2017 IEEE International Symposium on High Performance Computer Architecture

Controlled Kernel Launch for Dynamic Parallelism in GPUs

Xulong Tang[1], Ashutosh Pattnaik[1], Huaipan Jiang[1], Onur Kayiran[2], Adwait Jog[3],
Sreepathi Pai[4], Mohamed Ibrahim[3], Mahmut T. Kandemir[1], Chita R. Das[1]
[1]*Pennsylvania State University* [2]*Advanced Micro Devices, Inc.*
[3]*College of William and Mary* [4] *University of Texas at Austin*
Email: {xzt102, ashutosh, hzj5142, kandemir, das}@cse.psu.edu, onur.kayiran@amd.com,
{adwait, maibrahim}@cs.wm.edu, sreepai@ices.utexas.edu

***Abstract*—Dynamic parallelism (DP) is a promising feature for GPUs, which allows on-demand spawning of kernels on the GPU without any CPU intervention. However, this feature has two major drawbacks. First, the launching of GPU kernels can incur significant performance penalties. Second, dynamically-generated kernels are not always able to efficiently utilize the GPU cores due to hardware-limits. To address these two concerns cohesively, we propose SPAWN, a runtime framework that controls the dynamically-generated kernels, thereby directly reducing the associated launch overheads and queuing latency. Moreover, it allows a better mix of dynamically-generated and original (parent) kernels for the scheduler to effectively hide the remaining overheads and improve the utilization of the GPU resources. Our results show that, across 13 benchmarks, SPAWN achieves 69% and 57% speedup over the flat (non-DP) implementation and baseline DP, respectively.**

I. Introduction

Graphics Processing Units (GPUs) are known to provide significantly high performance and energy efficiency for a variety of applications from different domains, such as medical science [32, 38], finance [25, 36], social media, graphics [39], and computer vision [30]. The CUDA and OpenCL programming models allow most of these applications to naturally map thread computations to regular data structures. Such structured and load-balanced mapping of the computational workload facilitates efficient harnessing of the available compute throughput and memory bandwidth in GPUs. However, such balanced mapping is not always possible, especially for many emerging data-intensive applications that work on irregular and unstructured inputs (e.g., graphs [20, 21, 22] and adaptive meshes [19]). Consequently, with continuously growing dataset sizes, it is becoming increasingly harder to effectively map such applications to GPUs and achieve high throughput with the desired energy efficiency [2, 6, 14].

Dynamic Parallelism (DP), supported by both CUDA [26] and OpenCL [4], is a promising feature that enables superior portability of irregular applications on GPUs. It provides applications with the flexibility to launch kernels at the device (GPU) side. In other words, if some threads are assigned higher computational workload than other threads, these threads (*parent threads*) can offload their workload by launching additional kernels (*child kernels*). Such dynamically-generated kernels can expose additional parallelism to GPU and potentially improve resource utilization [26]. However, there are two primary drawbacks of DP. First, launching of such child kernels is not free. Aggressively launching too many child kernels can incur significant performance penalties arising from the *launch overheads* [44]. Second, as each GPU core can only run a fixed number of Cooperative Thread-Arrays (CTAs[1]) [3] and each GPU can execute a maximum number of concurrent kernels due to the hardware-limits [27], cores can be severely underutilized in phases where only child kernels[2] are executing. This leads to an increase in *queuing latency* for the CTAs and kernels that cannot be scheduled due to the hardware-limits.

To address the above two drawbacks, we develop a new runtime framework, called *SPAWN*, underpinned by our observation that a better workload distribution (partitioning) between the parent and child kernels can minimize the *exposed* launch overheads and queuing latencies, while maintaining enough parallelism to improve performance. SPAWN mitigates the aforementioned issues by dynamically controlling the launch of child kernels depending on the state of the GPU. The framework estimates the amount of launch overhead and queuing latency based on the current GPU workload, and based on this, it makes judicious decisions regarding child kernel launches. If the framework decides not to launch child kernels for specific parent threads, the overhead of launching child kernels is significantly reduced. Also, as more computations are performed in the parent threads, the number of pending child kernels and CTAs reduces. Therefore, the queuing latency that is exposed substantially reduces as well. We make the following **contributions** in this paper:

- We conduct an in-depth characterization of DP applications and quantitatively study three parameters (factors) that affect the performance of dynamic parallelism. We demonstrate that the workload distribution (partitioning between parent and child kernels) is the most significant factor that affects the performance of a dynamic parallel application. We observe that by tuning the workload distribution statically, one can achieve performance improvements ranging from 4% to as much as 8.6×.
- We propose a novel *runtime framework*, called SPAWN, which dynamically tunes the workload distribution between the parent and the child kernels. SPAWN improves the

[1]A CTA is called as a "Workgroup" in OpenCL, and a "Thread-Block" in CUDA.

[2]Most of the child kernels launched are lightweight, and the CTAs associated with each child kernel can have very few warps.

2378-203X/17 $31.00 © 2017 IEEE
DOI 10.1109/HPCA.2017.14

applications' resource utilization and minimizes the launch overhead and queuing latency, and therefore, improves performance.

• Experimental evaluations show that SPAWN significantly improves the performance of the baseline dynamic parallel execution with an average speedup of 57% across 13 benchmarks. It is also able to perform within 6% of the performance achieved by the best offline workload distribution. SPAWN outperforms the flat (non-DP) implementations by 69% on average, making dynamic parallelism a viable option in GPUs.

II. Background

In this section, we provide a brief background on dynamic parallelism (DP) and critical factors that affect its behavior.

A. Irregular Applications and DP

To help understand the inefficiencies of irregular applications running on a GPU, let us consider Breadth-First-Search (BFS) as an example. Assuming that each thread represents a vertex, threads that traverse more edges (the vertices that have high number of neighboring vertices) require more computation. Figure 1 shows a snippet of BFS threads. Threads T1, T5 and T7 have few edges to traverse, while the threads T3 and T6 traverse more edges. In such a scenario, when threads T1, T5 and T7 finish, a lot of compute resources are left underutilized. Clearly, the overall performance is determined by threads T3 and T6. Many other irregular applications also suffer from this workload imbalance, causing performance loss when running on GPUs [6, 7, 12, 13].

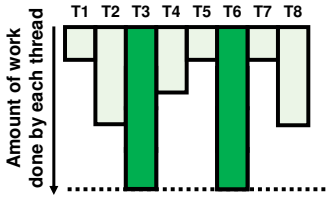
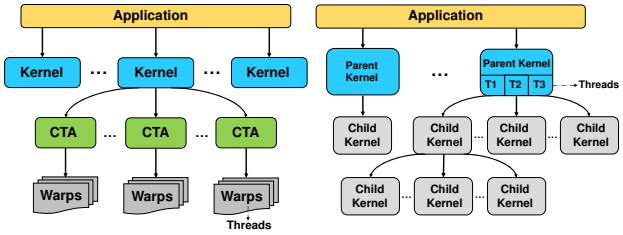


Figure 1: Illustrating workload imbalance in BFS.

Dynamic Parallelism (DP) is a mechanism supported by both CUDA [26] and OpenCL [4] that enables device-side kernel launches. Figure 2a shows the high-level structure of a conventional (non-DP) GPGPU application consisting of threads, CTAs, and kernels. A kernel contains multiple CTAs which can execute independently of each other. A CTA is a batch of threads which can communicate and synchronize with one another. The GPU hardware schedules threads into the pipeline in groups called "warps". As opposed to a conventional GPU application, a DP application can launch nested kernels from the device, as illustrated in Figure 2b. Each parent kernel can launch one or more child kernels. A child kernel itself can launch further child kernels and exhibit a nested launching pattern. Synchronizations are provided on device to guarantee the execution correctness. Through child kernel launches, a DP application can exploit more parallelism than its flat (non-DP) counterpart. This feature is particularly useful for irregular applications, where there can be large imbalances across the workloads assigned to different threads.



(a) Conventional application. (b) DP application.

Figure 2: High-level structures of conventional GPU applications and DP applications.

B. Properties of DP Applications

To trigger device kernels, a DP application is structured differently from a conventional GPU application. Figure 3 is an example code fragment extracted from BFS[3]. In this figure, (a) shows the code segment executing on the CPU (host), which is agnostic of any specific DP implementation. (b) shows the implementation of a parent kernel with the ability to launch device side kernels (child kernels), and (c) shows the application code for child kernels. For each child kernel, there are three unique parameters: *THRESHOLD*, (*c_grid*, *c_cta*), and *c_stream*, shown in red in Figure 3b.

(a)
```
1.  int main( int argc, char** argv){
2.      …
3.      dim3  p_grid; dim3 p_cta; /**parent kernel dimension*/
4.      parent<<< p_grid, p_cta>>>(type *workload); /**parent kernel launch*/
5.      …}
```

(b)
```
1.  __global__ void parent (type *workload){
2.      int pid = blockIdx.x*blockDim.x + threadIdx.x;
3.      type *local_workload = workload[pid]; /**each parent threads pick up its workload*/
4.      if (local_workload > THRESHOLD){
5.          dim3  c_grid; dim3 c_cta;
6.          cudaStream_t c_stream;
7.          cudaStreamCreateWithFlags(&s, cudaStreamNonBlocking);
8.          child<<< c_grid, c_cta, shmem, c_stream>>>(type *local_workload);}
9.      else
10.             while(local_workload){…}
11.         …
12.         cudaDeviceSynchronize(); /**waiting all children finishing*/
13.     }
```

(c)
```
1.  __global__ void child(*c_workload){
2.      int cid = blockIdx.x*blockDim.x + threadIdx.x;
3.      …
4.  }
```

Figure 3: Structure of BFS using DP. (a) Host code segment. (b) Parent kernel code segment. (c) Child kernel code segment.

***THRESHOLD*:** As explained previously, if a thread has a lot of edges to traverse in BFS, spawning a new kernel from that thread can increase parallelism. To achieve this, a *THRESHOLD* is set for a parent thread to decide whether to launch a child kernel or to traverse all the edges serially. For example, if the *THRESHOLD* is set to 128, threads

[3]Although the same approach is applicable to both OpenCL and CUDA, we show an implementation of BFS written in CUDA.

with more than 128 edges to traverse will launch a child kernel to perform the work. Other threads with less than 128 neighboring vertices will perform the traversal in loops (that is they will not create child kernels; instead, they will do the work by themselves in an iterative fashion). CUDA programming model allows applications to set any value as a *THRESHOLD*: a large value will result in a few heavyweight child kernels, whereas a small value will lead to a large number of lightweight child kernels. Clearly, setting a proper *THRESHOLD* value is a non-trivial task, as the value selected needs to reduce workload imbalance while avoiding significant overheads (Section II-C). Most DP applications [24, 41, 44, 45] make use of a small *THRESHOLD* value.

(*c_grid*, *c_cta*): Another important responsibility of the parent thread is to specify the dimensions of its child kernel. *c_grid* specifies the grid dimension in terms of the number of CTAs, and *c_cta* specifies the number of threads per CTA. *c_grid* and *c_cta* capture how the workload is parallelized in a child kernel.

***c_stream*:** The last important responsibility of the parent thread is to assign Software-managed Work Queue (SWQ) IDs to child kernels. These SWQs are called *c_stream* in CUDA programming. Child kernels with the same SWQ ID execute sequentially. In other words, all child kernels with the same SWQ ID execute sequentially but those with different SWQ IDs can potentially execute in parallel. An application creates a SWQ ID for each child kernel by initializing *c_stream* before launching a child kernel (lines 6 and 7 in Figure 3b). If the application does not specify *c_stream*, each parent CTA assigns the same SWQ ID to all its child kernels [28]. As a result, all the child kernels launched from the same parent CTA execute sequentially.

C. Hardware Architecture

The necessary architectural support for DP is shown in Figure 4. Similar to the traditional GPU applications (i.e., those without DP), a DP application starts running on the host (❶), and the parallel portion of the code is offloaded to the GPU through a runtime API (❷). A GPU kernel is tagged with a SWQ ID (❸), and pushed into Pending Kernel Pool located in Grid Management Unit (GMU) (❹). Kernels with the same SWQ ID are mapped into a single hardware work queue (HWQ). CTAs from a chosen HWQ's head-of-the-line kernel are dispatched to the GPU multiprocessor units (❺). The number of HWQs is 32 according to publicly-available documents from NVIDIA [27]. Therefore, the maximum number of kernels that can concurrently execute on the GPU is 32. Note that a CTA needs to wait in GMU if its required resources are not available or the hardware-limits are reached. The amount of time spent in GMU is called *queuing latency*.

Child kernels are launched through by invoking the related Runtime API function calls (❻). These API functions prepare the child kernel parameters and push the kernel

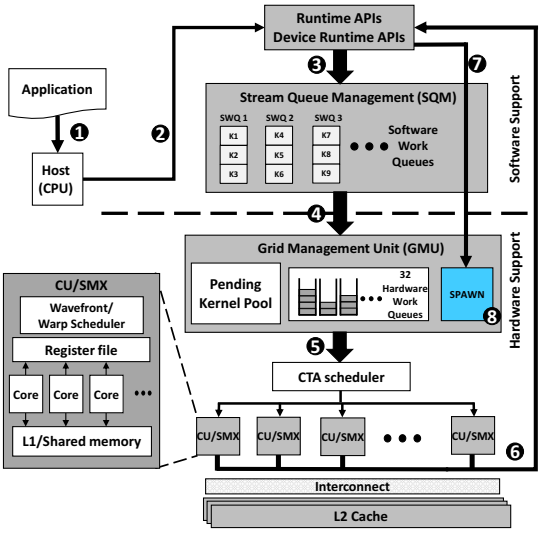


Figure 4: Hardware architecture realizing DP.

into Pending Kernel Pool in GMU. Note that these API calls are asynchronous [28], and allow the parent thread to continue its execution without waiting for the child kernel to be launched. The parent thread stops and waits for its child kernels to finish only when it finishes its execution or reaches an explicit synchronization point. If an entire parent CTA is waiting for synchronization, it relinquishes the occupied GPU resources so that other CTAs can be scheduled. It is important to emphasize that full memory consistency is only guaranteed at launching point and synchronization point; DP provides weak memory consistency between the launching point and synchronization point [28].

Launching a child kernel is not free, and entails performance overheads. The time spent on invoking the API (❻) and pushing the child kernel into Pending Kernel Pool (❸ + ❹) is called *launch overhead*. This launch overhead can potentially be hidden by overlapping the execution of other available warps on SMXs. However, in cases where a majority of running parent threads launch child kernels within a short period of time, such high number of API calls cannot be serviced simultaneously. As a result, the resulting launch overheads can degrade performance.

III. Application Characterization and Motivation

In this section, we first characterize all three parameters mentioned above using our benchmarks. We observe that *THRESHOLD* is the most significant contributor towards performance since it directly controls the workload distribution between the parent and child kernels. We next show how this workload distribution can affect: 1) the launch overheads and queuing latency, and 2) the GPU utilization.

A. Benchmarks, Metrics, and Observations

1) Benchmarks and Metrics: We use 8 applications and generate 13 benchmarks (each benchmark is an

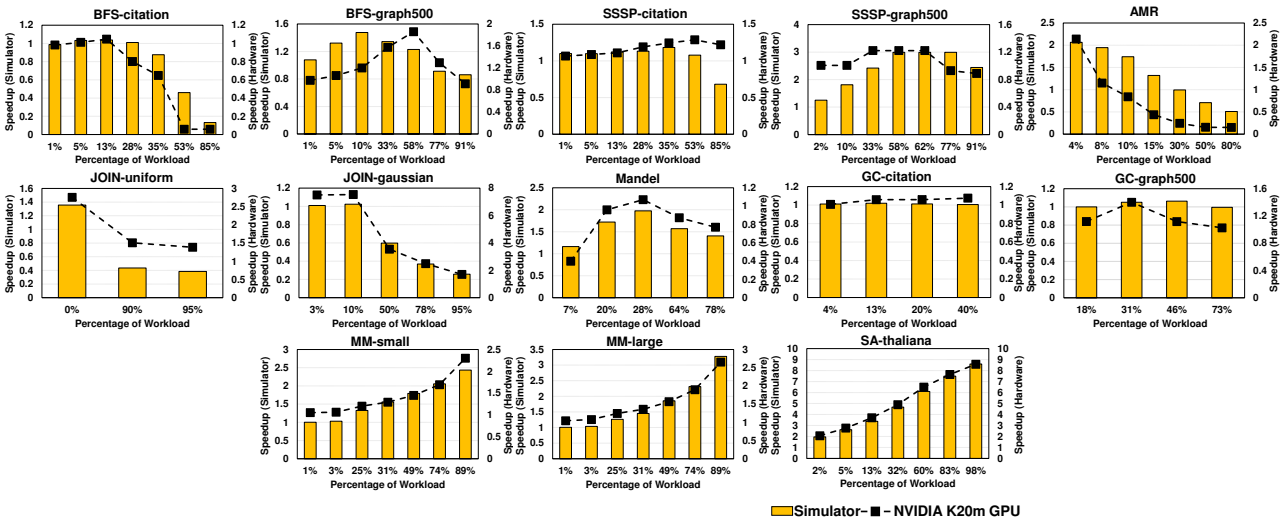


Figure 5: Effect of parent-child workload distribution on overall performance. We calculate the speedup in simulator (bars) and hardware (dashed curve) separately, by normalizing performance to the performance of running application's flat (non-DP) implementation on simulator and hardware, respectively. The x-axis shows the percentage of workload offloaded by launching child kernels.

<application, input> pair) by varying input sets of a few applications. The applications along with the benchmarks are listed in Table I. `MM` and `SA` are two applications written by our group. In `MM`, each parent thread multiplies one row (or couples of rows) of the multiplicand matrix with an entire multiplier matrix. In the DP version, a parent thread launches a child kernel and each thread of that child kernel picks up one column from the multiplier matrix to perform multiplication. In `SA`, all the reads [4] are divided into sections. Each parent thread handles one section of reads. For each read, there are several candidate locations in the reference index to match. The number of candidate locations varies among reads. In the DP version of this application, a thread launches a child kernel for a read if it has too many candidate locations. All the applications have a *flat* variant that does not use dynamic parallelism.

Table I: List of benchmarks.

Applications	Input Sets	Benchmarks
Adaptive Mesh Refinement [44]	Combustion Simulation [18]	AMR
Breadth-First Search [22, 44]	Citation Network [35]	BFS-citation
	Graph 500 [35]	BFS-graph500
Single Source Shortest Path [6, 44]	Citation Network [35]	SSSP-citation
	Graph 500 [35]	SSSP-graph500
Relational Join [10, 44]	Uniform Data	JOIN-uniform
	Gaussian Data	JOIN-gaussian
Graph Coloring [23]	Citation Network [35]	GC-citation
	Graph 500 [35]	GC-graph500
Mandelbrot Set	N/A	Mandel
Matrix Multiplication	Small sparse matrix	MM-small
	Large sparse matrix	MM-large
Sequence Alignment [9]	Arabidopsis Thaliana [1]	SA-thaliana

We measure performance using **speedup**, which is the ratio of the execution time of the flat (non-DP) implementation to the execution time of the DP implementation. We use *geometric mean* to represent the average speedup across all benchmarks. We also define **resource utilization** as the *maximum* of the register file utilization, shared memory utilization, and GPU compute unit (SMXs) utilization.

2) Observations: For our 13 benchmarks (Table I), we study the performance impact of varying the workload distribution ratio between the parent and child kernels. Each plot in Figure 5 represents one benchmark and the percentage numbers on x-axis represent the amount of workload offloaded to child kernels. Note that this analysis is *static (off-line)*, performed by changing *THRESHOLD* in the application code. It is important to emphasize that offloading 100 percent of a workload to child kernels is also possible. However, this would lead to intra-warp inefficiency because a very small workload might not use all the threads in a warp.

We show the results obtained from both the simulator and a real hardware in Figure 5. The yellow bars represent the performance results obtained using a modified version of GPGPU-Sim [5, 42], and the dashed lines represent the performance results obtained using NVIDIA Tesla K20m GPU. We use NVIDIA CUDA profiler [29] to profile the performance on hardware. The performance trends observed when using the simulator and the real hardware are similar. All the other observations and results provided in the rest of this paper are based on simulation results. From this analysis, one can make four major observations:

Observation 1: The preferred workload distribution ratio for each benchmark is different. Further, a given application (e.g., `BFS`) can have different preferred workload distribution ratios for *different inputs*.

Observation 2: Two of the benchmarks (`Join-uniform`

[4] A read is a substring of genome.

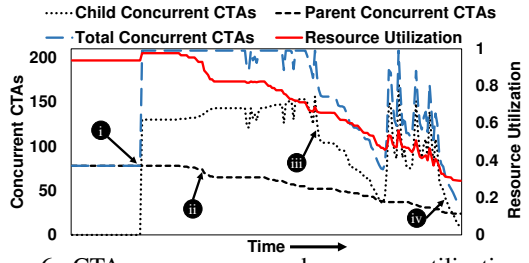


Figure 6: CTA concurrency and resource utilization over the course of execution of BFS-graph500 Baseline-DP. The maximum number of concurrently-running CTAs across all SMXs is 208. The total number of concurrently-running CTAs is the sum of the number of concurrent-executing child and parent CTAs.

and AMR) prefer processing the majority of work within the parent threads instead of launching child kernels. Join-uniform's input is regular, and the workload is balanced across all parent threads, leading to its preference of performing the workload within parent threads without launching child kernels. On the other hand, AMR launches nested child kernels and it is bottlenecked with the concurrent CTA limitation, and thus it also prefers to perform computations within the parent threads.

Observation 3: Three of the benchmarks MM-small, MM-large, and SA-thaliana prefer offloading a significant amount of workload to child kernels. In MM, both inputs are sparse matrices, resulting in severe workload imbalance among threads. Similarly, the number of candidate positions in SA varies among different reads, leading to workload imbalance among threads. Additionally, both MM and SA launch a small number of heavyweight child kernels, which means that the launch overheads have already been effectively hidden by the interleaved execution.

Observation 4: All the other benchmarks gain significant (8.6× in SA-thaliana) to modest (4% in Join-Gaussian) performance improvements by offloading parts of their computational workloads to child kernels, except GC-citation. In GC-citation, the number of child kernels is few ($<$ 2300 child kernels), and the amount of work in a parent is still significant to hide the launch overheads, leading to little variance between processing in the parent kernel and offloading to the child kernels.

To understand how a workload distribution impacts the GPU core utilization, consider Figure 6 which shows an execution snippet of BFS-graph500. The figure plots the number of concurrently-executing CTAs along with the resource utilization (as defined in Section III-A1). Initially, until cycle ❶, only the parent CTAs are executing. The child CTAs start their executions beyond that point, increasing resource utilization until the maximum concurrent CTAs is reached (between ❶ and ❸). Due to this hardware-imposed limit, even with enough available hardware resources, the GPU cannot run more CTAs. Starting from time ❸, the parent CTAs start to finish and relinquish resources, allowing more child CTAs to be scheduled. The resource utilization keeps decreasing because the child CTAs usually tend to be lightweight, not requiring as much hardware resources as the parent CTAs [44]. The number of concurrent child CTAs fluctuates between ❸ and ❹ because of two reasons. First, apart from the concurrent CTA limitation, there is a concurrent kernel limitation due to the limited number of HWQs. As a result, a large number of child kernels with a few CTAs per kernel will hit the concurrent kernel limit instead of the concurrent CTA limit, leading to a few concurrent child CTAs. Second, the trailing child kernels have long latencies before they can start executing, resulting in system idleness due to launch overheads. We show in Section IV how an intelligent workload balance can allow a better GPU core occupancy, thereby improving the overall GPU utilization.

(*c_grid*, *c_cta*): Figure 7 shows the performance variation with varying child CTA dimensions. The speedup is *normalized* to the CTA dimension with 32 threads. We observe from this plot that only certain applications such as AMR and SSSP-graph500 are sensitive to the CTA dimensions. AMR is bottlenecked by the hardware CTA concurrency limit under small CTA dimensions. Larger CTA dimensions prevent AMR from reaching this CTA concurrency limit. SSSP-graph500 prefers smaller child CTA dimensions, because the resource requirement for each of the child CTAs is high due to the unavailability of hardware threads. As a result, in SSSP-graph500, having smaller CTAs helps the CTA scheduler allocate more CTAs on SMXs, as the resource requirement is low compared to a larger-sized CTA.

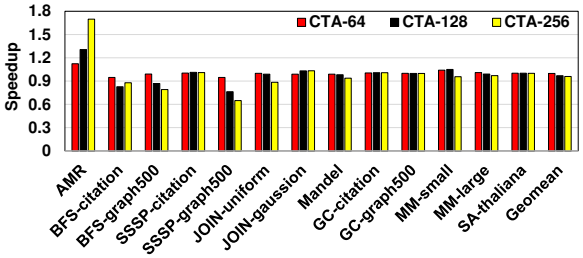


Figure 7: Performance sensitivity to different CTA sizes (64, 128, and 256 threads/CTA).

***c_stream*:** We also studied the impact of the number of SWQs on performance. As discussed in Section II-B, child kernels can be assigned with 1) a unique SWQ id for each child kernel, or 2) the same SWQ id for all child kernels being generated by a given parent CTA. The former enables more kernels to run concurrently, whereas the latter has fewer SWQs to manage. We compare these two mechanisms in Figure 8, and observe that assigning each child kernel a unique SWQ id always performs better. This is mainly because, in the second mechanism, a sequential execution of kernels limits concurrency. Therefore, we choose to assign each child kernel a unique SWQ id in all of the experiments presented in the rest of this paper.

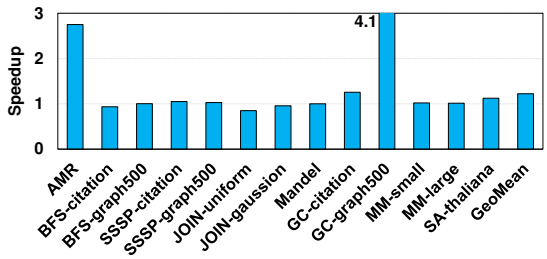


Figure 8: Performance of one SWQ per child kernel, normalized to performance of one SWQ per parent CTA.

In conclusion, our characterization shows that varying the workload distribution ratio (*THRESHOLD*) results in significant performance impact for our applications, while the other parameters do not affect most of the applications.

B. Potential Benefits of Parent-Child Workload Distribution

We now show the potential benefits of different workload distributions (partitioning) between the parent and child kernels with the help of an example. For the convenience of explanation, we assume there are 3 HWQs. In Figure 9, ① shows the execution time-line of the baseline DP scenario. At the very beginning, the parent kernel starts its execution, and there are multiple parent CTAs that are being executed concurrently. At some point during the execution, the local workload of some threads is found to be greater than *THRESHOLD*. These threads launch child kernels while the other threads proceed normally. As discussed in Section II-C, these child kernels need to wait for a period of time before they can start executing due to the *launch overhead* (Ⓐ). We further assume that each child kernel is associated with one unique SWQ id. However, since the number of HWQs is 3, there can be only 2 child kernels running concurrently along with the parent kernel. The remaining kernels have to wait and this results in increased queuing latencies. In ①, most of the parent threads launch child kernels, and consequently, the amount of computation performed by the parent kernel is less. As a result, most parent threads finish their executions faster and the GPU is under-utilized as child kernels are not able to start executing right away. There are two major shortcomings in this baseline DP execution. First, it *cannot* hide all the launch overheads. Second, due to the large number of child kernels in the queue and limited concurrency (number of HWQs) of the GPU hardware, the queuing latency of the child kernels can be quite high, leading to performance degradation.

Figure 9 ② shows a possible solution to mitigate these performance penalties. By limiting the workload offloaded to child kernels, first, the overall number of child kernels is reduced. This results in few and sparse child launching API calls and consequently reduces the launch overhead. In addition, more computation is performed within the parent threads. As a result, the parent thread execution is extended and can hide the launch overhead and queuing latency more effectively. A better workload balance, although not optimal, is achieved in ②. It saves us Ⓑ execution cycles.

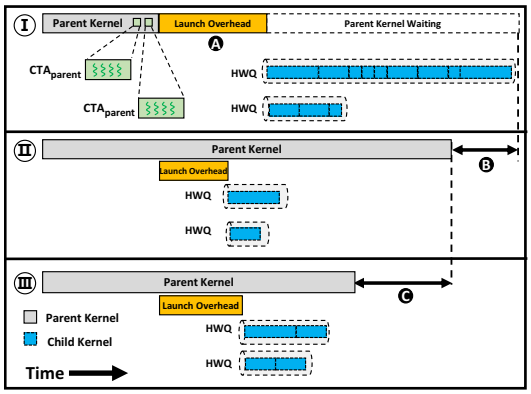


Figure 9: Time-line graph showing the benefits of balanced workload distribution between the parent and child kernels.

Obviously, in the best case scenario, the launch overhead is completely hidden while all necessary child kernels are launched to improve parallelism. Queuing latency also reduces since there are fewer pending kernels. ③ depicts such a case. Further execution time savings can be achieved by balancing the workload between the parent and child kernels if more concurrency is available. Such an approach takes full advantage of the available parallelism in a workload-balanced fashion, resulting in additional savings of Ⓒ cycles.

In summary, the workload distribution (partitioning) between the parent and child kernels is the most important parameter, and has a significant performance impact on DP applications. Since the preferred ratio varies among different applications (even with different inputs for the same application), setting a proper ratio is non-trivial and requires the knowledge of GPU runtime state. This, in turn, motivates the need for a *dynamic mechanism* that can control the workload distribution ratio between the parent and child kernels *on the fly* . To this end, we propose our runtime framework SPAWN.

IV. SPAWN: Dynamic Launch Control of Child Kernels

In this section, we describe our proposed approach to determine a balanced workload distribution between the parent and child kernels.

Overview: To achieve a balanced workload distribution between the parent and child kernels, we propose a runtime framework called SPAWN, oriented towards improving the GPU performance. The goal of SPAWN is to 1) improve GPU occupancy, 2) prevent the application from reaching the hardware-limits, and 3) dynamically control the performance trade-offs between increasing parallelism (launching child kernels) and incurring overheads.

Challenges: In order to effectively achieve a balanced workload distribution between the parent and child kernels, we should be able to estimate how beneficial it will be to launch a new child kernel, as opposed to performing the specified

computation within the parent thread. To better explain this, let us consider the example depicted in Figure 10, which shows the child kernel launches from three different parent threads (PT_i).

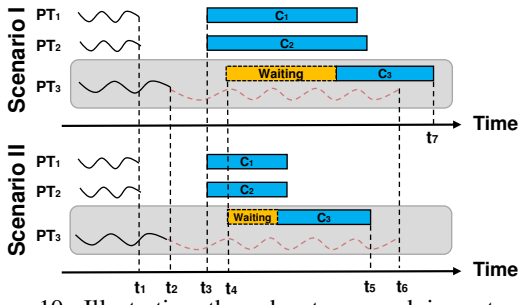


Figure 10: Illustrating the advantages and importance of knowing the runtime status while a parent thread is launching a child kernel.

At time t_1, two parent threads PT_1 and PT_2 launch their respective child kernels (C_1 and C_2), and these child kernels start their executions at time t_3. PT_3 makes a decision whether to launch C_3 or not at t_2. if C_3 is launched, it cannot start its execution immediately due to the launch overhead. Let us assume that C_3 is launched and can start its execution at time t_4. Based on the hardware requirements of C_1 and C_2 at time t_4, one can have two different scenarios. In **Scenario I**, C_1 and C_2 occupy most of the GPU resources for a long duration. In such a case, child kernel C_3 needs to wait for a long time for GPU resources to be freed up so that it can start its execution. Finally, C_3 finishes its execution at t_7. However, if PT_3 performs the computations itself without launching C_3 at time t_2, it finishes its execution at t_6, resulting in shorter execution time than the case where PT_3 launches C_3. On the other hand, as illustrated in **Scenario II**, C_1 and C_2 could be short running kernels and occupy resources for a short period of time. This would cause C_3 to start its execution earlier and thus, finish faster at t_5, where $t_5 < t_6$. Therefore, in this second scenario, launching a child kernel for PT_3 would be beneficial for improving performance.

A. The SPAWN Model

There are two major components of our SPAWN framework: Child CTA Queuing System (CCQS) and SPAWN Controller. As shown in Figure 11, CCQS monitors the launched child kernels and provides feedback information to the SPAWN controller enabling the latter to make a decision about child kernel launchings.

Child CTA Queuing System (CCQS): CCQS models the Grid Management Unit (GMU) as a "queue" and the SMXs as a server. The launch of child kernels generates CTAs, which act as "jobs" for CCQS[5]. As shown in Figure 11, the arrival rate of the jobs is denoted by λ. It conveys the spawning rate of CTAs from the new child kernels into the system. The throughput of CCQS is denoted by T. It conveys the rate of processing the child kernel CTAs on the GPU. Let n be the number of total jobs in CCQS, including both the running and pending child CTAs. Since CCQS works in a FCFS fashion, newly-launched child CTAs need to wait for the previous CTAs to be drained from CCQS and relinquish the occupied resources. Note that, if the child CTA arrival rate (λ) is greater than the throughput (T), CCQS accumulates more child CTAs, leading to long queuing latencies for newly-launched kernels.

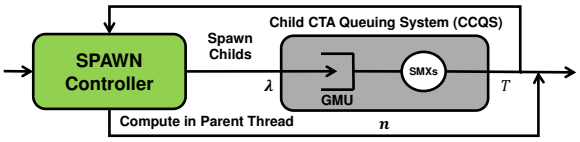


Figure 11: High-level view of SPAWN.

The SPAWN Controller: At each kernel launch call, SPAWN controller is invoked, and it is responsible for estimating the benefit of launching that child kernel, and making a decision on launching or not. For each child kernel, there are three time components involved: 1) launch overhead, 2) queuing latency, and 3) execution time. In our SPAWN framework, the launch overhead is modeled as the time to push child CTAs from SPAWN controller to CCQS. Note that we separate the launch overhead from CCQS, as CCQS tracks the child kernel CTAs only *after* they are pushed into GMU. Queuing latency is modeled in CCQS as queuing time, and is calculated by examining throughput (T) and the number of jobs (n) residing in CCQS. Execution time on cores is modeled as the service time in CCQS, and is calculated using throughput (T) and the number of CTAs (x) that the new kernel has. Therefore, we can approximate the time it takes for a new child kernel to finish its assigned workload using Equation 1,

$$t_{child} \approx Launch\ overhead\ + \frac{n}{T}\ + \frac{x}{T} \quad (1)$$

where:

T = $\frac{\text{Average Number of Concurrent CTAs}}{\text{Average Child CTA Execution Time}}$ and
x is the number of CTAs in the new kernel.

Similarly, Equation 2 estimates the time needed by the parent thread to perform the computations within itself rather than performing them in a child kernel. Generally, the parent thread will perform the computation in an iterative fashion. Each iteration time is approximately similar to the counterpart's child warp execution time.

$$t_{parent} \approx Workload \times t_{warp} \quad (2)$$

where:

t_{warp} is Average Child Warp Execution Time

By comparing the results of these two equations, our SPAWN controller chooses the option with the lower estimated execution time. Algorithm 1 gives the working of

[5]We use CTA granularity for our model because of two reasons: 1) each CTA execution is independent, and 2) CTAs cannot be preempted, or migrated to another core [28].

SPAWN in detail. Initially, it decides to launch child kernels because there is no CTAs in CCQS (line 2 to 3). Line 5 and line 6 represent Equations 1 and 2, respectively. Note that, there is a maximum queue size in CCQS, which we set to 65,536 in our implementation, based on the Kepler architecture [27].

Algorithm 1 SPAWN Controller

INPUT:

n : Total child CTAs in CCQS.
x : number of CTAs in new child kernel.
$workload$: Workload hold by parent thread.
$t_{overhead}$: Child launch overhead.
t_{cta} : Average child CTA execution time.
t_{warp} : Average child warp execution time.
n_{con} : Average number of concurrent CTAs.
t_{child} : Estimated child kernel execution time.
t_{parent} : Estimated parent thread execution time.

1: Initialization
2: **if** t_{cta} = 0 **then**
3: Spawn child kernel
4: **end if**
5: $t_{child} \leftarrow t_{overhead} + (x + n) \times t_{cta}/n_{con}$
6: $t_{parent} \leftarrow workload \times t_{warp}$
7: **if** $t_{child} \leqslant t_{parent}$ and $n + x \leqslant max\_queue\_size$ **then**
8: $n \leftarrow n + x$
9: Spawning Child Kernel
10: **else**
11: Process computation in parent thread
12: **end if**

Accuracy: SPAWN controller uses the historical average child CTA execution time to estimate the execution time of newly-launched child CTAs. In other words, SPAWN might make wrong decisions and lose opportunities if the execution time has a big variance among most child CTAs. However, this does not happen in most DP applications because: 1) all child CTAs share the same instructions and thus require similar hardware resources, and 2) child kernels are essentially lightweight and contain lightweight CTAs. Therefore, it is unlikely that the child CTA execution times significantly vary. In Figure 12, we show the PDF of child CTA execution time from four of our benchmarks. As one can see, 95% of the child CTAs (80% in `SSSP-graph500`) have their execution time within 10% of the average child CTA execution time. Because of this characteristic, even though SPAWN needs time to get t_{cta} converge to the average at the beginning of execution (within 5% of total execution), it can accurately estimate most child kernel execution times and make proper decisions for the remaining execution of the program.

B. Implementation Details

Figure 13 shows the high-level implementation of our SPAWN runtime framework. This implementation has two parts: 1) a source-to-source translator, and 2) an extension to the CUDA runtime that acts as a wrapper for the SPAWN controller function.

Source-to-Source Translator: Figure 14 shows the translated source code. First, the declaration of the kernel environment variables are moved outside the condition block, and the CUDA device launch function is used as the condition clause. The API function call returns with a flag of "success" when the child kernel is launched; otherwise, it returns with "fail" and the workload will be computed by the parent thread. Second, the child kernel launch needs to integrate the *local_workload* parameter into the CUDA runtime call to facilitate the estimation of the execution times in the SPAWN controller. This relieves the programmer from specifying any value of *THRESHOLD*.

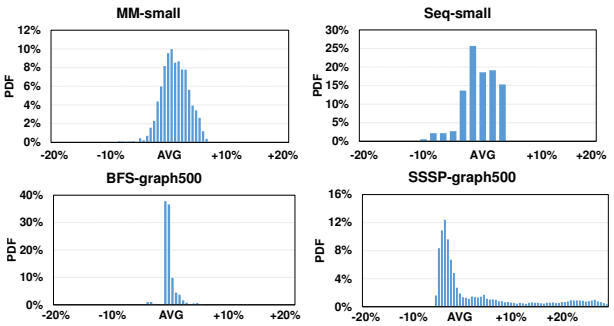


Figure 12: Child kernel CTA execution time. Each green point represents one child CTA. The red line shows the average execution of overall child CTAs.

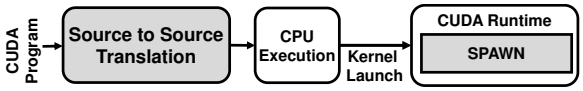


Figure 13: High-level view of SPAWN implementation.

```
1.  __global__ void parent (type *workload){
2.      type *local_workload = workload[pid]; //each parent threads pick up its workload
3.      dim3 c_grid; dim3 c_cta;
4.      cudaStream_t c_stream;
5.      cudaStreamCreateWithFlags(&s, cudaStreamNonBlocking);
6.      if (child<<< c_grid, c_cta, shmem, c_stream, local_workload>>>(type
*local_workload)){ … }
7.      else
8.          while(local_workload){…}
9.       …
10.     cudaDeviceSynchronize(); //waiting all children finishing.
11.  }
```

Figure 14: Translated version of the source given in Figure 3b.

CUDA Runtime Extension: We extend the CUDA Runtime, specifically the device kernel launch API call to integrate the SPAWN controller. At runtime, when the child kernel launch API is executed, SPAWN makes the decision regarding the launch of a child kernel by examining CCQS.

Monitored Metrics: We monitor the following metrics: 1) n, 2) t_{cta}, 3) n_{con} and 4) t_{warp}. As mentioned in Section IV-A, we need n and T to calculate the child kernel execution time. In order to compute T, we use two proxy metrics: i) t_{cta}, average child CTA execution time and ii) n_{con}, average number of concurrent child CTAs. Similarly, we monitor t_{warp}, average child warp execution time, to estimate the parent thread execution time. At the start of an application execution, all the metrics are initialized to 0. n is incremented/decremented in the SPAWN controller whenever a child CTA either enters or leaves CCQS. t_{cta} is updated only when a CTA finishes its execution and

leaves CCQS. We compute n_{con} over a window of 1024 cycles. At every cycle, we add the number of concurrently executing child CTAs to n_{con} and, at the end of the window, we bit-shift n_{con} by 10 bits to the right to obtain the average number of the concurrently-running child CTAs in the window. This average number is then used over the next window until a new value of n_{con} is calculated. Similarly, t_{warp} is also calculated in a windowed fashion.

Hardware Overheads: The main hardware overheads involve storing and updating the monitored metrics and computing the execution time. As shown in Figure 4, GMU is extended with the SPAWN logic (❽). It requires a 416 bytes table to keep track of each running child CTA's execution time[6]. It also requires one 16-bit register to hold n, two 16-bit adders and one shift register to calculate the estimated execution time. When a child CTA finishes its execution, it updates the related metrics located in GMU. Since the cores and GMU already communicate every cycle, this does not cause any extra communication overhead. The child kernel launch API communicates with SPAWN (❼) and returns the decision immediately, as the kernel launch API call is asynchronous.

V. Experimental Evaluation

A. Simulated System

We use a modified version of the cycle-accurate GPGPU-Sim v3.2.2 [5] that is able to simulate concurrent kernel execution and support dynamic parallelism. Table II provides the configuration details of the simulated system. The simulated system is modeled with 32 Hardware Work Queues (HWQs), therefore, limiting the maximum number of concurrently executing kernels to 32. In our simulation framework, we modify the GPU runtime to support SPAWN as described in Section IV-B.

Table II: GPU configuration parameters.

SMX	13 SMXs, 1400MHz, 5-Stage Pipeline
Resources per SMX	48KB Shared Memory, 64KB Register File, Max.2048 threads (64 warps, 32 threads/warp)
cache per SMX	16KB 4-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) [34] dual warp scheduler, Round-Robin (RR) CTA scheduler
Concurrency	16 CTAs/SMX, 32 HWQs across all SMXs
Interconnect	1 crossbar/direction (13 SMXs, 6 MCs) 1.4GHz, islip VC & Switch Allocators
DRAM Model	2 Memory Partition/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 warp [42]

B. Experimental Results

We study the effects of utilizing our SPAWN mechanism across 13 benchmarks (Table I). All the speedup results have been *normalized* to the execution of a flat (non-DP) variant of each benchmark. For each benchmark, we analyze the results for three different schemes: 1) the baseline dynamic parallelism execution (Baseline-DP), 2) the best workload distribution ratio[7] (Offline-Search), and 3) SPAWN. Figure 15 shows the speedups obtained when using three different schemes. Across the 13 benchmarks evaluated, we observe an average speedup of 69% and 57% compared to the flat variant and Baseline-DP execution, respectively. That is, although Baseline-DP performs better than flat version, our SPAWN significantly outperforms both flat and Baseline-DP. For the Offline-Search execution with the best workload distribution ratio, we obtain performance improvement of 61% over Baseline-DP execution.

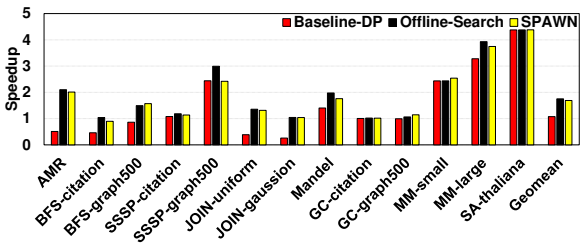


Figure 15: Speedup over the flat (non-DP) implementation.

We make three important observations based on these results. First, SPAWN is able to match the speedup obtained by Offline-Search, irrespective of whether the benchmark prefers launching child kernels or performing the computations within the parent thread. For example, `SA-large` has high performance when offloading a significant amount of work to the child kernels, whereas `AMR` prefers processing within the parent threads. SPAWN successfully captures the characteristics of these two dissimilar benchmarks. Note that, SPAWN is able to achieve within 4% of the Offline-Search's performance. Second, for three benchmarks, `BFS-graph500`, `GC-graph500`, `MM-small`, SPAWN performs better than Offline-Search. The slight performance improvement in SPAWN is due to Offline-Search being agnostic to the GPU hardware state. SPAWN is able to dynamically tune the workload distribution over the course of execution, taking into consideration the current state of the GPU, and also, it is able to control workload distribution decisions on a per kernel basis rather than using a statically fixed THRESHOLD value. Third, SPAWN under-performs for `SSSP-graph500` compared to Offline-Search and is similar to performance of Baseline-DP. This is because, for the monitored metrics to be useful to SPAWN, some child CTAs need to finish for the metrics to be updated to an accurate value. However, in `SSSP-graph500`, by the time the first child kernel finishes and updates the metrics, SPAWN had already made incorrect decisions and launched all the child kernels at this phase of execution.

Figure 16 shows the achieved occupancy across all SMXs.

[6]The table includes 208 entries, and each entry is a 16-bit cycle counter.

[7]We pick the best workload distribution ratio by performing an exhaustive sweep of the THRESHOLD metric, as mentioned in Section III-A2.

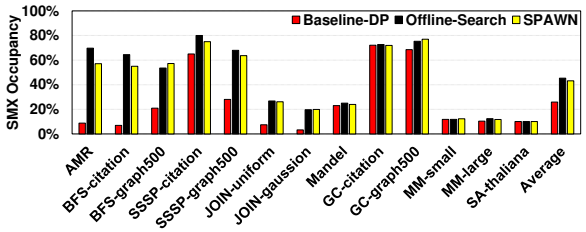


Figure 16: SMX occupancy.

SMX occupancy is defined as the ratio of the average active warps per active cycle to the maximum number of warps supported on all SMXs. A higher SMX occupancy could potentially improve the GPU performance and provide more latency tolerance towards child kernel launch overheads and queuing latency as seen by correlating Figure 15 and Figure 16. SPAWN achieves, on average, 1.96× higher SMX occupancy over Baseline-DP, and is within 4% of the SMX occupancy achieved by Offline-Search.

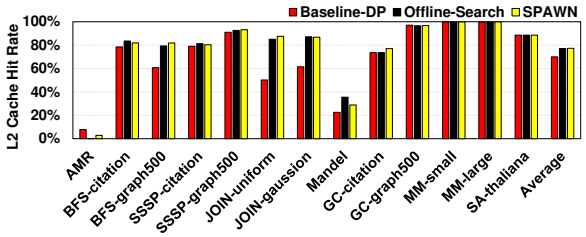


Figure 17: L2 cache hit rate.

We next evaluate the impact of SPAWN on cache performance. Figure 17 shows the L2 cache hit rate for the three evaluated schemes. Although SPAWN does not take data reuse and data access pattern into account, the L2 hit rate increases by around 10% compared to the Baseline-DP execution. This is mainly due to two reasons: 1) L2 cache contention is significant in Baseline-DP due to the high number of concurrently-executing child kernels, and 2) child kernels cannot execute *immediately* because of the launch overheads and queuing latency. This delay in execution of child kernels causes the loss in both temporal and spatial locality between the parent and child kernels [43]. SPAWN is able to increase locality by providing more computations to parent (improving spatial locality) and allowing the parent execution to last longer and overlap with the launched child kernels (improving temporal locality).

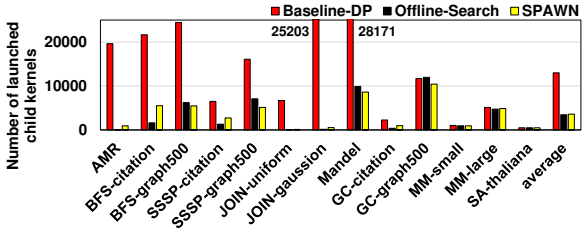
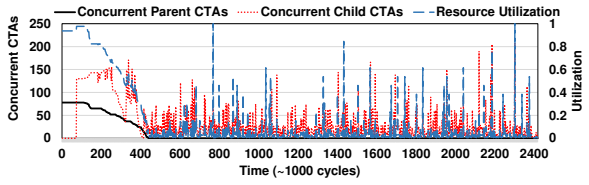


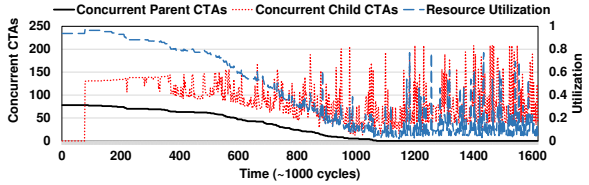
Figure 18: Number of child kernels launched.

Figure 18 shows the number child kernels that are launched during the benchmark's execution for the three different schemes. Note that the trend in the number of child kernels launched in Offline-Search execution and SPAWN are similar to each other. With SPAWN, the number of child kernels launched significantly reduces (by 73% on average). This reduction in the child kernel count helps in reducing the launch overhead and queuing latency. In the following subsection, we discuss the working of our SPAWN mechanism in detail.

C. Dynamic Workload Distribution



(a) Baseline-DP.



(b) SPAWN.

Figure 19: Concurrent CTAs of BFS-graph500 over time.

To better understand the working of our SPAWN mechanism, we describe the child kernel launch patterns for the Baseline-DP execution and our SPAWN mechanism. Figure 19a and Figure 19b show the number of concurrent CTAs in `BFS-graph500` scheduled on the SMXs at any given time during the course of execution for the Baseline-DP and SPAWN, respectively. Initially, only parent CTAs execute, following which the child CTAs start execution at 75k cycles. Since Baseline-DP of `BFS-graph500` gives significant work to child kernels, parent threads do not have much edges to traverse. As a result, they finish execution at 436k cycles, after which child kernels start dominating the SMXs' resources. However, there are two issues in the Baseline-DP. First, the child kernels cannot start execution immediately due to the launch overhead and queuing latency. Consequently, the concurrency and resource utilization dramatically drop. Second, many child kernels are launched in Baseline-DP, and they cannot execute concurrently because of the limited number of HWQs. Since each child kernel in `BFS` is lightweight (traversing only the neighboring nodes), the resource utilization is low during the phase when only child kernels execute (from cycle 436k to cycle 2,400k).

In SPAWN (Figure 19b), since more parent threads traverse the edges in a loop, the parent CTAs execute for longer duration and fewer child kernels are launched. As a result, the parent CTA execution is now able to hide the child kernels' launch overhead efficiently. In addition, as fewer child kernels are launched, it results in lower launch overhead and reduced queuing latency. Therefore, it leads

to higher resource utilization, and allows the application execution finish at 1600k cycles, unlike the Baseline-DP execution which takes 2400k cycles.

Figure 20 depicts the cumulative child kernel launch decisions that are taken over the entire execution for `BFS-graph500`. We see that SPAWN is dynamically able to make kernel launch decisions which are similar to the decisions taken by Offline-Search, and achieve similar workload distributions. From the figure, we see that Baseline-DP has considerable high child kernel launch rate compared to SPAWN. Since launch overhead and queuing latency dramatically increase when large number of child kernels are intensively launched[8], a reduction in child kernel launch rate effectively reduces the overheads and improves performance.

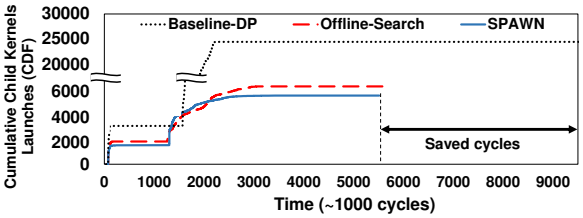


Figure 20: CDF of child kernels launched over time

D. Comparison with an Alternate Strategy

We are not aware of any runtime scheme that tunes the workload distribution (partitioning) between the parent and child kernels in DP applications. Wang *et al.* [42] proposed a mechanism called Dynamic Thread Block Launch (DTBL). Instead of launching child kernels, they propose to launch child CTAs and coalesce them with an existing kernel, thereby removing the launch overheads associated with launching kernels. However, this coalescing of CTA to an existing kernel can happen only when the CTA is equal in dimensions to the CTAs in the existing kernel and have the same instruction sequence for execution. This reduces the applicability of the scheme to a limited set of programs. Also, the number of CTAs launched remains the same in DTBL, which still incur significant queuing latency if the concurrent CTA limitation is reached. We show results from three representative applications in Figure 21. `SA` is bottlenecked due to concurrent CTA limitation and SPAWN outperforms DTBL by 1.8× and 1.4× in thaliana and elegans [1], respectively. `MM` launches a lot of large child kernels and suffers from both launch overhead and queuing latency. SPAWN and DTBL perform similarly in this scenario. SPAWN is able to reduce both the launch overheads and queuing latency while DTBL largely eliminates only the launch overhead. DTBL performs better than SPAWN in `SSSP` because `SSSP` launches small child kernels and the execution is bottlenecked by the launch overhead, which DTBL is designed to eliminate.

[8]With an intensive child kernel launch rate, the launch overhead and queuing latency gets exposed when there is lack of work in the GPU to hide this increased latency.

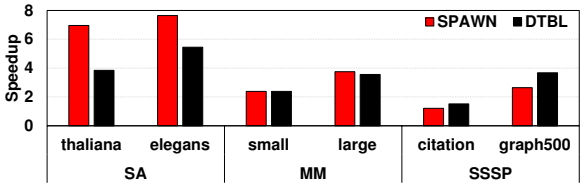


Figure 21: Comparison with DTBL [42]. Normalized performance to flat (non-DP) implementation.

VI. Related Work

To the best of our knowledge, this is the first work that dynamically tunes the workload distribution (partitioning) ratio among parent and child kernels, to find the sweet spot to minimize launch overhead and queuing latency while maximizing parallelism.

Dynamic Parallelism: Prior work on dynamic parallelism for GPUs has mainly dealt with the challenges of launch overhead. Wang *et al.* [44] characterize the overheads involved in dynamic parallel applications. They also compare the control-flow and memory behavior of the dynamic parallel applications against their non-dynamic parallel counterparts. Chen *et al.* [8] propose a compiler-based code transformation that replaces the child kernel launches in the parent threads with the child kernel code to reuse the already running parent threads. Therefore, they avoid the large runtime overheads involved in launching child kernels. Their code transformation also load balances the parent threads by reassigning the child tasks to different parent threads. In this paper, we dynamically tune the workload distribution by controlling the kernel launches, which effectively reduces not only the number of child kernels, but also the number of child CTAs. Consequently, we reduce both launch overheads and queuing latencies. Also, these overhead and latency can be hidden more effectively due to extended executions of parent threads.

Work Distribution: There has been considerable amount of research done on effectively mapping computations of conventional applications to multi threads [11, 15, 16, 17, 31, 33, 37, 40, 46]. Yang *et al.* [46] propose a compiler framework called CUDA-NP, that starts execution with a high number of threads which are activated/deactivated by control flow during runtime, essentially distributing the work among the threads. Shen et al. [37] develop a mechanism that can find an optimal partitioning of work between CPU and GPU based on the workload characteristics using a two-step quantitative model. Kim *et al.* [17] investigate a fine-grain hardware worklist for GPGPUs which acts as the center for all the warps to pick up work. This allows the work distribution to load balance itself dynamically during the source of execution.

VII. Conclusions

Although GPUs can be very effective in executing parallel programs, many irregular applications (e.g. graph algorithms with irregular data inputs) that have been ported to GPUs execute inefficiently due to the workload imbalances across

its threads. Dynamic parallelism supported by OpenCL and CUDA help in reducing this imbalance by allowing GPU kernels to launch additional kernels on-demand without involving the CPU. However, this approach entails extra performance overheads for launching child kernels; and a straightforward way of launching kernels can lead to both resource underutilization and uneven work across concurrently-executing kernels. Our proposed hardware-based solution, SPAWN, improves the GPU performance by hiding and reducing the performance overheads of child kernel launches, and improving the load balance across different kernels. Using our approach, programmers can port existing irregular applications to GPUs without having to go through extensive architecture-specific software optimizations that balance the work across different kernels.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their feedback. This research is supported in part by NSF grants #1205618, #1213052, #1212962, #1302225, #1302557, #1313560, #1320478, #1320531, #1409095, #1409723, #1439021, #1439057, #1526750, #1629129 and #1629915. Adwait Jog also acknowledges the start-up grant from College of William and Mary. AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

REFERENCES

[1] "National center for biotechnology information," http://www.ncbi.nlm.nih.gov, online, 2016.
[2] V. Adhinarayanan *et al.*, "Measuring and Modeling On-Chip Interconnect Power on Real Hardware," in *IISWC*, 2016.
[3] AMD, "AMD APP SDK OpenCL Optimization Guide," 2013.
[4] AMD, "AMD APP SDK OpenCL User Guide," 2013.
[5] A. Bakhoda *et al.*, "Analyzing CUDA workloads using a detailed GPU simulator," in *ISPASS*, 2009.
[6] M. Burtscher *et al.*, "A Quantitative Study of Irregular Programs on GPUs," in *IISWC*, 2012.
[7] S. Che *et al.*, "Pannotia: Understanding Irregular GPGPU Graph Applications," in *IISWC*, 2013.
[8] G. Chen and X. Shen, "Free Launch: Optimizing GPU Dynamic Kernel Launches Through Thread Reuse," in *MICRO*, 2015.
[9] H. Cheng *et al.*, "BitMapper: an efficient all-mapper based on bit-vector computing," in *BMC Bioinformatics*, 2015.
[10] G. Diamos *et al.*, "Relational Algorithms for Multi-bulk-synchronous Processors," in *PPoPP*, 2013.
[11] W. Ding *et al.*, "Optimizing Off-chip Accesses in Multicores," in *PLDI*, 2015.
[12] S. Hong *et al.*, "Accelerating CUDA Graph Algorithms at Maximum Warp," in *PPoPP*, 2011.
[13] A. Jog *et al.*, "Anatomy of GPU Memory System for Multi-Application Execution," in *MEMSYS*, 2015.
[14] A. Jog *et al.*, "Exploiting Core Criticality for Enhanced Performance in GPUs," in *SIGMETRICS*, 2016.
[15] M. Kandemir *et al.*, "Memory Row Reuse Distance and Its Role in Optimizing Application Performance," in *SIGMETRICS*, 2015.
[16] O. Kayiran *et al.*, "uC-States: Fine-grained GPU Datapath Power Management," in *PACT*, 2016.
[17] J. Y. Kim and C. Batten, "Accelerating Irregular Algorithms on GPGPUs Using Fine-Grain Hardware Worklists," in *MICRO*, 2014.
[18] A. Kuhl, "Thermodynamic States in Explosion Fields," in *IDS*, 2010.
[19] M. Kulkarni *et al.*, "Optimistic Parallelism Requires Abstractions," in *PLDI*, 2007.
[20] G. Liu *et al.*, "FlexBFS: A Parallelism-aware Implementation of Breadth-first Search on GPU," in *PPoPP*, 2012.
[21] M. Mendez-Lojo *et al.*, "A GPU Implementation of Inclusion-based Points-to Analysis," in *PPoPP*, 2012.
[22] D. Merrill *et al.*, "Scalable GPU Graph Traversal," in *PPoPP*, 2012.
[23] L. Nai *et al.*, "GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions," in *SC*, 2015.
[24] NVIDIA, "CUDA C/C++ SDK Code Samples."
[25] NVIDIA, "JP Morgan Speeds Risk Calculations with NVIDIA GPUs," 2011.
[26] NVIDIA, "Dynamic Parallelism in CUDA," 2012.
[27] NVIDIA, "Next Generation CUDA Compute Architecture: Kepler GK110," 2012.
[28] NVIDIA, "CUDA C Programming Guide," 2015.
[29] NVIDIA, "Profiler User's Guide," 2015.
[30] S. I. Park *et al.*, "Low-Cost, High-Speed Computer Vision using NVIDIA's CUDA Architecture," in *AIPR*, 2008.
[31] A. Pattnaik *et al.*, "Scheduling Techniques for GPU Architectures with Processing-In-Memory Capabilities," in *PACT*, 2016.
[32] G. Pratx and L. Xing, "GPU Computing in Medical Physics: A Review," in *Medical physics*, 2011.
[33] S. Puthoor *et al.*, "Implementing Directed Acyclic Graphs with the Heterogeneous System Architecture," in *GPGPU*, 2016.
[34] T. G. Rogers *et al.*, "Cache-Conscious Wavefront Scheduling," in *MICRO*, 2012.
[35] P. Sanders and C. Schulz, "10th Dimacs Implementation Challenge-Graph Partitioning and Graph Clustering," 2012.
[36] I. Schmerken, "Wall Street Accelerates Options Analysis with GPU Technology," 2009.
[37] J. Shen *et al.*, "Improving Performance by Matching Imbalanced Workloads with Heterogeneous Platforms," in *ICS*, 2014.
[38] S. S. Stone *et al.*, "Accelerating advanced MRI reconstructions on GPUs," *J. Parallel Distributed Computing*, 2008.
[39] X. Tang *et al.*, "A Video Coding Benchmark Suite for Evaluation of Processor Capability," in *SNPD*, 2013.
[40] X. Tang *et al.*, "Improving Bank-Level Parallelism for Irregular Applications," in *MICRO*, 2016.
[41] Y. Ukidave *et al.*, "NUPAR: A Benchmark Suite for Modern GPU Architectures," in *ICPE*, 2015.
[42] J. Wang *et al.*, "Dynamic Thread Block Launch: A Lightweight Execution Mechanism to Support Irregular Applications on GPUs," in *ISCA*, 2015.
[43] J. Wang *et al.*, "LaPerm: Locality Aware Scheduler for Dynamic Parallelism on GPUs," in *ISCA*, 2016.
[44] J. Wang and Y. Sudhakar, "Characterization and Analysis of Dynamic Parallelism in Unstructured GPU Applications," in *IISWC*, 2014.
[45] H. Wu *et al.*, "Compiler-Assisted Workload Consolidation For Efficient Dynamic Parallelism on GPU"," in *IPDPS*, 2016.
[46] Y. Yang and H. Zhou, "CUDA-NP: Realizing Nested Thread-level Parallelism in GPGPU Applications," in *PPoPP*, 2014.