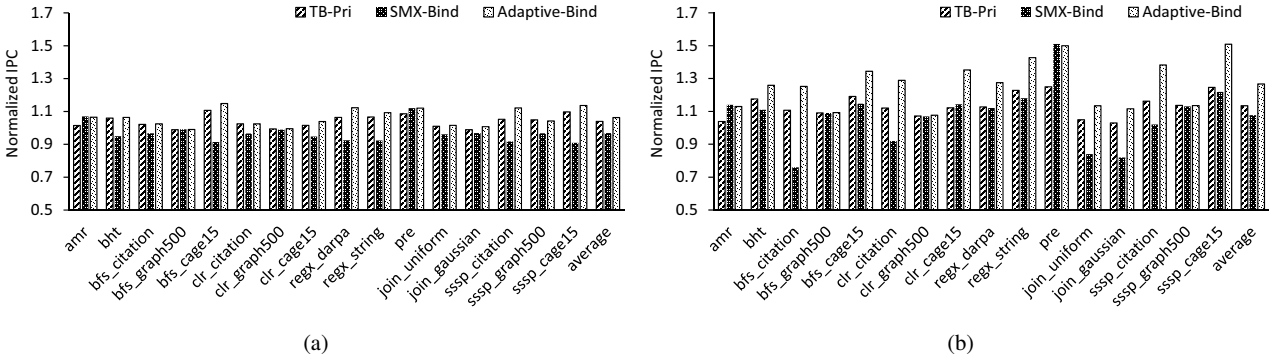
In this section we report the evaluation and analysis of the benchmark in various performance aspects. As the main focus of LaPerm is the memory system performance especially L1 and L2 cache, we use the cache hit rate as the metrics. We also analyze the impact of LaPerm on the IPC (instruction per cycle) metrics to evaluate the overall performance of the applications. All the evaluations are performed both for the CDP and DTBL model. Figure 7 and Figure 8 show the L2 and L1 cache hit rate respectively for the original CDP and DTBL using the RR TB scheduler as well as the three different schemes employed by LaPerm. Figure 9(a) and Figure 9(b) show the IPC normalized to the original IPC of CDP and DTBL implementations with RR scheduler respectively.

Performance of *TB-Pri*. As discussed in Section IV-A, the goal of *TB-Pri* is to increase the cache hit rate by prioritizing child TBs earlier after parent TBs. This is demonstrated by an average increase of 6.7% (CDP) and 8.7% (DTBL) for L2 cache hit rate and 1.1% (CDP) and 2.1% (DTBL) for L1 cache hit rate over RR scheduler. Together they also result in 4% and 13% normalized IPC increase for CDP and DTBL respectively.



(a) (b)

Fig. 8: L1 cache hit rate when applying LaPerm to (a) CDP and (b) DTBL.



(a) (b)

Fig. 9: Normalized IPC when applying LaPerm to (a) CDP and (b) DTBL.

Some of the benchmarks that achieve the highest L2 cache hit rate are *pre* and all the graph applications (*bfs*, *clr*, *sssp*) with the *cage15* input. These benchmarks generally have more dynamic child TB launching and higher parent-child shared footprint ratio as shown in Figure 2 and benefit more if the child TBs are able to reuse the data from the parent or the sibling TBs.

Performance of *SMX-Bind*. Although *TB-Pri* does not target L1 cache performance, we still observe a slight increase in the L1 cache hit rate. This is because child TB prioritizing can result in a few child TBs coincidentally being dispatched to the same SMX as the direct parent TB. This dispatching pattern is reinforced by *SMX-Bind* to achieve a L1 cache hit rate increase shown as 6.6% on average for CDP and 13.6% on average for DTBL.

The applications *pre* and *sssp_cage15* are again among the ones that achieve the highest L1 cache hit rate. In addition, *regx_string* also exhibits good L1 cache performance benefit. These applications have the characteristics that the workload performed by the child TBs focus on a relatively small memory region. For example, the production recommendation process of *pre* tends to search products that are highly related and thereby stored closer to each other in the memory. As a consequence, these applications can generate more closer memory accesses and higher child-sibling shared footprint ratio. When all these sibilant TBs are scheduled on the same SMX, they fundamentally increase the data reuse which result in substantial L1 cache hit rate increase.

In contrast, for the graph applications with *graph500* as input, *SMX-Bind* does not have any obvious L1 cache hit rate change from *TB-Pri*. Although these applications present some shared footprint ratio between the child TBs and their direct parent, the locality actually can also exist between any arbitrary non-direct parent TB and child TBs. The reason is that *graph500* is a graph with high and balanced connectivity that are evenly distributed across all the vertices. The data used by one parent TB exploring some of the vertices can be effectively reused by child TBs generated by a different parent exploring other vertices. The consequence of increased locality and cache performance from such data reuse patterns have already been captured by *TB-Pri*. Binding child TBs to specific SMX does not necessarily generate a higher L1 cache hit rate.

One major side effect of *SMX-Bind* is the SMX workload imbalance which may result in IPC decrease. Compared with *TB-Pri*, we observe the average normalized IPC decreases 9% for CDP and 5% for DTBL. For some of the DTBL applications (*bfs_citation*, *clr_citation*, *join*) and almost all of the CDP applications, normalized IPC even drops below 100% indicating performance loss from the baseline implementations with the original RR TB scheduler. Applications suffer from larger IPC loss generally have a more imbalanced child TB

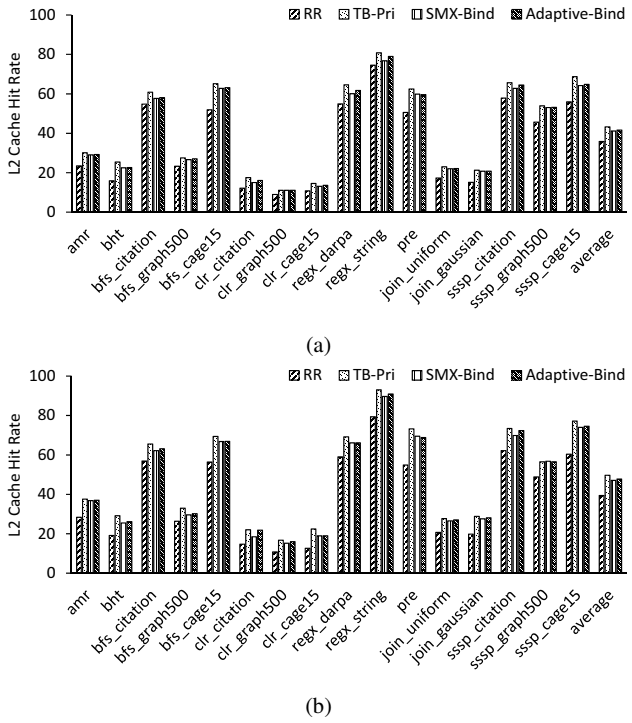


Fig. 7: L2 cache hit rate when applying LaPerm to (a) CDP and (b) DTBL.

launching patterns, i.e. some parent TBs may have substantially more child TBs and nested launching level than others, causing a long execution tail when these TBs are exclusively restricted to an SMX.

Performance of *Adaptive-Bind*. By using the adaptive SMX binding approach provided by *Adaptive-Bind*, we minimize the SMX workload imbalance side effect brought by *SMX-Bind*, which results in overall normalized increase of 6% for CDP and 27% for DTBL at the cost of some L1 cache hit rate decrease (by 2.3% for CDP and by 3.1% for DTBL compared with *SMX-Bind*). The study shows that IPC is impacted by L1 hit rate and load balancing – in fact IPC improvements due to the latter are greater than IPC reductions due to the drop in L1 rate. The results demonstrate that *Adaptive-Bind* effectively combines the benefits of prioritizing child TB execution, SMX binding and load-balance TB scheduling to achieve cache and overall performance gains for irregular applications that are implemented with the dynamic parallelism model. As a representative, application *sssp_cage15* achieves the highest IPC gain (11% for CDP and 51% for DTBL).

It is interesting to see some of the applications, such as *amr* and *pre*, have their normalized IPC increase from *SMX-Bind* and keep the value in *Adaptive-Bind* without any obvious further increase. The reason is that they have a more balanced kernel launching patterns among the some or all of the parent TBs. For example, *amr* has TBs in the grid centers to simulate the combustion patterns which all have similar temperature distribution, requiring similar refinement performed by the child TBs. Binding their child TBs to the SMX occupied by the direct parent will generate good L1 cache performance without causing many workload imbalance issues. Therefore, *SMX-Bind* would itself be a reasonable scheduling strategy for these applications to achieve IPC increase and does not require the SMX re-balancing process from *Adaptive-Bind*.

We also observe slight L2 cache hit rate changes compared with *TB-Pri* and *SMX-Bind*. In fact, increasing (decreasing) L1 cache hit rate may result in fewer (more) memory accesses falling into L2 cache, which could change the L2 cache behavior. According to our experiments, these changes do not affect L2 cache hit rate substantially and are not the major factors in affecting the overall performance.

C. Impact of Different Dynamic Parallelism Models

The microarchitecture and runtime differences of dynamic parallelism models such as CDP and DTBL can have impact on the effectiveness of the LaPerm scheduler. One of the major differences is the launching latency as described in Section IV-D. As we perform the evaluations for both CDP and DTBL, we observe that generally LaPerm in DTBL shows better cache performance and greater IPC increase (27% versus 6% in CDP) largely due to the fact that the high launching latency of the child kernels precludes LaPerm from timely dispatch to be executed closer to the parents in time. As for some applications such as *bfs*, parent TBs usually only have a small amount of work to do, long child launching latency leaves LaPerm no choice but only to schedule the remaining parent TB first before any child TBs arrive to fill the time gap.

Recall that CDP implementation today is subject to the 32 concurrent kernel limit in the KDU, reducing the number of child TBs that are available for LaPerm to schedule. As a result, the opportunity for LaPerm to perform an optimized TB prioritization using *TB-Pri* is dramatically reduced. The limit on the scheduling of TB candidates also causes poorer SMX imbalance for *SMX-Bind* as it is more likely to dispatch the available TBs to only a few SMXs but not other, which is the reason of the poor IPC of CDP that is even lower than using the original round-robin TB scheduler. As opposed to CDP, DTBL use dynamic TB coalescing to break the KDU limit and increase TB level concurrency, which makes LaPerm a more effective and efficient solution for the TB scheduler.

D. Insights

The experiments and results show that the three different scheduling decisions employed by LaPerm have various performance impacts on applications with different characteristics. Some of the insights include:

- *TB-Pri* uses child TB prioritizing to increase L2 cache performance and is specifically useful for applications where the locality is not restricted to the direct parent and its child TBs but also between multiple parents and their child TBs. Such locality facilitates the data reuse across different SMXs.

- For applications with more restricted locality between direct parent and child TBs, *SMX-Bind* is able to show the most obvious L1 cache performance improvement. On the other hand, the overall IPC may be optimized only when there are many parent-child launchings with similar workload to achieve SMX balance.
- There is a basic tradeoff between exploiting parent-child and child-sibling locality, and achieving higher SMX utilizations. For most irregular applications which show varying parent-child launching behavior across different parent TBs, *Adaptive-Bind* is the TB scheduler to achieve both the best cache performance and the balanced SMX workload which result in overall IPC increase.

VI. Related Works

There have been many recent works exploring dynamic parallelism on GPUs. Steffen et al. [38] use dynamically spawned warps to process subsections of the parent threads in supporting global rendering algorithm. Lars et al. [39] implement the NESL language on GPUs. Orr et al. [40] employ the channel abstraction which is a finite queue in CPU-GPU shared memory to process fine-grained dynamic tasks. Lee et al. [41] propose an auto-tuning framework that efficiently maps nested patterns in GPU applications. Kim et al. [19] implement a hardware work list to process dynamic generated parallel work elements.

With the prevalence of more complete dynamic parallelism execution models on GPUs such as CDP and OpenCL device-side enqueue, researchers have been proposing different extensions and optimizations. Yang et al. [18] propose a compiler transformation that can dynamically activate or deactivate the GPU threads to adapt to the evolving parallelism in the applications. Wang et al. [15] perform a systematic characterization and analysis on different aspects and performance impact of CUDA Dynamic Parallelism model on irregular application. To reduce the kernel launching latency associated with the CDP model, they also propose a new microarchitectural extension that is able to spawn light weight thread blocks on demand [16]. Chen et al. [17], on the other hand, propose a compiler technique "Free Launch" that reuses the parent threads to process the child kernel tasks, which is able to eliminate the runtime overhead of dynamic kernel launching.

As to the memory and cache performance on GPUs, people have been developing different approaches, including warp schedulers, thread block schedulers and new memory system designs. Rogers et al. [7] employs multiple warp-scheduler to achieve optimal cache performance. Narasiman et al. [9] propose a two-level warp scheduler to minimize the memory access latency. Jog et al. [10] advance the two-level warp scheduling technique to make it thread block aware so that the memory locality within and across thread blocks can be better accommodated. Kayıran et al. [12] demonstrate the memory contention caused by scheduling maximum possible number of thread blocks and use a dynamic thread block scheduling mechanism to minimize such contention. Lee et al. [13] introduce the idea of scheduling consecutive thread blocks on the same SMX to exploit inter-TB locality. Rhu et al. [21] design a locality-aware memory hierarchy that adapts to the fine-grained memory access patterns in irregular applications on GPU. Compared with these efforts, our solutions are based on the dynamic parallelism model that essentially seeks to optimize a new form of locality which exists between the parent and child thread blocks, and can be used in conjunction with state of the art warp schedulers to further exploit intra-warp and intra-thread block locality.

VII. Conclusion

In this paper, we propose a thread block scheduler, LaPerm, specifically designed for dynamic parallelism execution models on GPUs. The idea behind LaPerm is that the memory locality exists between the parent and child thread blocks that cannot be effectively exploit by existing round-robin TB scheduler on current GPUs. LaPerm employs three different scheduling decisions with new microarchitectural extensions to utilize such parent-child locality and improve the cache performance on GPUs. We evaluate LaPerm on a cycle-level GPU simulator with several CUDA irregular applications that are implemented with dynamic parallelism execution models, and demonstrate that by increasing both the L1 and L2 cache performance, LaPerm is able to achieve 27% IPC improvement over the original round-robin TB scheduler.

Acknowledgement

This research was supported by the National Science Foundation under grant CCF 1337177 and the equipment/technologies provided by NVIDIA. We would also like to acknowledge the detailed and constructive comments of the reviewers.

References

[1] J. A. Anderson, C. D. Lorenz, and A. Travesset, "General purpose molecular dynamics simulations fully implemented on graphics processing units," *Journal of Computational Physics*, vol. 227, no. 10, 2008.

[2] J. Mosegaard and T. S. Sørensen, "Real-time deformation of detailed geometry based on mappings to a less detailed physical simulation on the gpu," in *Proceedings of the 11th Eurographics Conference on Virtual Environments*, pp. 105–111, Eurographics Association, 2005.

[3] V. Podlozhnyuk, "Black-scholes option pricing," 2007.

[4] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, *et al.*, "Optix: a general purpose ray tracing engine," in *ACM Transactions on Graphics (TOG)*, vol. 29, p. 66, ACM, 2010.

[5] NVIDIA, "Cuda dynamic parallelism programming guide," 2015.

[6] Khronos, "The opencl specification version 2.0," 2014.

[7] 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-45)*, 2012.

[8] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Divergence-aware warp scheduling," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*, pp. 99–110, 2013.

[9] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving gpu performance via large warps and two-level warp scheduling," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, (MICRO-44)*, 2011.

[10] 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'13)*, 2013.

[11] P. Xiang, Y. Yang, and H. Zhou, "Warp-level divergence in gpus: Characterization, impact, and mitigation," in *Proceedings of 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA-20)*, 2014.

[12] O. Kayıran, 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'13)*, 2013.

[13] 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 *Proceedings of 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA-20)*, 2014.

[14] S.-Y. Lee, A. Arunkumar, and C.-J. Wu, "Cawa: coordinated warp scheduling and cache prioritization for critical warp acceleration of gpgpu workloads," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA-42)*, 2015.

[15] J. Wang and S. Yalamanchili, "Characterization and analysis of dynamic parallelism in unstructured gpu applications," in *Proceedings of 2014 IEEE International Symposium on Workload Characterization (IISWC'14)*, 2014.

[16] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili, "Dynamic thread block launch: A lightweight execution mechanism to support irregular applications on gpus," in *Proceedings of the 42nd Annual International Symposium on Computer Architecuture (ISCA-42)*, 2015.

[17] G. Chen and X. Shen, "Free launch: Optimizing gpu dynamic kernel launches through thread reuse," in *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-48)*, 2015.

[18] Y. Yang and H. Zhou, "Cuda-np: Realizing nested thread-level parallelism in gpgpu applications," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'14)*, 2014.

[19] J. Kim and C. Batten, "Accelerating irregular algorithms on gpgpus using fine-grain hardware worklists," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*, 2014.

[20] S. Beamer, K. Asanovic, and D. Patterson, "Locality exists in graph processing: Workload characterization on an ivy bridge server," in *Proceedings of the 2015 IEEE International Symposium on Workload Characterization (IISWC'15)*, 2015.

[21] M. Rhu, M. Sullivan, J. Leng, and M. Erez, "A locality-aware memory hierarchy for energy-efficient gpu architectures," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*, 2013.

[22] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads," in *Proceedings of 2010 IEEE International Symposium o nWorkload Characterization (IISWC'10)*, 2010.

[23] NVIDIA, "Cuda c programming guide," 2015.

[24] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on gpus," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA-41)*, 2014.

[25] NVIDIA, "Nvidia geforce gtx 980 whitepaper," 2014.

[26] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *Proceedings of 2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'09)*, 2009.

[27] A. Kuhl, "Thermodynamic states in explosion fields," in *14th International Symposium on Detonation, Coeur d'Alene Resort, ID, USA*, 2010.

[28] M. Burtscher and K. Pingali, "An efficient cuda implementation of the tree-based barnes hut n-body algorithm," *GPU computing Gems Emerald edition*, p. 75, 2011.

[29] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*, 2012.

[30] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, "10th dimacs implementation challenge: Graph partitioning and graph clustering," 2011.

[31] J. Cohen and P. Castonguay, "Efficient graph matching and coloring on the gpu," in *GPU Technology Conference*, 2012.

[32] L. Wang, S. Chen, Y. Tang, and J. Su, "Gregex: Gpu based high speed regular expression matching engine," in *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2011 Fifth International Conference on*, pp. 366–370, IEEE, 2011.

[33] J. McHugh, "Testing intrusion detection systems: a critique of the 1998 and 1999 darpa intrusion detection system evaluations as performed by lincoln laboratory," *ACM Transactions on Information and System Security*, vol. 3, no. 4, pp. 262–294, 2000.

[34] C. H. Nadungodage, Y. Xia, J. J. Lee, M. Lee, and C. S. Park, "Gpu accelerated item-based collaborative filtering for big-data applications," in *Proceedings of 2013 IEEE International Conference on Big Data*, 2013.

[35] J. L. Herlocker, J. A. Konstan, A. Borchers, and J. Riedl, "An algorithmic framework for performing collaborative filtering," in *Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 1999.

[36] G. Diamos, H. Wu, J. Wang, A. Lele, and S. Yalamanchili, "Relational algorithms for multi-bulk-synchronous processors," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles andPractice of Parallel Programming (PPoPP'13)*, 2013.

[37] M. Kulkarni, M. Burtscher, C. Casçaval, and K. Pingali, "Lonestar: A suite of parallel irregular programs," in *Proceedings of 2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'09)*, 2009.

[38] M. Steffen and J. Zambreno, "Improving simt efficiency of global rendering algorithms with architectural support for dynamic micro-kernels," in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-43)*, 2010.

[39] L. Bergstrom and J. Reppy, "Nested data-parallelism on the gpu," in *ACM SIGPLAN Notices*, vol. 47, pp. 247–258, ACM, 2012.

[40] M. S. Orr, B. M. Beckmann, S. K. Reinhardt, and D. A. Wood, "Fine-grain task aggregation and coordination on gpus," in *Proceedings of the 41st Annual International Symposium on Computer Architecuture (ISCA-41)*, 2014.

[41] H. Lee, K. Brown, A. Sujeeth, T. Rompf, and K. Olukotun, "Locality-aware mapping of nested parallel patterns on gpus," in *Proceedings of the 47th International Symposium on Microarchitecture (MICRO-47)*, 2014.