Dynamic Thread Block Launch: A Lightweight Execution Mechanism to Support Irregular Applications on GPUs

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

Abstract

GPUs have been proven effective for structured applications that map well to the rigid 1D-3D grid of threads in modern bulk synchronous parallel (BSP) programming languages. However, less success has been encountered in mapping data intensive irregular applications such as graph analytics, relational databases, and machine learning. Recently introduced nested device-side kernel launching functionality in the GPU is a step in the right direction, but still falls short of being able to effectively harness the GPUs performance potential.

We propose a new mechanism called Dynamic Thread Block Launch (DTBL) *to extend the current bulk synchronous parallel model underlying the current GPU execution model by supporting dynamic spawning of lightweight thread blocks. This mechanism supports the nested launching of thread blocks rather than kernels to execute dynamically occurring parallel work elements. This paper describes the execution model of DTBL, device-runtime support, and microarchitecture extensions to track and execute dynamically spawned thread blocks. Experiments with a set of irregular data intensive CUDA applications executing on a cycle-level simulator show that DTBL achieves average 1.21x speedup over the original flat implementation and average 1.40x over the implementation with device-side kernel launches using CUDA Dynamic Parallelism.*

1. Introduction

There has been considerable success in harnessing the superior compute and memory bandwidth of GPUs to accelerate traditional scientific and engineering computations [3][24][31][30]. These computations are dominated by structured control and data flows across large data sets that can be effectively mapped to the 1D-3D massively parallel thread block structures underlying modern bulk synchronous programming languages for GPUs such as CUDA [27] and OpenCL[17]. However, emerging data intensive applications in analytics, planning, retail forecasting and similar applications are dominated by sophisticated algorithms from machine learning, graph analysis, and deductive reasoning characterized by irregular control, data, and memory access flows. Thus, it is challenging to effectively harness data parallel accelerators such as GPUs for these applications. This paper presents, evaluates, and demonstrates an effective solution to this challenge.

We first observe that within many irregular applications, segments of the computation locally exhibit structured control and memory access behaviors. These *pockets* of parallelism occur in a data dependent, nested, time-varying manner. The CUDA Dynamic Parallelism (CDP) [28] 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. However, recent studies have shown that while these extensions do address the productivity and algorithmic issues, the ability to harness modern high performance hardware accelerators such as GPUs is still difficult in most cases [36][39]. This is primarily due to the cost of a device-side kernel launch that is in turn a function of the semantics of kernels.

We also observe that much of the dynamic parallelism that occurs in these applications can be effectively harnessed with a lighter weight mechanism to spawn parallel work. Accordingly we introduce the *Dynamic Thread Block Launch (DTBL)* mechanism to launch light weight thread blocks dynamically and on demand from GPU threads. These thread blocks semantically correspond to a cooperative thread array (CTA) in CUDA or a work group in OpenCL. The overhead of launching a thread block is considerably smaller than launching a kernel. The finer granularity of a thread block provides effective control of exploiting smaller-scale, dynamically occurring pockets of parallelism during the computation. We extend the GPU microarchitecture to integrate the execution of these dynamically spawned parallel work units into the base GPU microarchitecture. The results show improved execution performance for irregular applications with relatively modest hardware investments and minimal extensions to the base execution model in CUDA or OpenCL. In fact, DTBL can be integrated within existing languages retaining device-side nested kernel launch capability and reusing much of the existing device API stack

ISCA'15, June 13-17, 2015, Portland, OR, USA

http://dx.doi.org/10.1145/2749469.2750393

without any impact on applications that do not use DTBL functionality. Consequently extensions to the device side runtime are minimal.

Accordingly this paper makes the following contributions.

1. We introduce DTBL as an effective lightweight execution mechanism for spawning dynamically created parallel work. DTBL can be used to effectively harness the compute and memory bandwidth of GPUs for data intensive irregular applications.
2. We implement a device side runtime API for supporting DTBL with a single call by leveraging the API for CUDA Dynamic Parallelism.
3. The base GPU microarchitecture is extended at a modest cost to support the dynamic generation, tracking, and efficient management of the spawned thread blocks.
4. An evaluation from multiple performance perspectives of several CUDA applications is presented executing on a cycle-level simulator. DTBL achieves average 1.21x speedup over the original implementation with flat GPU programming methodology and average 1.40x over the implementation with device-side kernel launches using CDP.

The remainder of the paper introduces DTBL, microarchitecture extensions to support DTBL, a detailed description of its execution behavior, and the evaluation across a range of representative benchmarks.

2. Baseline GPU Architecture and Computation Execution

This section provides a brief background on modern general purpose GPUs and the details of their execution model.

2.1. Baseline GPU Architecture and Execution Model

We model the baseline GPU architecture according to the NVIDIA GK110 Kepler architecture so we adopt the NVIDIA and CUDA terminology. However, the basic ideas and resulting analysis are applicable to other GPU architectures. In Figure 1, a GPU is connected to the host CPU by the interconnection bus and accepts operation commands such as memory copy and kernel launching from the CPU. In the CUDA programming model, programmers express a GPU program as a kernel function that specifies a 1D-3D array of thread blocks called cooperative thread arrays (CTAs), all threads executing the same code. Thread blocks are scheduled and executed on the GPU computation units called Streaming Multiprocessors (SMXs), each of which features multiple cores, registers and a scratch memory space used as L1 cache or shared memory. The GPU connects to the L2 cache and then to the DRAM through the memory controller and uses it as global memory.

2.2. Host-side Kernel Launching and Scheduling

The CPU launches GPU kernels by dispatching kernel launching commands. Kernel parameters are passed from CPU to GPU at the kernel launching time and stored in the GPU global

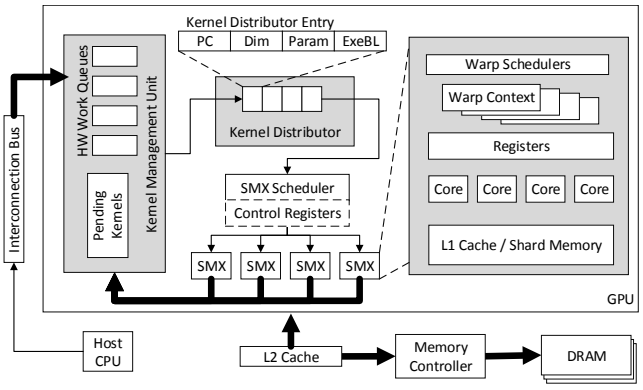

Figure 1: Baseline GPU Architecture

memory with necessary alignment requirements. The parameter addresses are part of the launching command along with other kernel information such as dimension configuration and entry PC address. All the launching commands are passed to the GPU through software stream queues (e.g. CUDA stream). Kernels from different streams are independent from each other and may be executed concurrently while kernels from the same stream should be executed in the order that they are launched. The streams are mapped to Hardware Work Queues (HWQ) in the GPU that create hardware-managed connections between the CPU and the GPU. The current generation of NVIDIA GPUs introduce Hyper-Q[26] - a technique which constructs multiple HWQs and maps individual streams to each HWQ to realize concurrency. However, if the number of software streams exceeds the number of HWQ, some of them will be combined and serialized. In our baseline GPU architectures, the Kernel Management Unit (KMU) manages multiple HWQs by inspecting and dispatching kernels at the head of the queue to the Kernel Distributor. Once the head kernel is dispatched, the corresponding HWQ stops being inspected by the KMU until the head kernel completes. The KMU also manages all the kernels dynamically launched or suspended by an SMX (e.g. through use of the CUDA Dynamic Parallelism feature) as discussed in section 2.4.

The Kernel Distributor holds all the active kernels ready for execution. The number of entries in the Kernel Distributor is the same as that of HWQs (32 in the GK110 architecture) as this is the maximum number of independent kernels that can be dispatched by the KMU. Each entry manages a set of registers that record the kernel status including kernel entry PC, grid and thread block dimension information, parameter addresses and the number of thread blocks to complete. The SMX scheduler takes one entry from the Kernel Distributor in first-come-first-serve (FCFS) order and sets up the SMX control registers according to the kernel status. It then distributes the thread blocks of the kernel to each SMX limited by the maximum number of resident thread block, threads, number of registers, and shared memory space per SMX.

During execution, a thread block is partitioned into groups of 32 threads called a *warp* as the basic thread group executed

on a 32-lane SIMD unit. SMXs maintain the warp contexts for the lifetime of the thread block. The warp scheduler selects a warp from all the resident warps on the SMX that have no unresolved dependency according to a scheduling policy (e.g. round-robin) and then issues its next instruction to the cores. By interleaving warps, an SMX is able to achieve hardware multithreading and hide memory latency. All the threads in a warp execute the same instruction in a lock-step fashion. When there is a branch and threads in a warp take different paths, the execution of threads on different paths will be serialized. This is referred to as control flow divergence and results in low SIMD lane utilization. Our baseline architecture uses a PDOM reconvergence stack [13] to track and reconverge the threads that take different branches. Memory accesses generated by 32 threads in a warp for consecutive word addresses are coalesced into one memory transaction. Otherwise memory instructions are replayed multiple times which may increase the memory access latency for the entire warp. This pattern of irregular memory accesses is referred as memory divergence [22]. The SMX scheduler keeps updating the control register and the register in each Kernel Distributor entry to reflect the number of thread blocks that remain to be scheduled as well as those still being executed. When all the thread blocks of a kernel finish, the Kernel Distributor will release the corresponding kernel entry to accept the next kernel from the KMU.

2.3. Concurrent Kernel Execution

Concurrent kernel execution is realized by distributing thread blocks from different kernels across one or more SMXs. If one kernel does not occupy all SMXs, the SMX scheduler takes the next kernel and distributes its thread blocks to the remaining SMXs. When a thread block finishes, the corresponding SMX notifies the SMX scheduler to distribute a new blocks either from the same kernel or from the next kernel entry in the Kernel Distributor if the current kernel does not have any remaining thread blocks to distribute and the SMX has enough resources available to execute a thread block of the next kernel. Therefore, multiple thread blocks from different kernels can execute on the same SMX [11]. Large kernels which either have many thread blocks or use a large amount of resources are not likely to be executed concurrently. On the other hand, if the kernels in the Kernel Distributor only use a very small amount of SMX resources, the SMX may not be fully occupied even after all the kernels in the Kernel Distributor are distributed, which results in under-utilization of the SMX.

2.4. Device-Side Kernel Launch

Recent advances in the GPU programming model and architecture support device-side kernel launches - *CUDA Dynamic Parallelism* [28] - which provides the capability of launching kernels dynamically from the GPU. In this model, parent kernels are able to launch nested child kernels by invoking several device-side API calls to specify the child kernel configuration, setup parameters, and dispatch kernels through device-side software streams to express dependencies. Child kernels can start any time after they are launched by their parents. However, parent kernels may request explicit synchronization with the children, which results in an immediate yielding from parent to child. Device-side kernel launching preserves the original memory model used in the GPU: global memory is visible to both parent and children while shared memory and local memory are private and cannot be passed to each other.

In our baseline GPU architecture, there is a path from each SMX to the KMU so that all the SMXs are able to issue new kernel launching commands to the KMU. Similar to host-side launched kernels, parameters are stored in the global memory and the address is passed to the KMU with all other configurations. When a parent decides to yield to a child kernel, the SMX suspends the parent kernel and notifies the KMU to hold the suspended kernel information. The KMU dispatches device-launched or suspended kernels to the Kernel Distributor along with other host-launched kernels in the same manner. Therefore device-launched kernels also take advantage of concurrent kernel execution capability.

Current architecture support of device-side kernel launching comes with non-trivial overhead. Wang et al. [36] analyze the CDP overhead on the Tesla K20c GPU and show that the total kernel launching time scales with the number of child kernels, decreasing the performance by an average of *36.1%*. The launching time is composed of the time spent in allocating the parameters, issuing a new launching command from SMX to the KMU, and dispatching a kernel from the KMU to the Kernel Distributor.

Device-side kernel launches also require a substantial global memory footprint. A parent may generate many child kernels which can be pending for a long time before being executed, thus requiring the GPU to reserve a fair amount of memory for storing the associated information of the pending kernels. On the other hand, the device runtime has to save the states of the parent kernel when they are suspended to yield to the child kernels at the explicit synchronization points.

3. Dynamic Parallelism

We propose to extend the BSP execution model to effectively and efficiently support dynamic parallelism. This section introduces major characteristics of dynamic parallelism and introduces our extension to the BSP model.

3.1. Parallelism in Irregular Applications

Emerging data intensive applications are increasingly irregular by operating on unstructured data such as trees, graphs, relational data and adaptive meshes. These applications have inherent time-varying, workload-dependent and unpredictable memory and control flow behavior that may cause severe workload imbalance, poor memory system performance and eventually low GPU utilization. Despite the above observations, they still exhibit **D**ynamically **F**ormed pockets of structured data **P**arallelism (**DFP**) that can locally effectively exploit the

GPU compute and memory bandwidth. For example, typical GPU implementations for vertex expansion operations that are commonly used in graph problems assign one thread to expand each vertex in the vertex frontier with a loop that iterates over all the neighbors. Since the number of neighbors for each vertex can vary, the implementation may suffer from poor workload balance across threads. However, DFP exists within each thread as the neighbor exploration can be implemented as a structured parallel loop where each neighbor is examined independently.

The introduction of device-side kernel launching in GPUs enables a implementation scheme that new child kernels are dynamically invoked for any detected DFP in irregular applications. In the vertex expansion example, the original neighbor exploration loop can be replaced by a dynamically launched kernel that employs uniform control flow. The approach can potentially increase the performance by reducing control flow divergence and memory irregularity. For some common data structures used in this problem such as Compressed Sparse Row (CSR) where neighbor IDs of each vertex are stored in consecutive addresses, parallel neighbor exploration may also generate coalesced memory accesses. Prior studies have identified several characteristics of of such implementations [36]:

High Kernel Density: Depending on the problem size, DFP in irregular applications can show substantially high density where a large number of device kernels are launched. For example, a typical breadth first search (BFS) iteration can have about 3K device kernel launches. The high DFP density results in high kernel launching overhead and memory footprint.

Low Compute Intensity: Device kernels launched for DFP are usually fine-grained and have relatively low degrees of parallelism. Measurements across several irregular applications show that the average number of threads in each device-launched kernel is around 40 which is close to the warp size.

Workload Similarity: As DFP may exist within each thread in a kernel, and all threads are identical, the operations performed by each dynamically launched kernel are usually similar. However their instantiation may be with different degrees of parallelism. As per the nature of DFP, most device-launched kernels invoke the same kernel function but can have different configurations and parameter data.

Low Concurrency and Scheduling Efficiency: DFP generated by different threads are independent of each other and are implemented by launching device kernels through different software streams to enable concurrency. Because of the low compute intensity in these device kernels and thereby the low resources usage when executed on SMX, multiple device kernels are able to execute concurrently. However, current kernel scheduling strategy on the GPU imposes a limit on kernel-level concurrency as the number of independent HWQs determines the maximum number of kernels that can be executed concurrently which is 32 in the GK110 architecture. As fine-grained device kernels can only be scheduled and executed concurrently up to this limit, there may not be enough warps to fully occupy the SMX. Consider an example where the device kernels have only 64 threads (2 warps) and 32 device kernels are concurrently scheduled to the GPU which result in 64 warps running on the SMXs simultaneously. This is only 1/13 of the maximum number of resident warps on a Tesla K20c GPU (13 SMXs, each has maximum 64 resident warps). The limited warp concurrency could potentially cause either low utilization (if some of the SMXs are not assigned with any warps) or poor memory latency hiding ability (if SMXs are assigned with small number of warps).

3.2. Light Weight Thread Blocks

While current support of device-side kernel launch on the GPU provides substantial productivity for handling DFP, the major issues of kernel launching overhead, large memory footprint, and less efficient kernel scheduling prevent the performance effective utilization of this functionality.

We propose to extend the current GPU execution model with *DTBL* where thread blocks rather than entire kernels can be dynamically launched from a GPU thread. Thread blocks (TBs) can be viewed as light weight versions of a kernel. A kernel can make nested TB calls on demand to locally exploit small pockets of parallel work as they occur in a data dependent manner. When a GPU kernel thread launches a TB, it is queued up for execution along with other TBs that are initially created by the kernel launch. Using DTBL, the set of TBs that comprise a kernel are no longer fixed at launch time but can vary dynamically over the lifetime of a kernel.

As we will demonstrate, the dynamic creation of thread blocks can effectively increase the SMX occupancy, leading to higher GPU utilization. The dynamic TB launch overhead as well as memory footprint are significantly lower than that of kernel launch. Thus, DTBL enables more efficient support of irregular applications by introducing a light weight mechanism to dynamically spawn and control parallelism.

4. Dynamic Thread Block Launch

In this section, we define the execution model of DTBL, propose the architecture extensions, and analyze the potential benefits.

4.1. DTBL Execution Model

The execution model of DTBL allows new TBs to be dynamically launched from GPU threads and coalesced with existing kernels for scheduling efficiency. We extend the current GPU BSP execution mode with several new concepts and terms to support these new features.

Figure 2 shows the execution model and thread hierarchy of DTBL. Any thread in a GPU kernel can launch multiple TBs with a single device API call (see later in this section). These TBs are composed as a single *aggregated group* utilizing a three dimensional organization similar to those of a native

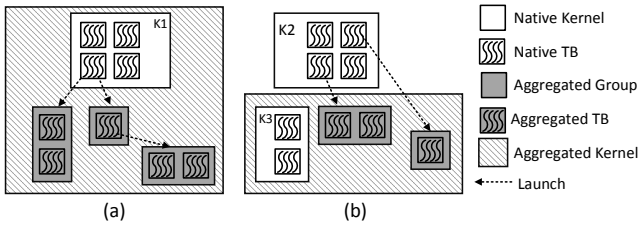

Figure 2: DTBL execution model and thread hierarchy where (a) shows the aggregated groups launched by kernel K1 are coalesced to itself and (b) shows the aggregated groups launched by kernel K2 are coalesced to another kernel K3.

kernel. An aggregated group is then coalesced with a kernel - this simply means the TBs in the aggregated groups are added to the existing pool of TBs remaining to be scheduled and executed for that kernel. In fact, an aggregated group may be coalesced with the kernel of the parent thread (Figure 2a) *or* with another kernel (Figure 2b). In either case, the newly generated aggregated group execute the same function code as the kernel with which it is coalesced, and may have different input parameter values. Multiple aggregated groups can be coalesced to a single kernel.

In DTBL, coalescing is essential to increasing the TB scheduling performance due to i) TBs with the same configuration can be scheduled together to achieve the designed occupancy for the original kernel, possibly leading to higher GPU utilization and ii) coalesced aggregated groups only require one common context setup including kernel function loading, register and shared memory partitioning which can reduce the scheduling overhead. More details are described in the microarchitecture later in Section 4.2.

The kernel that is initially launched either by the host or by the device using the kernel launching API is called a *native kernel*. The TBs that compose the native kernel are *native TBs*. TBs in an aggregated group are called *aggregated TBs*. When a native kernel is coalesced with new aggregated groups, it becomes an *aggregated kernel*.

The idea of DTBL can be illustrated with two examples. The first example is Adaptive Mesh Refinement (AMR) corresponding to the execution model in Figure 2a. The DTBL implementation uses a native kernel K1 for the initial grid where each thread may launch nested aggregated groups for recursively refining the cells that are processed by the thread. All the new aggregated groups are then coalesced with K1 which become one aggregated kernel. The second example is BFS corresponding to the execution model in Figure 2b where a parent kernel K2 assigns threads to all the vertices in the vertex frontier and each parent thread may launch new TBs to expand the vertex neighbors. The kernel K3 is a native kernel previously launched by the host or the device for vertex expansion. The new TBs generated by K2 are coalesced to K3 rather than the parent.

Thread Hierarchy Within an Aggregated TB: As in GPUs today, DTBL uses a three-dimensional thread index to identify the threads in an aggregated TB. When coalesced to a native kernel, the number of threads in each dimension of an aggregated TB should be the same as that of a native TB. Therefore, aggregated TBs use the same configuration and the same amount of resources as native TBs, minimizing the overhead when scheduled on an SMX.

Device Runtime API Calls	Description
`cudaGetParameterBuffer`	Reused from the original CUDA device runtime library to allocate parameter buffer for a new aggregated group.
`cudaLaunchAggGroup`	A new API call introduced by DTBL programming interface which launches a new aggregated group.

Table 1: List of Device Runtime API calls for DTBL

Aggregated TB Hierarchy Within an Aggregated Group: An aggregated group in DTBL is analogous to a device-launched kernel. Within an aggregated group, aggregated TBs are organized into one/two/three dimensions, identified by their three-dimensional TB indices. The value of each TB index dimension starts at zero. Similar to launching a device kernel, the programmers supply data addresses through parameters and use TB indices within an aggregated group as well as thread indices within an aggregated TB to index the data values used by each thread.

Synchronization: DTBL uses the same synchronization semantics as the current GPU execution model, i.e., threads within an aggregated TB can be synchronized explicitly by calling a barrier function. However, like the base programming model no explicit barrier is valid across native or aggregated TBs. Unlike the parent-child synchronization semantics in the device-kernel launching model in CDP, aggregated groups cannot be explicitly synchronized by its invoking kernel. Therefore, it is the programmers' responsibility to ensure the correctness of the program without any assumption on the execution order of aggregated groups. When we implement various irregular applications using device kernel launching, we avoid any explicit synchronization between the child and parents due to its high overhead in saving the parent state to the global memory, so these applications can be easily adapted to the new DTBL model. A more thorough analysis of the usage of explicit synchronization is left as future work.

Memory Model: DTBL also preserves the current GPU memory model, i.e., global memory, constant memory and texture memory storage are visible to all native and aggregated TBs. Shared memory is private to each thread block and local memory is private to each thread. No memory ordering, consistency, or coherence is guaranteed across different native or aggregated TBs.

Programming Interface: DTBL defines two device runtime API calls on top of the original CUDA Device Runtime Library for CDP listed in Table 1. The API call `cudaGetParameterBuffer` is the same as in the original device runtime library that is used to allocate parameter space for an aggregated group. The second API call `cudaLaunchAggGroup` is newly defined for dynamical-

```
__global__ parent(…) {
  cudaStream_t s;
  cudaStreamCreateWithFlags(&s, …);
  void *buf=cudaGetParameterBuffer();
  …… //fill the buf with data
  cudaLaunchDevice(child, buf,
    grDim, tbDim, sharedMem, s);
}

__global__ child(…) {
}
```

(a)

```
__global__ parent(…) {

  void *buf=cudaGetParameterBuffer();
  …… //fill the buf with data
  cudaLaunchAggGroup(child, buf,
    aggDim, tbDim, sharedMem);
}

__global__ child(…) {
}
```

(b)

Figure 3: Example code segments for (a) CDP and (b) DTBL

ly launching an aggregated group. Programmers can pass the kernel function pointer when calling this API to specify the kernel to be executed by and possibly coalesce with the new TBs. Similar to the device kernel launching API call `cudaLaunchDevice` in CDP, `cudaLaunchAggGroup` configures the new aggregated group with thread and TB numbers in each dimension, shared memory size, and parameters. Note that unlike a device kernel launching which should be configured with an implicit or explicit software stream to express dependency on other kernels, the aggregated thread groups are automatically guaranteed to be independent of each other. We show example code segments for both CDP and DTBL implementations in Figure 3 where a parent kernel launches child kernels in CDP and corresponding aggregated groups in DTBL. The similarity between the two code segments demonstrate that DTBL introduces minimal extensions to the programming interface.

4.2. Architecture Extensions and SMX Scheduling

To support the new DTBL execution model, the GPU microarchitecture is extended to process the new aggregated groups that are launched from the GPU threads. The baseline microarchitecture maintains several data structures for keeping track of deployed kernels and the TBs that comprise them. These data structures are extended to keep track of dynamically formed aggregated groups and associating them with active kernels. This is achieved in a manner that is transparent to the warp schedulers, control divergence mechanism, and memory coalescing logic. Figure 4 illustrates the major microarchitecture extensions to support DTBL. With the extended data structure and SMX scheduler, new aggregated groups are launched from the SMXs, coalescing to existing kernels in the Kernel Distributor and scheduled to execute on SMX with all other TBs in the coalesced kernel. The detailed procedure and functionality of each data structure extension are described as follows.

Launching Aggregated Groups

This is the first step that happens when the aggregated group launching API is invoked by one or more GPU threads. The SMX scheduler will react correspondingly to accept the new aggregated groups and prepare necessary information for TB coalescing in the next step.

Similar to the device kernel launching command, DTBL introduces a new aggregation operation command in the microarchitecture. This command will be issued when the aggregated group launching API calls are invoked simultaneously by one or more threads within the same warp. These aggregated group launches are then combined together to be processed by the aggregation operation command.

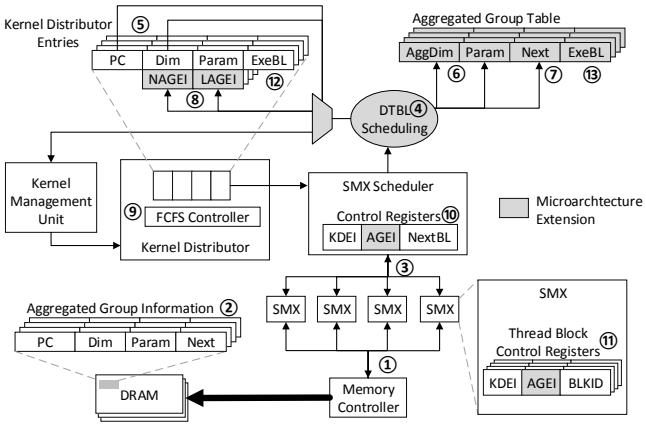

Figure 4: Microarchitecture Flow for DTBL

For each newly formed aggregated group, the SMX allocates global memory blocks through the memory controller① to store the parameters and configuration information②. The request procedure is the same as that of a device-side kernel launch. After parameters are loaded to the parameter buffer, the SMX passes the aggregation operation command to the SMX scheduler with the information for each aggregated group③.

Thread Blocks Coalescing

In this step, the SMX scheduler receives the aggregation operation command and attempts to match the newly launched aggregated groups with the existing kernels in the Kernel Distributor Entries (KDE) for TB coalescing based on aggregated group configurations. If the coalescing is successful, the SMX scheduler will push the new TBs in a scheduling TB pool for the corresponding kernel. The scheduling pool is implemented with several registers in microarchitecture to form a linked-list data structure for efficient TB scheduling. The process is implemented as a new part of the DTBL scheduling policy④ which is illustrated in Figure 5 and described in the following.

For each aggregation group, the SMX scheduler first searches the KDE to locate any existing eligible kernels that can accept the new TBs⑤. Eligible kernels should have the same entry PC addresses and TB configuration as the aggregated group. If none are found, the aggregated group is launched as a new device kernel. Our experiments show that an aggregated group is able to match eligible kernels on average 98% of the time. Mismatches typically occur early, before newly generated device kernels fill the KDE.

If an eligible kernel is found, the SMX scheduler allocates an entry in the *Aggregated Group Table* (AGT) with the three-dimensional aggregated group size and the parameter address⑥. The AGT is composed of multiple Aggregated Group Entries (AGE) and serves to track all the aggregated

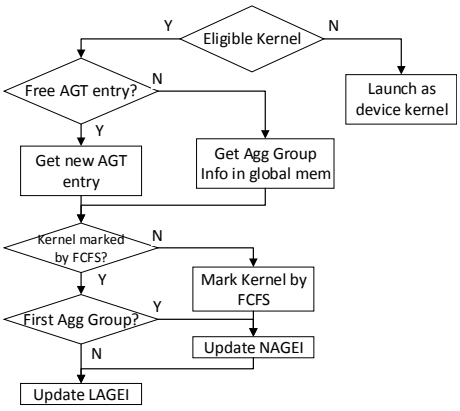

Figure 5: DTBL scheduling procedure in SMX scheduler

groups. Aggregated groups that are coalesced to the same eligible kernel are linked together with the *Next* field of the AGE⑦. The AGT is stored on chip for fast accesses with a limit on the number of entries. When the SMX scheduler searches for a free entry in the AGT, it uses a simple hash function to generate the search index instead of a brute-force search. The hash function is defined as *ind = hw_tid & (AGT_size - 1)* where *hw_tid* is the hardware thread index in each SMX and *AGT_size* is the size of AGT. The intuition behind the hash function is that all threads on an SMX have the same probability in launching a new aggregated group. The SMX scheduler is able to allocate an entry if the entry indexed by *ind* in AGT is free and the *ind* is recorded as aggregated group entry index (AGEI). Otherwise it will record the pointer to global memory where the aggregated group information is stored②.

Now that an eligible kernel and the corresponding KDE is found, the TBs in the new aggregated group are added to the set of TBs in the eligible kernel waiting be executed. The AGEI or the global memory pointer of the new aggregated group information is used to update the two KDE registers *Next AGEI* (NAGEI) and *Last AGEI* (LAGEI) if necessary⑧. NAGEI indicates the next aggregated group to be scheduled in the kernel. It is initialized when a kernel is newly dispatched to the Kernel Distributor to indicate no aggregated groups exists for the kernel. LAGEI indicates the last aggregated group to be coalesced to this kernel.

All the kernels in the Kernel Distributor are marked by the FCFS⑨ with a single bit when they are queued to be scheduled and unmarked when all its TBs are scheduled. We extend FCFS controller with an extra bit to indicate if it is the first time the kernel is marked by the FCFS. This is useful when the SMX scheduler attempts to update NAGEI under two different scenarios.

At the first scenario, when a new aggregated group is generated, the corresponding eligible kernel may have all its TBs scheduled to SMXs, be unmarked by the FCFS controller and only be waiting for its TBs to finish execution. In this case, the NAGEI is updated with the new aggregated group and the kernel is marked again by the FCFS controller so that the new aggregated group can be scheduled the next time the kernel is selected by the SMX scheduler.

At the other scenario, the eligible kernel is still marked by FCFS as it is either waiting in the FCFS queue or is being scheduled by the SMX scheduler. In this case, there are still TBs in the eligible kernel to be scheduled and NAGEI is only updated when the new aggregated group is the first aggregated group to be coalesced to this kernel.

Unlike NAGEI, LAGEI is always updated every time a new aggregated group is generated for the kernel to reflect the last aggregate group to be scheduled. With NAGEI, LAGEI and the *Next* field of AGE, all the aggregated groups coalesced to the same kernel are linked together to form a scheduling pool.

Aggregated Thread Blocks Scheduling on SMX

The last step in DTBL scheduling manages to schedule all the aggregated TBs on the SMXs. The SMX scheduler first determines whether the native kernel or a specific aggregated group should be scheduled according to the registers value generated by the previous step for the scheduling pool. Then it distributes the TBs in the kernel or the aggregated group to the SMXs with a set of registers to track their status. As described in the follows, this is implemented by updating the algorithm used by the baseline GPU microarchitecture to distribute and execute the native TBs.

When the SMX scheduler receives a kernel from the Kernel Distributor, it checks if it is the first time the kernel is marked by the FCFS controller. If so, the SMX scheduler starts distributing the native TBs followed by aggregated TBs pointed to by the NAGEI (if any). Otherwise it directly starts distributing the aggregated thread blocks pointed by NAGEI since the native TBs have already been scheduled when the kernel was previously dispatched by the FCFS controller. Another possibility is that the new aggregated groups are coalesced to a kernel that is currently being scheduled, the SMX scheduler will then continue to distribute the new aggregated groups after finishing distributing the TBs from the native kernel or current aggregated group. The SMX scheduler updates the NAGEI every time after finishing scheduling the current aggregated group and starts the next aggregated group indicated by the *Next* field of AGE pointed by NAGEI.

Once the SMX scheduler determines the native kernel or aggregated group to schedule, it records the corresponding index of KDE (KDEI) and AGEI in its control registers (SSCR)⑩. SSCR also has a *NextBL* field to store the index of the next TB to be distributed to the SMX. Note that since the TBs in the native kernel and the aggregated groups have the same configuration and resource usage as constrained by the DTBL execution model, the SMX scheduler can use a static resource partitioning strategy for both the native and aggregated TBs, saving the scheduling cost.

The SMX scheduler then distributes TBs to each SMX. The Thread Block Control Register (TBCR)⑪ on each SMX is updated correspondingly using the same value of *KDEI*

and *AGEI* in SSCR to record the kernel index in the Kernel Distributor and the aggregated group index in the AGT so the SMX can locate the function entry and parameter address correctly for the scheduled TB. The *BLKID* field records the corresponding TB index within a kernel or an aggregated group. Once the TB finishes execution, the SMX notifies the SMX scheduler to update the *ExeBL* field in the KDE⑫ and AGE⑬ which track the number of TBs in execution.

When all the TBs of the last aggregated group marked by LAGEI have been distributed to an SMX, the SMX scheduler notifies the FCFS controller to unmark the current kernel to finish its scheduling. The corresponding entries in the Kernel Distributor or AGT will be released once all the TBs complete execution.

4.3. Overhead Analysis

The hardware overhead is caused by extra data structures introduced by the architectural extensions (shaded boxes in Figure 4). New fields in the KDE (NAGEI and LAGEI), FCFS Controller (the flag to indicate if the kernel has been previously dispatched), SSCR (AGEI) and SMX TBCR (AGEI) together take 1096 Bytes of on-chip SRAM. The size of AGT determines how many pending aggregated groups can be held on-chip for fast accesses. A 1024-entry AGT takes 20KB of on-chip SRAM (20Bytes per entry) which composes the major hardware overhead (about 0.5% of the area taken by the shared memory and registers on all SMXs). In Section 5.2D we analyze the sensitivity of performance to the size of the AGT.

The major timing overhead of launching aggregated groups includes time spent on allocating parameters, searching the KDE and requesting free AGT entries. As discussed before, launching from the threads within a warp are grouped together as a single command. Therefore, the overhead is evaluated on a per-warp basis. The procedure of allocating a parameter buffer for an aggregated group is the same as that for a device-launch kernel, so we use the measurement directly from a K20c GPU. The search for eligible KDE entry can be pipelined for all the simultaneous aggregated groups launches in a warp, which takes a maximum of 32 cycles (1 cycle per entry). Searching for a free entry in AGT only takes one cycle with the hash function for each aggregated group. If a free entry is found, there will be zero cost for the SMX scheduler to load the aggregated group information when it is scheduled. Otherwise the SMX scheduler will have to load the information from the global memory and the overhead is dependent on the global memory traffic. It should be noted that allocating the parameter buffer and searching the KDE/AGE can happen in parallel, the slower of which determines the overall time overhead of aggregated group launching.

An alternative approach to the proposed microarchitecture extension is to increase the number of KDE entries so that each aggregated group can be independently scheduled from KDE. The argument is that the hardware overhead introduced by AGT could be potentially saved. However, there are also some major side effects for this approach.

First, since aggregated groups are scheduled independently, they are not coalesced so that TBs with different configurations are more likely to be executed on the same SMX. In consequence, the designed occupancy for the original kernels is less likely to be achieved and the execution efficiency could be decreased. For the same reason, the context setup overhead such as kernel function loading and resource allocation across SMXs could be increased. The context setup overhead is expected to scale with the number of aggregated group scheduled from KDE.

Second, hardware complexity and scheduling latency in the KMU and FCFS controller scales with number of KDE. For example, the number of HWQ could be increased to keep up the kernel concurrency, and the overhead for FCFS controller to track and manage the status of each aggregated group also increases linearly.

4.4. Benefits of DTBL

DTBL is beneficial primarily for the following three reasons. First, compared to device-side kernel launching, dynamic TB launches have less overhead. Instead of processing the device-side launching kernel command through a long path from the SMX to KMU and then to the Kernel Distributor, TBs are directly grouped with active kernels in the Kernel Distributor by the SMX scheduler. For irregular applications that may generate a large amount of dynamic workload, reducing the launch overhead can effectively improve the overall performance.

Second, due to the similarity of the dynamic workload in irregular applications, dynamically generated TBs are very likely to be coalesced to the same kernel which enables more TBs to be executed concurrently. Recall that the concurrent execution of fine-grained device kernels are limited by the size of Kernel Distributor. The DTBL scheduling breaks this limit as aggregated TBs are coalesced into a single native kernel that can take full advantage of the TB level concurrency on the SMX. This more efficient scheduling strategy may increase the SMX occupancy which is beneficial in increasing GPU utilization, hiding memory latency and increasing the memory bandwidth.

Third, both the reduced launch latency and increased scheduling efficiency helps to consume the dynamically launched workload faster. As the size of reserved global memory depends on the number of pending aggregated groups, DTBL can therefore reduce the global memory footprint.

5. Experiments and Evaluation

5.1. Methodology

We perform the experiments on the cycle-level GPGPU-Sim simulator [5]. We first configure GPGPU-Sim to model the Tesla K20c GPU as our baseline architecture. The configu-

SMX Clock Freq.	706MHz
Memory Clock Freq.	2600MHz
# of SMX	13
Max # of Resident Thread Blocks per SMX	16
Max # of Resident Threads per SMX	2048
# of 32-bit Registers per SMX	65536
L1 Cache / Shared Mem Size per SMX	16KB / 48KB
Max # of Concurrent Kernels	32

Table 2: GPGPU-Sim Configuration Parameters

`cudaStreamCreateWithFlag` (CDP only)	7165
`cudaGetParameterBuffer` (CDP and DTBL)	b: 8023, A: 129
`cudaLaunchDevice` (CDP only)	b: 12187, A: 1592
Kernel dispatching	283

Table 3: Latency Modeling for CDP and DTBL (Unit: cycles)

Application	Input Data Set
Adaptive Mesh Refinement (AMR)	Combustion Simulation[18]
Barnes Hut Tree (BHT) [8]	Random Data Points
Breadth-First Search (BFS) [23]	Citation Network[4] USA Road Network[4] Cage15 Sparse Matrix [4]
Graph Coloring (CLR) [10]	Citation Network[4] Graph 500 Logn20[4] Cage15 Sparser Matrix [4]
Regular Expression Match (REGX) [37]	DARPA Network Packets [21] Random String Collection
Product Recommendation (PRE) [25]	Movie Lens [16]
Relational Join (JOIN) [12]	Uniform Distributed Data Gaussian Distributed Data
Single Source Shortest Path (SSSP) [19]	Citation Network[4] Fight Network [1] Cage15 Sparser Matrix[4]

Table 4: Benchmarks used in the experimental evaluation.

ration parameters are shown in Table 2. We also modify the SMX scheduler to support concurrent kernel execution on the same SMX. The warp scheduler is configured to use the greedy-then-oldest scheduling policy [32]. As discussed before, our proposed microarchitecture extension is transparent to the warp scheduler so DTBL can take advantage of any warp scheduling optimization that is useful to the baseline GPU architecture.

To support the device-side kernel launch capability (CDP on K20c), we extend the device runtime of GPGPU-Sim with the implementation of corresponding API calls. We model the latency of these API calls which is part of the kernel launching overhead by performing the measurement on the K20c GPU with the `clock()` function and use the average cycle values from 1,000 measurements across all the evaluated benchmarks. According to our measurements, the API `cudaGetParameterBuffer` and `cudaLaunchDevice` have a linear latency model per warp basis denoted as $Ax+b$ where b is the initialization latency for each warp, A is the latency for each API called by one thread in the warp and x is the number of the threads calling the API in a warp. Note that the execution of the device API calls will be interleaved for all the warps so that some portion of the latency introduced can also be hidden by the interleaving, similar as the memory latency hiding. Besides the API latency, there is also a kernel dispatching latency (from KMU to Kernel Distributor). We measure this using the average time difference between the end of the first kernel and the start of the second kernel that is dependent of the first kernel.

We verify the accuracy of the simulation, especially the kernel launching overhead, by running all the benchmarks both on the K20c GPU and the simulator and use the same correlation computation method by GPGPU-Sim. We also implement the proposed architecture extension for DTBL with overhead assignment described in Section 4.3 where the latency for parameter buffer allocation is the same as the `cudaGetParameterBuffer` API call and all other aggregated group launching latency is directly modeled by the microarchitecture extension. The latency numbers used in the simulator are shown in Table 3.

We select 8 irregular applications with different input data sets as shown in Table 4. The source code of these applications are from the latest benchmark suites or implemented as described in recently published papers. We refer to the original CUDA implementations as *flat* implementations since the nested algorithmic structure is flattened and effectively serialized within each thread. An exception is the *bfs* implementation [23] where dynamic parallelism for DFP is implemented by employing TB and warp level vertex expansion techniques. For this application, our implementation uses CDP device kernels or DTBL aggregated group to replace the TB or warp level vertex expansion. We implemented the benchmarks with CDP in the way that a device kernel is launched for any DFP with sufficient parallelism available. The same methodology applies to DTBL except that a device kernel is replaced with an aggregated group for a fair comparison. Note that the data structures and algorithms of the original implementations are not changed in the CDP/DTBL implementations for a fair comparison. The proposed DTBL model for dynamic parallelism can also be orthogonal to many optimizations, e.g. worklist for work pulling and pushing to achieve high-level workload balance, as they can be applied in either flat or nested implementations.

DTBL only uses device runtime API for thread block launching and does not introduce any new instructions or syntax. Therefore, we use the CUDA compiler NVCC6.5 directly to compile the benchmarks. Extending the syntax to support higher-level programming interface similar as the CUDA kernel launching annotation "`<<<>>>`" is left as future work. We use the same dataset for the GPU and the simulator and run the entire application from the beginning to the end except for *regx*. We break *regx* into several sections, manually populate the memory in GPGPU-Sim, and run only computation kernels. We then trace all the computation kernels of the benchmarks to generate the performance data.

5.2. Result and Analysis

In this section we report the evaluation and analysis of the benchmark in various performance aspects.

A. Control Flow and Memory Behavior

We evaluate the control flow behavior using the warp activity percentage which is defined as average percentage of active threads in a warp as shown in Figure 6, and memory behavior using DRAM efficiency which is computed as dram_efficiency=(n_rd+n_write)/n_activity where n_rd and n_write are the number of memory read and write commands issued by the memory controller and n_activity is the active cycles when there is a pending memory request. DRAM efficiency reveals the memory bandwidth utilization and increases when there are more coalesced memory accesses in a given period of time, as shown in Figure 7. On average, warp activity percentage of both CDP and DTBL increases 10.7% from the flat implementations and DRAM efficiency increases 0.029 or 1.14x for CDP and 0.053 or 1.27x for DTBL, demonstrating that one important benefit of both CDP and DTBL is to dynamically generate parallel workload for DFP that have more regular control flow and coalesced memory accesses.

Since both DTBL and CDP launch dynamic parallel workloads, they fundamentally behave the same in reducing control flow divergence and obtain the same amount of increase in warp activity percentage. Some benchmarks, such as *amr* and *join_gaussian*, have highly irregular computation workload and severe imbalance problem across the threads in their flat implementations and achieve most substantial increases in warp activity percentage (45.3% and 21.3%). We can also observe warp activity percentage increase for the two *bfs* benchmarks. Although the baseline *bfs* implementation has already utilized TB-level and warp-level vertex expansion to handle dynamic parallelism, CDP and DTBL are able to use variable TB sizes to achieve even better workload balance. The benchmark *clr_graph500* does not show obvious changes and *clr_cage15* even shows a slight drop (-5.9%) because the graphs *graph500* and *cage15* already have relatively small variance in vertex degree that generate balanced workload even in the flat implementation. Launching dynamic parallel workloads for some vertices but not for others may break the original balance and cause more control flow divergence. This is consistent with the understanding that CDP and DTBL are intended to work well over unbalanced workloads.

The *clr_cage15* and *sssp_cage15* are two benchmarks that achieve highest DRAM efficiency increase. In their flat implementations, as the graph *cage15* has a distributed neighbor list, the threads access vertex data far away from each other in memory and result in more non-coalesced memory accesses and memory transactions. In comparison, CDP and DTBL implement the DFP such that threads are more likely to access consecutive memory addresses. Memory irregularity could be significantly reduced in this case which is demonstrated by increasing DRAM efficiency.

B. Scheduling Performance

By coalescing dynamically generated TBs to existing kernels on the fly, DTBL is able to increase the TB-level concurrency for fine-grained parallel workloads. Lower launching latency

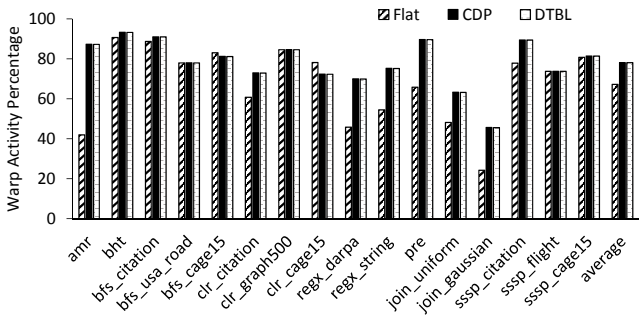

Figure 6: Average Percentage of Active Threads in a Warp

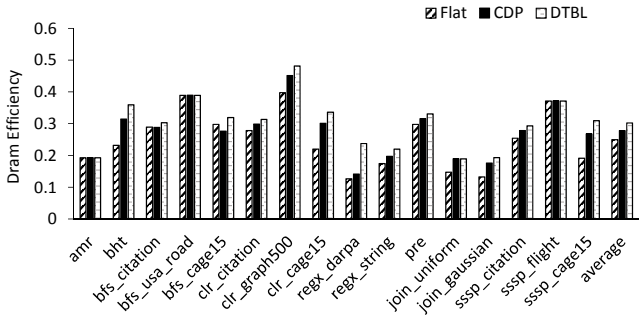

Figure 7: DRAM Efficiency

for DTBL also contributes to the increase in the number of available TBs that can be scheduled concurrently by the SMX scheduler. Therefore, DTBL is able to outperform CDP by increasing SMX occupancy. We evaluate the SMX occupancy by measuring the average number of active warps in each cycle on all of the SMXs divided by maximum number of resident warps per SMX. We isolate the influences of scheduling strategy and launching latency by comparing the measurement with and without launching latency. The results are shown in Figure 8 where the SMX occupancy achieved by CDP and DTBL without modeling launching latency are denoted as CDP-Ideal (CDPI) and DTBL-Ideal (DTBLI) respectively. DTBLI has average of 17.9 or 1.24x increase over CDPI. The benchmark *bht* achieves the highest occupancy increases (24.6 or 1.38x) since it generates many fine-grained parallel workloads (average number of threads in a device kernel or an aggregated group is 33.4 which is close to warp size). Therefore, in its CDP implementation, the limited kernel-level concurrency on the GPU causes only a few threads to be active on GPUs which results in low SMX occupancy and utilization. DTBL, on the other hand, is able to aggregate all these fined-grained TBs together to fully utilize SMX with a higher occupancy. Other benchmarks that generate dynamic workloads with higher parallelism (coarse-grain workload) have less of a significant increase in SMX occupancy, represented by *pre* (0.46 or 1.01x) with an average 1527.9 threads in a device kernel or an aggregated group.

If the launching latency is included for both CDP and DTBL, SMX occupancy decreases from both CDPI and DTBLI (average -10.7 or -13.5% for CDP and -5.2 or -7.6% for DTBL).

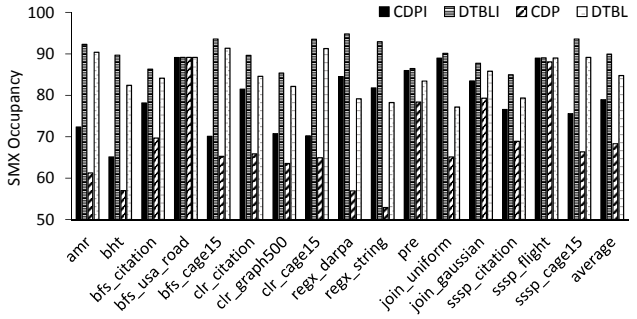

Figure 8: SMX Occupancy

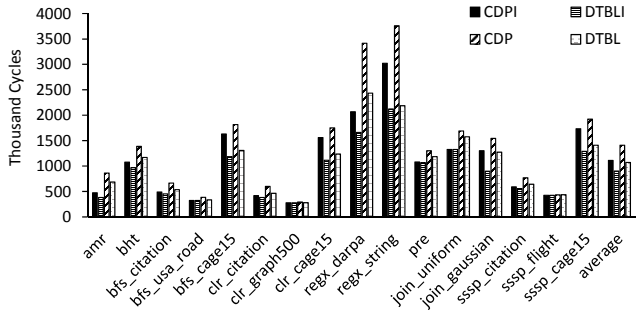

Figure 9: Average Waiting Time for a Kernel or an Aggregated Group

The launching latency for a device kernel is higher than an aggregated group and causes a larger drop in SMX occupancy for CDP. In *regx_string*, DFP has high occurrence and generates a large number of dynamic parallel workloads. While DTBLI outperforms CDPI in SMX occupancy (11.2 or 1.14x) because of the increased thread block concurrency, the launching latency even enlarges the gap (25.4 or 1.48x). The increased SMX occupancy of DTBL also improves the DRAM efficiency as shown in Figure 7 (average 0.022 or 1.08x higher than CDP) because of the memory latency hiding capability.

We further evaluate DTBL scheduling efficiency by comparing the average waiting time (time between launching and starting execution) and memory footprint for dynamically generated kernels or aggregated groups as shown in Figure 9 and Figure 10 respectively. Again, we compare the waiting time with and without launching latency. On average, DTBLI reduces the waiting time by 18.8% from CDPI while DTBL reduce the waiting time by 24.1% and the memory footprint by 25.6% from CDP. Similar to the SMX occupancy behavior, DTBLI of *pre* and *join_uniform* show little change in average waiting time compared because they generate coarse-grained dynamic workloads. The benchmark *regx_string* has the highest DFP occurrence so it benefits most from DTBL by showing the largest waiting time decrease (-41.8%) and significant memory footprint reduction (-51.2%). The benchmark *clr_graph500* does not show much change in the average waiting time when including the launching latency while its SMX occupancy is significantly affected by the launching latency because all the dynamically launched kernels or aggregated

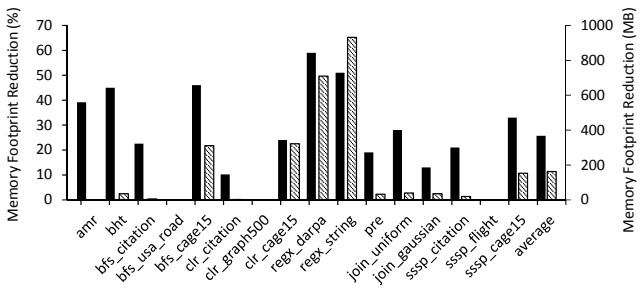

Figure 10: Memory Footprint Reduction of DTBL from CDP

groups are forced to wait for other kernels to complete and release resources before they can be executed, which takes much longer than the launching latency. For the same reason, this benchmark also does not have any memory footprint reduction as the information of all the aggregated groups have to be saved while they are pending. One solution to this problem is to enable the spatial sharing for the native kernels and the aggregated thread groups using the software techniques introduced in [2] or hardware preemption introduced in [35]. This way the aggregated groups are able to execute on the SMX soon after they are generated and the memory reserved for holding their information could be released for new aggregated groups.

C. Overall Performance

We show the overall speed up of CDPI, DTBLI, CDP and DTBL over the flat implementation in Figure 11. Note that data transferring time between CPU and the GPU is excluded. As CDPI and DTBLI decrease control flow divergence and increases memory efficiency, they achieve average 1.43x and 1.63x ideal speedup respectively. However the non-trivial overhead of kernel launching negates the CDP performance gain, which results in an average of 1.16x slow down from the flat implementations. DTBL, on the other hand, shows an average of 1.21x speedup over the flat implementation and 1.40x over the CDP, which demonstrates that DTBL preserves the capability of CDP in increasing control flow and memory regularity for irregular applications while using a more efficient scheduling strategy with lower launching overhead to increase the overall performance. The benchmark *bfs_usa_road* and *sssp_flight* show very little change in the DTBL speedup. The reason is that most of vertices in the input graphs have very low vertex degree. The DFP rarely occurs in these two benchmarks so that very few device kernels or aggregated groups are launched. Therefore, both CDP and DTBL have very limited effect on the overall performance. In fact, these two benchmarks also show limited changes in other characteristics evaluated and discussed previously. Two benchmarks have slow down instead of speedup: *clr_graph500* (0.97x) and *regx_string* (0.95x). The benchmark *clr_graph500* operates on the *graph500* input data set which has a very balanced vertex degree. Therefore, the flat implementation has good control flow and memory behavior. Using CDP or DTBL does

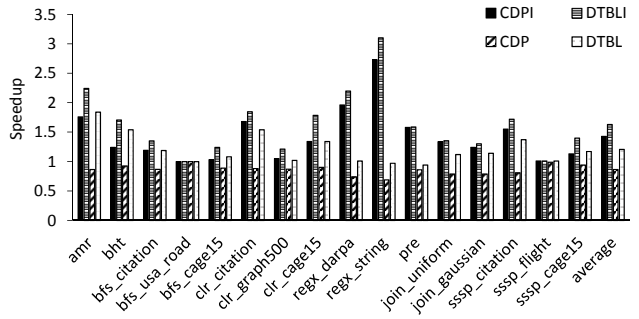

Figure 11: Overall Performance in terms of Speedup over Flat Implementation

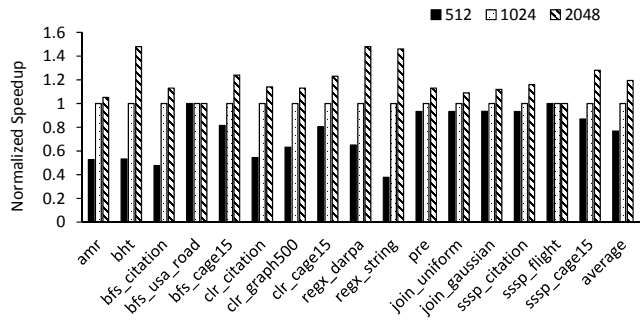

Figure 12: Performance Sensitivity to AGT Size Normalized to 1024 Entries

not help reduce the control flow or memory irregularity but introduces extra launching overhead. For *regx_string*, while the large number of dynamically generated aggregated groups in brings significant speedup ideally (2.73x for CDPI and 3.10x for DTBLI), they also introduce substantial launching overhead even for DTBL and negates the performance gains.

D. Sensitivity to AGT Size

As the major architecture extension, the size of AGT determines the hardware overhead as well as the application performance. A larger AGT can increase the number of aggregate groups stored on-chip and thereby the scheduling efficiency at the cost of more on-chip SRAM. We try to identify the trade off by investigating the performance change over different AGT sizes as shown in Figure 12. In average, decreasing AGT size from 1024 to 512 causes 1.31x slow down and increasing to 2048 causes 1.20x speedup. Benchmarks that use relatively high number of dynamic aggregated groups such as *bht* and *regx* are more sensitive to AGT size.

6. Related Work

Characterization of various benchmarks show that implementations of irregular GPU applications mainly suffer from workload imbalance and scattered memory accesses that cause control flow and memory irregularity [7][9]. Researchers have been investigating and seeking more efficient solutions for these irregular applications by redesigning data structures and re-organizing memory accesses through algorithm, compiler and runtime optimizations [33][40][23][38] to harness the GPU capability.

Increasing performance of irregular applications by handling nested parallelism is an important and challenging question in the GPGPU programming community. Lars et al. [6] implement the NESL language on GPU and Lee et al. [20] propose an auto-tuning framework that efficiently maps nested patterns in GPU applications. Yang et al. [39] propose a compiler technique that can activate or disable threads in runtime to handle nested parallelism in GPU applications. Compared with their work, our methodology is to generate workloads dynamically for any nested parallel patterns.

Gupta et al. [15] introduce the persistent threads programming style on GPUs where enough thread blocks to occupy all the SMX are initially launched and stay on GPU for the life time of the kernel. These thread blocks dynamically generate tasks that are appended to a globally visible software queue while persistently consuming tasks. Steffen et al. [34] propose the idea of dynamic micro-kernel architecture for global rendering algorithm which supports dynamically spawning threads as a new warp to execute a subsection of the parent threads code. Orr et al. [29] design a task aggregation framework on GPU based on the channel abstraction proposed by Gaster et al [14]. Each channel is defined as a finite queue in virtual memory (global memory space that is visible to both CPU and GPU) whose elements are dynamically generated tasks that execute the same kernel function. Our approach is embedded in the basic GPU execution model and can accommodate the preceding optimizations providing uniformity in optimizing all of the aspects addressed in these schemes. Thus, we argues it has a wider range of applicability.

7. Conclusions

In this paper, we propose a new extension to the current GPU execution model that enables dynamic thread block launching and coalescing to existing kernels on the fly. The proposed model is specifically designed to provide a more efficient solution for executing dynamically formed pockets of parallelism in irregular applications. We define the execution model, propose minimal modification to the programming interface and discuss the microarchitecture extension. Through experimental evaluation on various irregular CUDA applications, we demonstrate that by increasing GPU scheduling efficiency and decreasing launching overhead, the proposed model achieves average 1.21x speedup over the original flat implementation and average 1.40x over the implementations using device-kernel launch functionality.

Acknowledgement

This research was supported by the National Science Foundation under grant CCF 1337177 and by an NVIDIA Graduate Fellowship. We would also like to acknowledge the detailed and constructive comments of the reviewers.

References

[1] "Global flight network." [Online]. Available: http://www.visualizing.org/datasets/global-flights-network

[2] J. Adriaens, K. Compton, N. S. Kim, and M. Schulte, "The case for gpgpu spatial multitasking," in *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, 2012.

[3] 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, pp. 5342–5359, 2008.

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

[5] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software(ISPASS 2009)*, April 2009, pp. 163–174.

[6] L. Bergstrom and J. Reppy, "Nested data-parallelism on the gpu," in *ACM SIGPLAN Notices*, vol. 47, no. 9. ACM, 2012, pp. 247–258.

[7] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on gpus," in *Workload Characterization (IISWC), 2012 IEEE International Symposium on*. IEEE, 2012, pp. 141–151.

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

[9] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular gpgpu graph applications," in *Workload Characterization (IISWC), 2013 IEEE International Symposium on*. IEEE, 2013, pp. 185–195.

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

[11] B. W. Coon, J. R. Nickolls, J. E. Lindholm, R. J. Stoll, N. Wang, and J. H. Choquette, "Thread group scheduler for computing on a parallel thread processor," US Patent 8,732,713, 2014.

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

[13] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient gpu control flow," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2007, pp. 407–420.

[14] B. R. Gaster and L. Howes, "Can gpgpu programming be liberated from the data-parallel bottleneck?" *Computer*, vol. 45, no. 8, pp. 42–52, 2012.

[15] K. Gupta, J. A. Stuart, and J. D. Owens, "A study of persistent threads style gpu programming for gpgpu workloads," in *Innovative Parallel Computing (InPar), 2012*. IEEE, 2012, pp. 1–14.

[16] 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.

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

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

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

[20] H. Lee, K. Brown, A. Sujeeth, T. Rompf, and K. Olukotun, "Locality-aware mapping of nested parallel patterns on gpus," in *the 47th International Symposium on Microarchitecture (MICRO '14)*, 2014.

[21] 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.

[22] J. Meng, D. Tarjan, and K. Skadron, "Dynamic warp subdivision for integrated branch and memory divergence tolerance," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, 2010, pp. 235–246.

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

[24] 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*. Eurographics Association, 2005, pp. 105–111.

[25] 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 *Big Data, 2013 IEEE International Conference on*. IEEE, 2013, pp. 175–180.

[26] NVIDIA, "Hyperq sample," 2012.

[27] ——, "Cuda c programming guide version 6.5," 2014.

[28] ——, "Cuda dynamic parallelism programming guide," 2014.

[29] M. S. Orr, B. M. Beckmann, S. K. Reinhardt, and D. A. Wood, "Fine-grain task aggregation and coordination on gpus," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14, 2014.

[30] 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, no. 4. ACM, 2010, p. 66.

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

[32] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012.

[33] S. Solomon and P. Thulasiraman, "Performance study of mapping irregular computations on gpus," in *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–8.

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

[35] 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*, 2014.

[36] J. Wang and S. Yalamanchili, "Characterization and analysis of dynamic parallelism in unstructured gpu applications," in *2014 IEEE International Symposium on Workload Characterization*, October 2014.

[37] 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*. IEEE, 2011, pp. 366–370.

[38] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili, "Kernel weaver: Automatically fusing database primitives for efficient gpu computation," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012.

[39] 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*, 2014.

[40] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen, "On-the-fly elimination of dynamic irregularities for gpu computing," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, 2011, pp. 369–380.