BlockMaestro: Enabling Programmer-Transparent Task-based Execution in GPU Systems

AmirAli Abdolrashidi*, Hodjat Asghari Esfeden*, Ali Jahanshahi*, Kaustubh Singh†*
Nael Abu-Ghazaleh†* and Daniel Wong†
*Department of Computer Science and Engineering
†Department of Electrical and Computer Engineering
University of California, Riverside
Riverside, CA 92521
{aabdo001,hasgh001,ajaha004,ksing057,naelag,danwong}@ucr.edu

***Abstract*—As modern GPU workloads grow in size and complexity, there is an ever-increasing demand for GPU computational power. Emerging workloads contain hundreds or thousands of GPU kernel launches, which incur high overheads, and exhibit data-dependent behavior between kernels, which requires synchronization, leading to GPU under-utilization. Task-based execution models have been proposed to solve these issues, but they require significant programmer effort to port applications to proprietary task-based programming models in order to specify tasks and task dependencies. To address this need, we propose BlockMaestro, a software-hardware solution that combines command queue reordering, kernel-launch-time static analysis, and runtime hardware support to dynamically identify and resolve thread-block level data dependencies between kernels. Through static analysis of memory access patterns at kernel-launch-time, BlockMaestro can extract inter-kernel thread block-level data dependencies. BlockMaestro also introduces kernel pre-launching to reduce the kernel launch overheads experienced by multiple dependent kernels. Correctness is enforced by dynamically resolving thread block-level data dependency at runtime through hardware support. BlockMaestro achieves an average speedup of 51.76% (up to 2.92x) on data-dependent benchmarks, and requires minimal hardware overhead.**

***Index Terms*—GPGPU, SIMD, Data Dependency, Thread Block Scheduling, Just-in-time**

I. Introduction

Graphics processing units (GPUs) today are computationally powerful, power-hungry, and massively parallel devices, capable of processing applications using thousands of threads at once, taking advantage of its single-instruction, multiple-thread (SIMT) paradigm [2], [3], [6], [21], [31], [36], [37], [44], [52], [55], [57], [58]. As modern workloads grow in size and complexity, GPUs are stressed more than ever before [32], [33], [41]. For example, they are one of the main accelerators behind modern machine learning frameworks [1], [14], [45] where typically every layer is encapsulated in a GPU kernel, and the main accelerator behind future exascale computers [16], [17], [50] where scientific computing applications make heavy use of iterative structured grid computations exhibiting wavefront parallelism [9], [10], [29], with multiple GPUs working together through specialized interconnects [40], [48].

These emerging workloads place significant burden on GPUs. By launching hundreds of kernels over the course of an application's execution, kernel launch overheads can become significant [13], [20], [27], [39]. These kernels also typically exhibit significant data dependencies between them [4], [10], [26], [30]. For example, layers in CNNs produce data that is consumed in the next layer. In stencil computations, which are common in scientific computing, operations performed on elements are dependent on the state of neighboring elements. These inter-kernel data dependencies are typically enforced in a coarse-grain manner through implicit barrier synchronizations in the form of kernel launches, which can result in stalling of computation that already have satisfied dependencies.

To circumvent these issues, many task-based execution models and runtimes have been proposed [4], [5], [10], [11], [19], [24], [26], [28], [35], [46], [54], [59]. These frameworks require programmers to decompose the application into *tasks*, and express task dependencies through proprietary programming models (such as AMD ATMI [5], CUDA Graphs [43], OpenMP Tasks [7], etc.), which will then be enforced by the runtime. The main benefit of task-based execution is that (1) kernel launch overhead can be significantly reduced by collectively launching groups of kernels as a whole [5], [43] or by launching a persistent kernel which process tasks that enter its work queue [10]; and (2) dependent tasks can begin executing as soon as their data dependencies are met. However, to gain these benefits, existing GPU applications must be refactored into these proprietary task-based programming models.

In this work, we propose *BlockMaestro*, which provide the benefits of task-based execution using existing SIMT programming models (such as CUDA or AMD HIP) and avoids the need for heavy code modification. The key insight behind BlockMaestro, is that *kernel pre-launching* and *fine-grain inter-kernel data dependency resolution* achieves the benefits of task-based execution models. By pre-launching dependent kernels, we are able to mask kernel launch overheads. To enforce correctness, thread blocks (TBs) of pre-launched dependent kernels are not executed until thread block-level data dependencies are resolved. Inter-kernel thread block data dependencies between neighboring kernels (which we denote as parent and child kernels, K_p and K_c, respectively) can be represented as bipartite graphs as illustrated in Figure 1. The

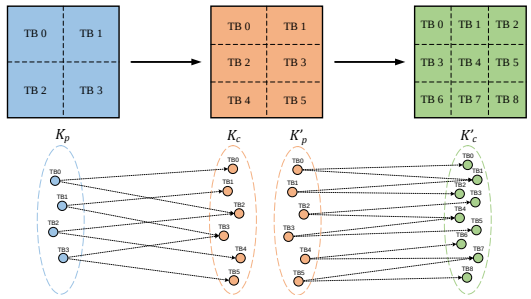


Fig. 1. Data shared by the kernels constitutes dependencies among their TBs, shown as a series of bipartite graphs.

entire GPU application can then be represented as a series of these bipartite graphs, collectively representing a task graph. We present a "Thread Blocks as Tasks" tasking paradigm that leverages the SIMT programming model's property where grids of thread blocks are inherently tasks with explicit input/output through global memory as defined in kernel launch parameters.

Therefore, the key to achieving the benefits of task-based execution is to automatically extract and enforce these bipartite dependency graphs while pre-launching dependent kernels. In short, this paper makes the following contributions:

- We propose kernel pre-launching in order to mask kernel launch overheads of dependent kernels. In addition, we introduce command queue reordering to increase the opportunity for kernel pre-launching.
- We leverage compiler support to extract inter-kernel data dependency of existing GPU applications without the need for programmer intervention. The well-defined structure of GPU applications provided by the SIMT programming model allows us to extract data dependencies in the form of bipartite dependency graphs.
- We propose solutions to resolve fine-grain data dependencies between inter-kernel thread blocks. This ensures the correctness of pre-launched kernels and enables dependent thread blocks to start executing as soon as their data dependencies are satisfied.

In Section II, we provide background and motivate our work. Section III details BlockMaestro. We will then present the results of our evaluation in Section IV. Relevant contributions in literature are discussed in Section V, and the paper is concluded in Section VI.

II. Background

A. GPU Execution

API calls and command queue: In GPU applications, the host calls a series of API functions to interact with the GPU. Common API calls include kernel launch, memory transfer to/from the host, synchronization, etc. All API calls are sent to a command queue (also known as *Stream* in CUDA and HIP terminology) for processing. The API calls (also known as *Events*) are serialized in the command queue with only one event being processed at a time. Therefore, only a single kernel may be executing from a single command queue. To support concurrent execution of *independent* kernels, kernels can be issued to multiple command queues and it is possible to synchronize kernels across command queues through complex synchronization events.

From the host's point-of-view, not all API calls are blocking. By default, memory operations, such as memory allocation and transfer to/from the GPU, are synchronous (blocking), and kernel launches are asynchronous (non-blocking). Therefore, when the host launches a GPU kernel, the host can continue to execute code, but it must explicitly synchronize and wait until the GPU kernel has completed before using the kernel's output. Therefore, the programmer should be aware of any dependencies between the host and the GPU, , including read-after-write (RAW), and ensure proper synchronization.

Compiling GPU programs to assembly: GPU programs are typically written in high-level languages, such as CUDA, OpenCL, or HIP. During the compilation process, these programs are translated into assembly. For example, HIP is compiled into GCN assembly and CUDA is compiled into PTX, a form of intermediate language (IR), which is then just-in-time (JIT) compiled at kernel-launch-time into SASS assembly. Depending on the target GPU and the language used, there is an offline compilation stage (HIP to GCN, CUDA to PTX) and potentially a second just-in-time compilation stage (PTX to SASS). The just-in-time compilation stage enables further optimization because additional parameters at kernel-launch-time, such as thread block size and grid size, are known and can be further optimized.

Kernel launch overheads: Due to the complexity in launching a computation kernel on the GPU, kernel launch overhead is not negligible. Prior works have found that each kernel launch can incur an overhead of $5 - 30\mu s$ [4], [27]. To make matters worse, many GPU applications are also scaling in complexity and size. For example, modern machine learning frameworks that utilizes GPUs for compute-heavy operations (such as convolution) can incur hundreds of kernel calls as ML models grow. Many workloads also require significant synchronization which are implemented implicitly as kernel calls. Towards this end, many prior works have explored how to reduce kernel launch overheads [13], [20], [27], [39]. (See related works section for details of prior works.)

Another common approach to reduce kernel launch overheads is to port programs in the SIMT programming model into a task-based programming model. Task-based runtimes can avoid kernel launch overheads and dynamically resolve data dependencies between tasks, for example, by using persistent kernels to process tasks in the work queue.

B. Task-based execution model paradigms

Many task-based programming models allow programmers to specify series of operations (tasks) and the dependencies between them. Existing task-based programming models can be categorized broadly as following a *"Tasks as Kernels"* or *"Tasks as Thread Blocks"* paradigm. We will detail each

paradigm and discuss the strength and weaknesses of each, and propose a new *"Thread Blocks as Tasks"* approach.

"Tasks as Kernels": In this paradigm, tasks in a task graph are mapped to kernels. For example, AMD ATMI [5] and CUDA Graphs [43] allow users to define kernels and the dependencies between them. To alleviate the effects of kernel launch overheads, these frameworks aim to identify common static operation graphs consisting of many kernels and consolidate the kernel launch into a single task graph launch. While these frameworks can lower the overhead from kernel launches, they fail to take advantage of fine-grain data dependencies that exist between kernels. For example, thread blocks in a dependent kernel may be ready to execute due to satisfied dependencies from thread blocks from the kernel before it, but cannot begin execution until their kernel is launched. Therefore, to handle these *dependency-stalled thread blocks*, finer-grain tasking paradigms are warranted.

"Tasks as Thread Blocks": A finer-grain approach to task-based execution is to map and execute tasks in a task graph as thread blocks. These task graphs can be defined by the programmer using a variety of task-based programming models [4], [10], [11], [24], [59] which defines tasks and their dependencies. At a high-level, these GPU task-based runtimes resolve dependencies between the tasks and then sends ready tasks to a job queue, where they are processed by a persistent kernel. By using a persistent kernel approach, kernel launch overheads are avoided.

This fine-grain dynamic dependency resolution, along with persistent kernel, can reduce the amount of dependency-stalled tasks waiting for execution. However, there are run-time overheads with task management and require significant programmer effort to map algorithms into new task-based programming models. Specifically, it requires the programmer to have domain-specific knowledge of the algorithm and be able to decompose the steps and express it in the form of a task graph.

"Thread Blocks as Tasks": The goal of our work is to enable the benefits of task-based execution models with minimal programmer intervention. At the core of every task-based programming model is the ability to define a task graph for execution. Towards this end, we propose a "Thread Blocks as Tasks" paradigm where, instead of programmers defining tasks which are mapped and executed as thread blocks, we extract and derive tasks and task graphs from the existing thread blocks in the SIMT programming model. We take the view that *grids of thread blocks are essentially tasks with explicit input/output through global memory as specified in kernel launch parameters.* We leverage the properties of multi-kernel GPU applications in order to build a task graph from a series of bipartite dependency graphs (essentially, a decomposed task graph). To alleviate the overhead of kernel launch overheads, we propose pre-launching kernels before their dependencies are met and relying on dynamic data dependency resolution in hardware to enforce dependencies. This effectively removes the kernel launch overhead from the critical path by masking the kernel launch.

Towards this goal, our main challenges in achieving this new tasking paradigm are: (1) How to identify and extract inter-kernel thread block-level data dependency to derive task graphs? (2) How to reduce kernel launch overheads? and (3) How to dynamically resolve task data dependencies in a lightweight manner?

Task partitioning limitations: In order to remap applications into task-based execution, the problem space must be partitioned into tasks. The tasks can be partitioned either statically based on the algorithmic properties of the workload (for example, pipeline stages in image rendering) or dynamically based on the input data. For example, CUDA Graph can record and capture a static task graph of kernels executing across streams. This operation is time-intensive and, in essence, profiles the workload to create a static graph which executes repeatedly. Static task graphs are not well-suited for workloads which are input-dependent. For example, task graphs structure are based on input sparse matrix for sparse solvers.

Typically, input-dependent task graphs require run-time information in order to partition tasks. Due to this, it is difficult to extract task graphs from existing applications in our proposed framework. Thus, the goal of this work is to demonstrate the ability to extract static fine-grain task graphs, similar to CUDA Graph, in order to provide programmer-transparent task-based execution. We leave the ability to handle input-dependent task graphs for future work.

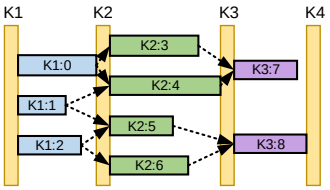
III. BlockMaestro

A. Overview

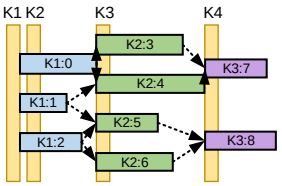
In this section, we present BlockMaestro, which enables programmer-transparent support for the "Thread blocks as Tasks" tasking paradigm on GPUs. BlockMaestro consists of three main components: (1) extracting fine-grain inter-kernel data dependencies from existing GPU applications; (2) kernel pre-launching to mask kernel launch overheads; and (3) dynamic inter-kernel data dependency resolution to ensure correctness of pre-launched kernels. Combined, these techniques allow GPU programs written in existing SIMT programming models to gain the benefits of task-based execution without the overhead of proprietary task-based programming models.

Figure 2 illustrates the operation of BlockMaestro. In this illustrative figure, we show the launching of four kernels, `K1` to `K4`, along with their corresponding thread blocks (labeled `0-2` for `K1`, `3-6` for `K2`, `7-8` for `K3`; blocks omitted for `K4`). The vertical bars represents each kernel's launch overhead. Figure 2a shows the execution timeline of the baseline GPU where kernel execution are serialized. In the baseline scenario, inefficiencies exist due to kernel launch overheads and dependency-stalled thread blocks resulting in GPU under-utilization. For example, even if `K2:5` and `K2:6` have already completed, `K3:8` cannot start until all of `K2` has completed.

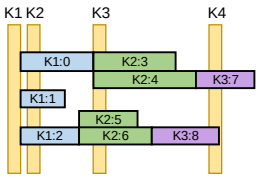
To alleviate these issues, task-based runtimes allow programmers to express task execution by dynamically creating tasks and specifying their dependencies. This allows blocks to begin executing whenever dependencies are satisfied and



(a) Baseline execution model. In BlockMaestro, inter-kernel thread block-level data dependencies are identified and extracted. (`K4` tasks are not shown.)



(b) Kernel pre-launching masks kernel launch overheads. Blocking of dependent thread blocks are enforced by hardware en masse.



(c) Dynamic inter-kernel thread block-level data dependency resolution enables overlapped execution of kernels, achieving benefits of task-based runtimes.

Fig. 2. Baseline execution model suffers from high kernel launch overheads, dependency stalls and resource under-utilization. BlockMaestro's key insight is that kernel launch hiding and inter-kernel data dependency resolution can enable the benefits of task-based runtimes without the programmer burden. Kernel launch overhead is displayed as a vertical bar.

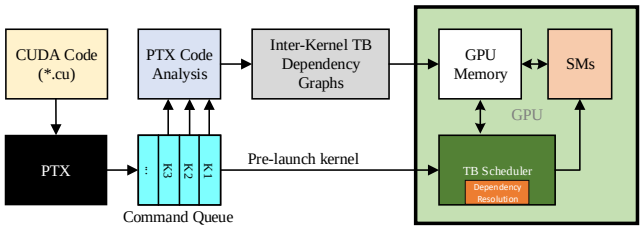


Fig. 3. Overview of BlockMaestro.

can avoid kernel launch overheads with persistent kernels. In task-based runtimes, `K3:8` would be able to execute immediately once `K2:5` and `K2:6` have completed. To achieve the same goals, BlockMaestro introduces *kernel pre-launching* and *inter-kernel data dependency resolution* to eliminate kernel launch overheads and to enable overlapped execution of thread blocks from dependent kernels, respectively.

Figure 2b illustrates kernel pre-launching in order to hide the overhead of kernel launches. After kernel `K1` launches, BlockMaestro will pre-launch kernel `K2`. In order to enforce correctness and resolve data dependencies between `K1` and `K2`, the thread block scheduler conservatively blocks `K2`'s execution until all blocks from `K1` has completed. While kernel launch overheads are eliminated, dependency stalls and under-utilization of resources can still exist.

To fully achieve the benefits of task-based execution, we further identify thread block-level data dependencies that exist between pairs of dependent kernel (annotated with arrows in the figure) at kernel-launch-time where just-in-time compilation occurs from PTX to SASS. Figure 2c illustrates inter-kernel data dependency resolution which utilizes the thread block-level data dependency information between dependent kernel pairs. These data dependencies are enforced by the thread block scheduler at run-time and are dynamically resolved. This enables any ready thread blocks to begin execution, regardless of which kernel they are running in.

Figure 3 shows the system overview of BlockMaestro. Data dependencies between inter-kernel thread blocks are acquired from just-in-time analysis at kernel launch time when PTX code is compiled to SASS assembly. These dependency graphs are then passed in the the hardware, where the dependencies are dynamically resolved. BlockMaestro further eliminates kernel launch overhead by enabling the application to pre-launch dependent kernels at the same time, without the need for synchronization. Once the data dependencies of any thread block from the dependent kernel are met, that TB will also be eligible to be issued to the execution units.

In the remainder of this section, we will discuss how BlockMaestro will enable kernel pre-launching and identify, represent, and enforce inter-kernel data dependencies.

B. Identifying Inter-kernel Dependencies

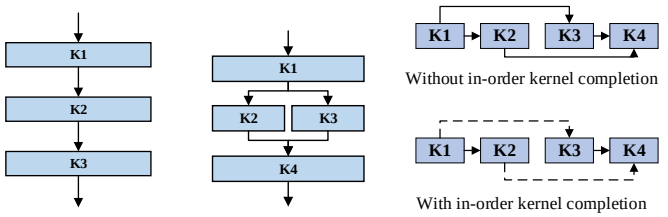
In many task-based runtimes, task dependencies are specified directly by the programmer. Dependencies can be conveyed at high-level task-based programming models such as CUDA Graph [43] or AMD ATMI [5], where programmers define a graph of operations. The key to BlockMaestro providing programmer-transparent support for the "Thread blocks as Task" paradigm is to identify inter-kernel thread block-level data dependencies.

1) Identifying kernel-kernel dependencies:

Data dependencies between kernels occur in data residing in global memory. Due to the SIMT programming model, the inputs and outputs of kernels are well-defined. Every region in global memory used by kernels are allocated with API calls, such as `cudaMalloc` in CUDA or `hipMalloc` in HIP. Load and store addresses can be identified through static analysis of the kernel's PTX or SASS code during the just-in-time compilation phase at kernel launch time.

If a kernel is to read from or write to a region of allocated global memory, the base pointer of the memory allocation must be passed to the kernel launch API. For memory APIs, base pointers are similarly passed. Writes are `cudaMemcpy` host-to-device operations and reads are device-to-host operations. Therefore, data dependencies between kernels and API calls can be identified within the command queue in a fairly straightforward manner.

Handling arbitrary inter-kernel dependencies: BlockMaestro can support both linear and non-linear patterns; examples of which are shown in Figure 4(a)-(b). When issued, these kernel launch commands would be serialized in the command queue. For example, for the application in Figure 4(b), the kernel launch order would be `K1` to `K4`. With multiple kernels being able to run at a time, it's possible that kernels can



(a) Linear dependency pattern (b) Non-linear dependency pattern (c) Dependency tracking required for (b).

Fig. 4. Example types of inter-kernel dependencies and dependency tracking required for correctness. By enforcing in-order kernel completion we significantly reduce the amount of dependency tracking required (solid lines) due to implicit dependencies (dashed lines).

complete out of order. For example, `K2` and `K3` can execute concurrently but `K3` can be shorter and finish before `K2`. Therefore, to ensure correctness for `K4` we would need to keep track of dependency information for both `K2` and `K3` (Figure 4(c)(top)). This would require each dependent child kernel to keep track of dependencies for arbitrary number of parent kernels. *Clearly, this is not scalable to arbitrary inter-kernel dependency patterns.*

To simplify the amount of dependency tracking required, we enforce *in-order kernel completion*. As shown in Figure 4(c)(bottom), even if `K3` finishes, we do not mark it as complete yet or else `K4` will be incorrect. Instead, `K3` will only be marked complete if `K2` is complete. This way, any dependencies of `K4` on kernels prior to `K3` are implicit (dashed lines) and are guaranteed to be satisfied when `K3` is complete. This greatly reduces the amount of dependency tracking required (solid lines) and *limits dependency tracking to consecutive kernels*.

In addition, let us hypothetically assume `K2` completes before `K1`, in-order kernel completion would implicitly enforce the dependency between `K3` and `K1`. Note that if `K1` completes before `K2`, `K3` can begin execution since there's no explicit dependency between `K2` and `K3`, and `K1` is implicitly satisfied if we only allow 2 kernels to concurrently execute. Essentially, *BlockMaestro allows out-of-order execution of kernels, while enforcing in-order completion of kernels.* While we trade-off some potential kernel overlapping opportunities here, we gain the benefit of being able to scalably represent inter-kernel dependency using a series of bipartite graphs between all kernel pairs.

2) Identifying thread block-level dependencies:

While kernel-level data dependency can enable kernel pre-launching (as shown in Figure 2b), it does not realize the full potential of task-based runtimes (as shown in Figure 2c). In order to achieve task-based runtime benefits, BlockMaestro needs to avoid dependency-stalled thread blocks by enforcing thread block-level data dependency. By enforcing inter-kernel data dependency at the granularity of thread blocks, we can overlap the execution of thread blocks from dependent kernels.

BlockMaestro performs just-in-time compiler static analysis (at kernel launch time) to identify read-after-write (RAW) dependencies in global memory. These RAW dependencies are enforced at runtime by the thread block scheduler. The key to identifying RAW dependencies at thread block granularity is to identify the array indexing that each thread block touches.

In the CUDA programming model, programmers already specify the mapping of threads to data by calculating indices deriving from indexing variables such as `threadIdx`, `blockIdx`, `blockDim`, etc. Using a simple vector add as an example, kernel maps threads to an index in the array using `int i = threadIdx.x + blockDim.x * blockIdx.x`. Then, each thread reads in an element in the arrays `A[i]` and `B[i]`, and store the sum into `C[i]`. Arrays `A[]`, `B[]`, and `C[]` are passed into the kernel function after being allocated with `cudaMalloc`. Based on the application's data-flow graph, we can identify all loads and stores in the program to identify the read and write sets, respectively.

To identify thread block-level read and write sets, we identify the indexing used to access the loads and stores by extracting the index representation as a function of parameters known at kernel-launch-time, e.g., `A + 4 * (threadIdx.x + blockDim.x * blockIdx.x)`. Each of these variables are known at kernel-launch-time, along with their possible values. Therefore, we can perform *value range analysis* to identify the range of array indices that each TB will access and create a read and write set per TB.

Value range analysis: We implement and perform this value range analysis for indices in load and store instructions per thread using the built-in PTX parser in GPGPU-Sim [8] as shown in Algorithm 1. Note that this algorithm is general enough that it can also be performed on any compiler frameworks that supports PTX, such as GPUOcelot [22].

Algorithm 1 Psuedo-code of PTX static analysis

```
1: Find all global loads/store instructions in kernel K
2: for all I ∈ instructions do
3:     S = {src(I)}
4:     while S is not empty do
5:         Go to the previous instruction j
6:         if dst(j) ∈ S then
7:             if j is a global load then
8:                 END (Possible non-static dependency)
9:             end if
10:            Remove dst(j) from S
11:            if src(j) is in local register (e.g. not immediate) then
12:                Add src(j) to S
13:            end if
14:        end if
15:        if j is first instruction then
16:            Break
17:        end if
18:    end while
       {Value range analysis}
19:    for all t ∈ Threads do
20:        Add address(I) to load and store sets, L_K and S_K
21:    end for
22: end for
23: Intersect L_K with S_{K-1} to find TB RAW dependencies
24: ...
```

We perform a backward pass on the CFG representation of the kernel and identify all global load and stores and track the origins of their source operands (lines 2-22). If we encounter

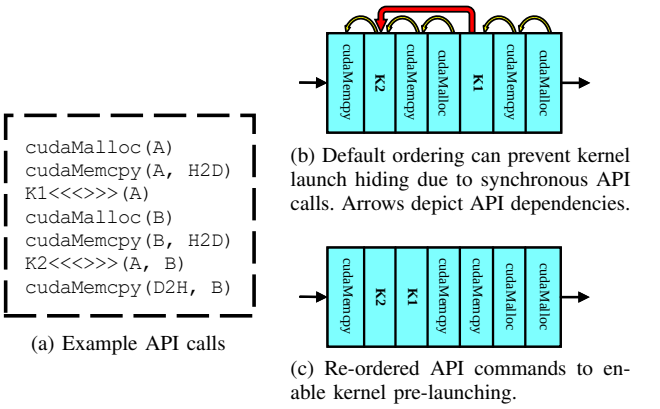
a source operand that originates from the result of another load (an indirect memory access), we terminate and conservatively assume the entire kernel is dependent on the previous kernel (lines 7-9). Otherwise, we know that all load/store source operands are derived from known kernel-launch-time variables. Then given the kernel launch parameters, such as block and grid sizes, we perform the value range analysis to identify the read and write sets of each thread block in the kernel (lines 19-21) and identify dependencies with the intersect of the read and write sets (line 23) of neighboring kernels.

To identify the inter-kernel thread block-level data dependency, every kernel call in the command queue is analyzed to obtain a read and write set. This kernel-launch time analysis overhead is performed off the critical path and is masked by the proposed kernel pre-launching technique. By comparing the read set of later kernels to the write set of earlier kernels, the intersection of the sets will determine where the RAW data dependencies exist. We create a dependency graph to represent data dependency where each node is a thread block and an edge represents a dependency. Since the nodes can be divided into two disjoint and independent sets (each belonging to a separate kernel), our dependency graph is a bipartite graph. This is illustated in Figure 1.

Why JIT analysis and not compile-time? This analysis can only be done at kernel-launch-time during just-in-time compilation from PTX to SASS as certain parameters are only known at kernel-launch time. For example, the grid size is dependent on the input data set. Similarly, `blockDim` (the number of threads in a thread block) and the range of `blockIdx` (range determined by the grid size) are also known at kernel-launch-time. The value range of these variables are unknown at compile-time (from CUDA to PTX) and therefore value range analysis cannot be performed to identify thread block-level data dependency. Furthermore, conducting this at kernel-launch-time from kernel API calls in the command queue allows us to dynamically create bipartite dependency graphs which can allow us to represent larger task graphs in a decomposed manner.

Limitations and other considerations: In this work, we focus on *static memory analysis*, i.e., analysis of memory locations that are known before runtime. They can include device variable addresses, immediate values, and kernel parameters. However, we cannot process global accesses that derive from another memory value (such as `A[B[i]]`), pointer chasing, etc. Such instances are only known at runtime and would require runtime analysis, which is out of scope of this paper.

Note that even if the application makes use of Unified Memory, we can still identify read and write sets through value range analysis. Unified Memory are allocated with `cudaMallocManaged`, so we know which global memory address range needs to be monitored for RAW dependencies. Within the CUDA kernel, memory access occurs in the same manner as non-Unified Memory.



(a) Example API calls

(b) Default ordering can prevent kernel launch hiding due to synchronous API calls. Arrows depict API dependencies.

(c) Re-ordered API commands to enable kernel pre-launching.

Fig. 5. Effect of API ordering in command queue on kernel launch hiding.

C. Enabling Kernel Pre-launching

In this section, we will discuss the changes necessary to enable kernel pre-launching. Overlapping is achieved by having future kernels launching before the completion of the previous kernel. In order to accomplish this, we need to (1) enable multiple kernels to be simultaneously executed from a command queue, and (2) prevent certain CUDA API calls from blocking the command queue or from blocking the issue of future CUDA APIs in order to allow the command queue to fill.

To highlight the challenges in enabling kernel launch hiding, we will refer to Figure 5. In Figure 5a we show an example trace of CUDA API calls. When the host executes the code and reaches a CUDA API call, it sends the call along with the necessary data to a command queue (the default CUDA Stream[1]). Common CUDA API calls are to allocate memory, transfer data to/from GPU global memory and host memory, launch kernels, and device synchronize. Kernel calls are asynchronous (non-blocking) by default. However, CUDA memory API calls are synchronous (i.e., host is blocked until the functions return) and can cause limited opportunities for kernel launch hiding.

Handling blocking APIs: In order to maximize opportunities for kernel pre-launching, we need to be able to have multiple kernel commands in the command queue. However, this scenario can be prevented due to the blocking behavior of certain API calls. In the example shown in Figure 5a, memory operations such as `cudaMalloc` and `cudaMemcpy` are blocking the host. So as kernel `K1` is executing, `cudaMalloc(B)` can be processed in parallel (CUDA commands which use different hardware engines can be executed in parallel), blocking the host until it completes. The host will have to wait until `cudaMalloc(B)` returns before being able to issue `cudaMemcpy(B)` to the command queue. Depending on the length of `K1`, this would prevent the opportunity for `K1` and `K2` to overlap, as `K2` may not be called by the host until after `K1` completes. Therefore, we need to be able to fill multiple

[1]Note that we may use stream and command queue interchangeably. CUDA Streams is equivalent to OpenCL command queue and AMD HIP Streams.

kernel commands in the command queues to maximize kernel pre-launching opportunities.

We can overcome this issue by treating certain blocking operations as non-blocking. Since BlockMaestro can resolve dependencies in the hardware, we can shift the burden of implicit synchronization to the hardware. The only API call requiring implicit synchronization to be enforced is when there is a RAW hazard with the host, e.g., a `cudaMemcpy` call from device to host. Explicit synchronization API calls, such as `cudaDeviceSynchronize`, can also be bypassed as long as no call after it incurs RAW hazard with the host. As long as data is not modified on the host, but only in the GPU, we can enforce correctness of implicit synchronization in the GPU.

Note that asynchronous memory APIs are used by programmers when programming with CUDA Streams. If the target application already utilizes CUDA Streams, then the command queue is already filled and is not an issue. BlockMaestro can also seamlessly support pre-launching in CUDA Stream-based application by overlapping kernel launches within the same stream. The only other consideration is to handle `cudaStreamSyncrhonize` in a similar manner to `cudaDeviceSyncrhonize` for API commands within the same stream. While BlockMaestro can generalize to support CUDA Streams, the remainder of the paper focuses on single default stream applications which experience worse kernel launch overheads and under-utilization issues.

Programmer-transparent API command reordering: Figure 5b shows the state of the command queue after all the CUDA APIs are called. Commands in the queue are implicitly ordered which can cause orderings that limit the amount of kernel launch hiding. For example, as `K1` is launched and executing, we cannot proactively pre-launch `K2` as the `cudaMalloc` and `cudaMemcpy` commands must complete first. One potential solution to maximize kernel launch hiding is to identify the true data dependencies between APIs in the command queue and reorder the commands to maximize kernel launch hiding. This is achieved by moving the kernel launches as close as possible. Figure 5c shows such an order that still satisfies the data dependencies between API calls. Kernels can then be launched if memory could be allocated to them. Otherwise, they will have to wait until resource becomes available.

Enabling multiple kernels to execute from the same command queue: In the baseline, kernel commands are blocking in the command queue. That is, only a single kernel from a command queue can be running at a time. Therefore, one modification that we require is to let the command queue process multiple kernel commands at once. This feature is already available in NVIDIA Hyper-Q which enables multiple kernel commands from different streams (with our modification, from the same stream). In our experience, we find that enabling the execution of only 2-3 kernels per command queue is sufficient to completely overlap kernel launches.

To enforce correctness and resolve data dependency between two running kernels in the same command queue, we rely on the thread block scheduler to enforce the second dependent kernel to only begin executing after it has detected that the first kernel has completed execution. This way, the second kernel's launch overhead overlaps with the first kernel's execution. If the kernels are independent (no data dependency exist between them), then the independent kernel can begin executing right away. Otherwise, data dependencies are enforced by the hardware. Later in this section, we will discuss in detail how the thread block scheduler can enforce inter-kernel data dependencies. Note that all thread blocks in the current kernel can be executed in an out-of-order fashion. However, BlockMaestro enforces the completion of the parent TBs before starting their child TBs from the next kernel.

D. Enforcing Inter-kernel Dependencies

Inter-kernel dependency is enforced using a dependency list representing the bipartite dependency graph. In BlockMaestro, the dependent kernel owns the dependency list. We use this information in the hardware to enforce inter-kernel dependency through the use of a *Dependency List Buffer* and a *Parent Counter Buffer*. We will first illustrate in Figure 6 how BlockMaestro uses these structures to enforce dependencies and then detail architectural support. Recall that we only need to keep track of dependencies between consecutive launched kernels. Thus we illustrate using two kernels and then will discuss how to generalize to support multiple pre-launched kernels.

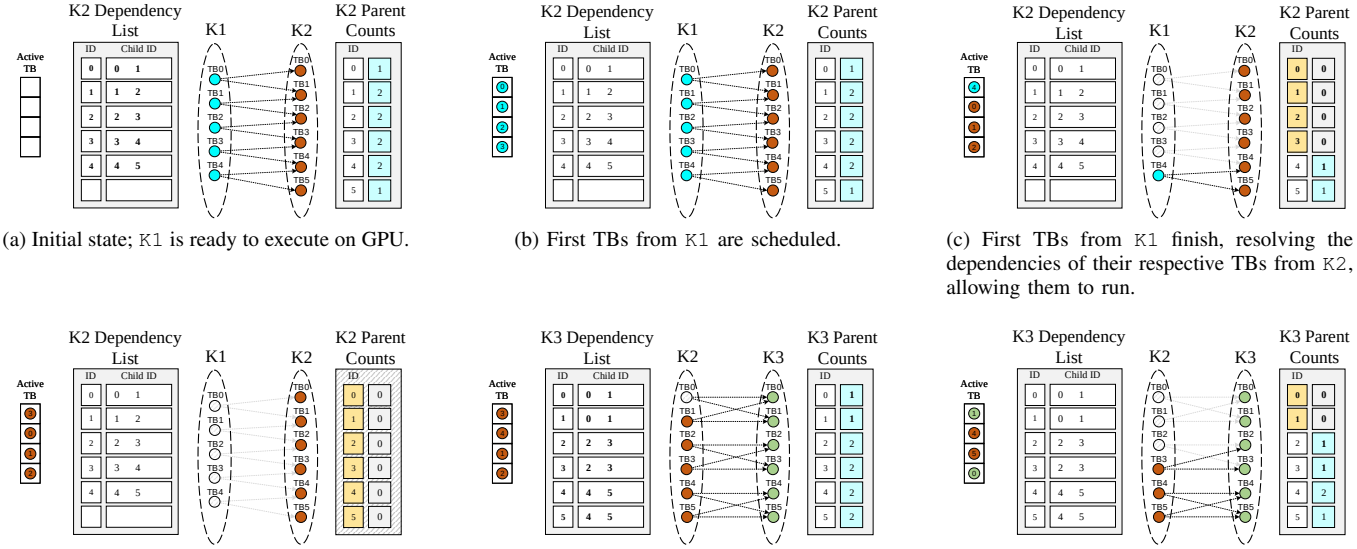
Let us assume we have a GPU that can execute 4 TBs at once. The dependency list stores the bipartite dependency graph which is indexed by the thread block ID of a parent kernel (`K1`) and contains a list of dependent child kernel (`K2`). For example in **(a)**, TB0 of `K1` is a dependee of TB0 and TB1 of `K2`. The parent count table keeps track of how many pending dependencies are outstanding for the child kernel. For example, TB1 of `K2` is dependent on two thread blocks in the parent kernel.

The initial state of the example is shown in **(a)** with `K1` launched and `K2` pre-launched. `K1` is the first kernel, and so it has no dependencies. Thus, the device can start scheduling TBs 0-3 as shown in **(b)**. After TB0 finishes, the remaining TB from `K1` starts. At the same time, children of TB0 are read from the dependency list (TB0 and TB1 from `K2`), and their respective parent counter decrements, making TB0 from `K2` ready to execute once there are available resources in the SM. Soon after, TBs 1-3 from `K1` also finish, allowing TBs 1 and 2 from `K2` to be ready for scheduling **(c)**.

When TB 4 from `K1` finishes **(d)**, `K1` is marked as complete, `K2` is now the designated parent kernel, and we shift our attention to the next pre-launched kernel `K3`. In **(e)**, we now show the dependency list of `K3` which specifies the dependencies in `K2` and the parent counts of pending dependencies for `K3`. As the TBs of `K2` continue executing, we follow the same scheme as in parts **(b)**-**(d)** where `K3` will begin executing **(f)**.

1) Architectural Support:

Figure 7 depicts the proposed supporting hardware for the TB scheduler. When the device receives a kernel from the host, the dependency list and initial parent counters are



(a) Initial state; `K1` is ready to execute on GPU.

(b) First TBs from `K1` are scheduled.

(c) First TBs from `K1` finish, resolving the dependencies of their respective TBs from `K2`, allowing them to run.

(d) `K1` is complete. It can now be replaced by the kernel after `K2`. Other TBs from `K2` can now run. Parent count is invalidated.

(e) `K3` has launched and its dependency list is read from the memory. TB 0 from `K2` has finished. Thus its children's parent counts should be decremented in the TB scheduler.

(f) More TBs from `K2` finish, resolving the dependencies of their child TBs from `K3`. First `K3` TBs begin execution.

Fig. 6. TB scheduling example in BlockMaestro. Inter-kernel thread block-level dependencies are maintained using a dependency list and parent counter.

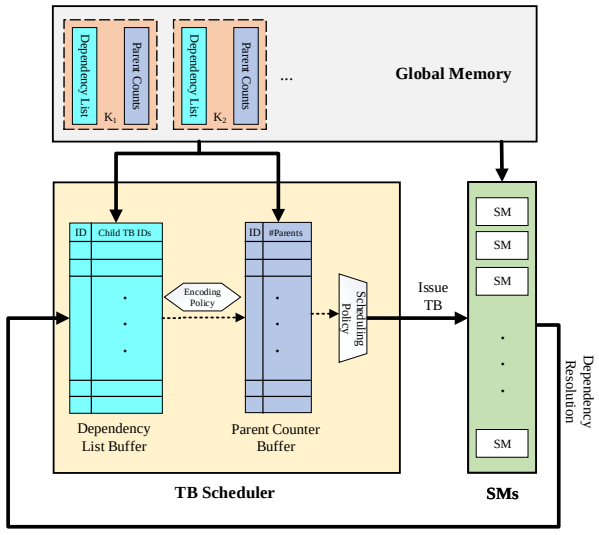


Fig. 7. Supporting TB scheduler architecture.

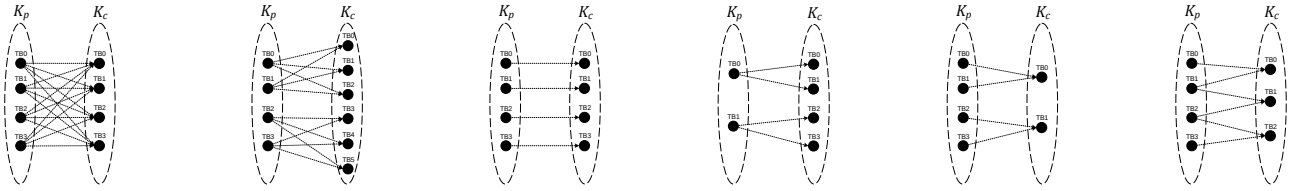
stored in global memory. Thus, for every (pre-)launched kernel the GPU needs to keep track of the *dependency list base address* and *parent counters base address* in global memory. To minimize the amount of global memory access, we include a *Dependency List Buffer* and *Parent Counter Buffer* in the thread block scheduler. The dependency list buffer keeps track of dependencies of actively executing thread blocks and the parent counter buffer keeps track of child thread blocks' pending unresolved dependencies.

When a thread block is scheduled for execution the thread block's entry in the dependency list is buffered in the dependency list buffer. Then the entry is read to identify the child thread blocks. If an entry does not already exist in the parent counter buffer, we allocate an entry and fetch the child thread block's parent counter value. Since the information in this dependency list and parent counter entry is not needed until the thread block finishes execution, this buffering process is off the critical path.

When a TB completes, we identify every child TB ID with the dependency list buffer and index into the parent counter buffer to decrement the parent counts. When a parent count hits 0, the corresponding child TB is now ready for execution. We deallocate an entry in the dependency list buffer when a parent TB completes and we delallocate an entry in the parent counter buffer when that child TB is selected for execution.

Resolving dependencies of multiple kernels: This design is easily scalable to support resolving dependencies of multiple running (pre-)launched kernel execution by simply appending bits to the thread block ID to represent the relative kernel IDs. For example, in order to resolve the dependency of 4 kernels, we can append 2 bits to the thread block ID as a kernel identifier. This kernel identifier is incremented whenever a new kernel is launched and wraps around to 0 when saturated. Since we only need to track dependencies between neighboring kernels, the kernel identifier is essentially the least significant 2 bits of the kernel ID.

Scheduling policies: BlockMaestro can support several scheduling policies across kernel TBs. By default, BlockMaestro gives more priority to the TBs in the producing kernel. TBs in the consuming kernel will not be scheduled until all producing kernel's TBs has been scheduled. It is also possible to give priority to the TBs from the consuming kernel. This will enable more opportunity to concurrently execute dependent kernels and improve utilization by essentially allowing more TBs to "run-ahead".



(a) Fully connected (b) N-group fully connected (c) 1-to-1 (d) 1-to-n (e) n-to-1 (f) Overlapped

Fig. 8. Examples of common dependency patterns between TBs from adjacent kernels

Note that these policies does not face any deadlock issues for any producer kernels under synchronization events. For example, the producing kernel can be deadlocked if some TBs are waiting on a barrier but other TBs cannot be scheduled since the consuming kernel is taking up resources and starving the producer kernel. This scenario does not occur since we will never fully starve a producer kernel to the point of deadlock. In the worse case scenario, eventually the consumer kernel TBs will face unmet dependencies which will allow the producer kernel to schedule, thus avoiding any permanent deadlock.

E. Representing and Storing Inter-kernel Dependencies

We utilize a buffer in the TB scheduler to store the dependency list of the parent kernel. To reduce the amount of storage, we can take advantage of the dependency patterns among the kernels themselves to encode them. These patterns are rarely arbitrary, since the code is usually written to globally load and store the data using a large number of threads, e.g., each thread loading a block. Therefore, by analyzing the pattern, the graph can be stored on the device in an encoded fashion, which can greatly reduce the memory usage.

TABLE I
HARDWARE OVERHEAD W.R.T. DEPENDENCY PATTERN BETWEEN K1 OF SIZE N AND K2 OF SIZE M THREAD BLOCKS.

P#	Pattern	Overhead
(1)	Fully connected	$O(1)$ ($O(MN)$ without encoding)
(2)	n-group fully connected	$O(M+N)$
(3)	1-to-1 ($M=N$)	$O(N)$
(4)	1-to-n	$O(M+N)$
(5)	n-to-1	$O(N)$
(6)	Overlapped	$O(N+M.deg_{max})$
(7)	Independent	$O(1)$

Table I displays the additional memory overhead that BlockMaestro would utilize for a graph with N parent TBs and M child TBs on the hardware. Even though it can be more difficult for a random dependency graph, the overhead can be drastically reduced by detecting specific patterns that can usually occur among the kernels (some shown in Figure 8) and using encoding to reduce the requirements. For example, for a fully connected pattern, a single bit is enough to signal the GPU to simply prevent the consuming kernel from running until the producing kernel is finished. For the n-group case, TBs parenting the same child TBs could be encoded to be grouped together in the memory as well, all TBs in the same group referring to one location containing the child TB group, thus $O(M+N)$. For 1-to-n, every TB from K2 (with M TBs) is mapped to a single parent TB, hence $O(M)$. In 1-to-n, each parent TB has exclusive child TBs, i.e., no child TB is shared between two parents. In the overlapped pattern, parent TBs can share multiple child TBs. Therefore, the overhead will be $O(N)$ plus $O(M)$ times the maximum degree of a child TB. In addition, if the dependency resolution yields little benefits for the execution speedup, e.g., too large a dependency degree, the device can ignore the fine-grained dependency resolution and treat the kernels as if they are fully connected (as it is shown in Figure 12).

TABLE II
LIST OF BENCHMARKS USED, NUMBER OF KERNELS, AND TYPE OF DEPENDENCY PATTERN EXHIBITED (SEE TABLE I).

Name	Description	# Kernels	P#
3MM [25]	3 Matrix Multiplications	3	(2,7)
AlexNet [34]	AlexNet network	22	(1,3,4)
BICG [25]	BiCG Sub Kernel of BiCGStab Linear Solver	2	(7)
FDTD-2D [25]	2D Finite Different Time Domain	24	(5,7)
FFT [18]	Fast Fourier Transform	60	(3,5,7)
GAUSSIAN [12]	Gaussian Elimination	510	(4,5)
GRAMSCHM [25]	Gram-Schmidt Decomposition	192	(1,4,5)
HS [12]	Hotspot	10	(6)
LUD [12]	LU Decomposition	46	(3,4,5)
MVT [25]	Matrix Vector Product and Transpose	2	(7)
NW [12]	Needleman-Wunsch	255	(4,5)
PATH [12]	Path Finder	5	(6)

IV. EVALUATION

A. Methodology and benchmarks

We use a modified version of GPGPU-Sim v3.2.2 [8] as our baseline, with a Titan X Pascal-like configuration with 28 SMs, each able to run up to 32 TBs at once, though BlockMaestro should generalize to any SIMT architecture. Greedy-then-oldest (GTO) warp scheduling policy is used [49]. For kernel launch overhead calculations, we have used the average baseline launch overhead of $5\mu s$ from [27]. As shown in Table II, we evaluate against various applications from Rodinia [12], PolyBench [25], SHOC [18], and Tango [34] benchmark suites, which cover a wide-range of multi-kernel applications from different domains.

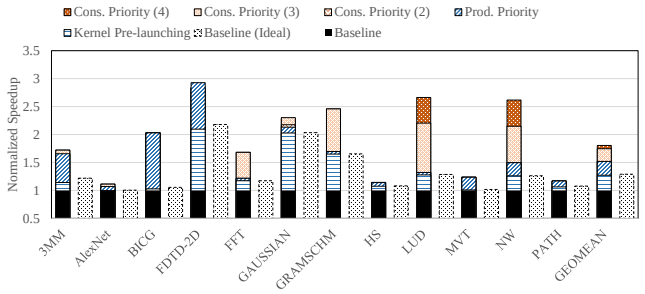


Fig. 9. Normalized speedup w.r.t. baseline.

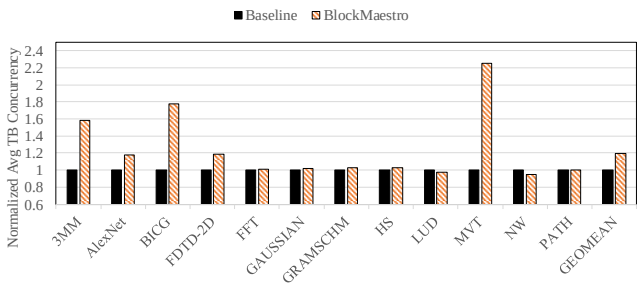


Fig. 10. Normalized average TB concurrency w.r.t. baseline.

B. Results

Speedup: Figure 9 shows the amount of speedup achieved by BlockMaestro with respect to the baseline. For reference, we have included the *ideal baseline* case with no kernel launch overheads (the bar on the right of each stacked bar). *Kernel Pre-launching* uses no synchronization APIs, but enforces the dependency by not allowing any consumer kernel TBs to schedule until all of the producer kernel TBs have completed. In addition, there is *Producer Priority*, which adds the fine-grain dependency resolution and gives scheduling priority to producer kernel's TBs. We have also included *Consumer Priority* which allows 2, 3, and 4 concurrently running kernels corresponding to 1, 2, and 3 pre-launched kernels, respectively. This consumer priority scheme prioritizes the consumer kernel's TB for scheduling.

It can be seen that increasing the number of pre-launched kernels can increases the geometric mean speedup to 80.28%. However, we observe diminishing returns with more than 3 pre-launched kernels. This behavior can be best explained by the degree of TB data dependencies that exist between kernels. Workloads that most benefit from more pre-launched kernels require significant number of kernels in an application and with less connected data dependencies. For example, AlexNet has significant fully-connected dependencies while LUD has only 1-to-1/1-to-n/n-to-1 dependencies which are amenable to TBs running ahead.

Certain applications, such as GAUSSIAN and GRAMSCHM, experience significant speedup from just kernel pre-launching (and no thread block-level dependency resolution). These workloads tend to have large number of kernels, each of which finishes fast. Therefore, kernel launch overhead is the major bottleneck alleviated.

Other benchmarks, such as 3MM, BICG and FDTD, gained most of their benefit from from fine-grain dependency resolution with simple producer priority scheduling. These workloads tend to have data dependencies that are easier to satisfy and captures more benefit with only two kernels active. Sometimes their kernels are independent and able to run in parallel. Therefore more TBs can be ready to execute and advance the application's progress.

Utilization: This increase in utilization can also be seen in Figure 10 which displays the increase in normalized average TB concurrency with respect to the baseline. Workloads which are more compute-intensive with kernels consisting of hundreds or thousands of thread blocks, such as AlexNet, tend to suffer less from the overheads of kernel launches. Therefore, these workloads benefit less from kernel pre-launching alone but still sees an increase in TB concurrency due to fine-grain TB dependency resolution. We still observe that AlexNet achieve speedup of 6.9%. It can be seen that more number of kernels along with simpler dependency patterns can produce more opportunities for overlapped kernel execution and utilizing the resources on the device more efficiently.

Figure 8 showcases various examples of basic patterns in data dependency of TBs from the child kernel K_c on those from its parent kernel K_p, which can be extracted from the PTX code. As the dependency pattern gets more complicated, it becomes more difficult to take advantage of. The fully connected pattern in Figure 8a is the worst-case scenario and is functionally the same as a synchronization barrier between the kernels. Therefore, the opportunity to speed up the application via execution overlapping ends with kernel launch overhead hiding. However, simpler patterns offer a greater opportunity, since after the execution of each TB in the producing kernel, TBs from the consuming kernel become ready for execution, which means more efficient utilization of the device.

Dependency Stall Distribution: Figure 11 displays a distribution of the amount of dependency stalling each TB in an application is going through during the execution. Recall that dependency stalled thread blocks are dependent thread blocks that has dependencies that are satisfied but cannot execute yet due to its kernel not yet started. The box plot borders designate the first and third quartiles of the distribution, with the line in the box representing the median. In addition, the values are normalized to each TB's execution time. For example, a value of 2 for a TB means that that TB has waited for double the amount of time it would spend executing on the GPU. As we can see, BlockMaestro can visibly decrease the amount of dependency stalling for most of the TBs in the applications. However, in some cases where the GPU capacity for TB execution is full, some of the remaining TBs in a dependent kernel will have to wait more than their peers to be run, increasing their stalling, as is shown in the case of AlexNet. Also, note that the two kernels in BICG and MVT can run in parallel in BlockMaestro, hence their dramatic stall reduction. These workloads are also reflective of CUDA

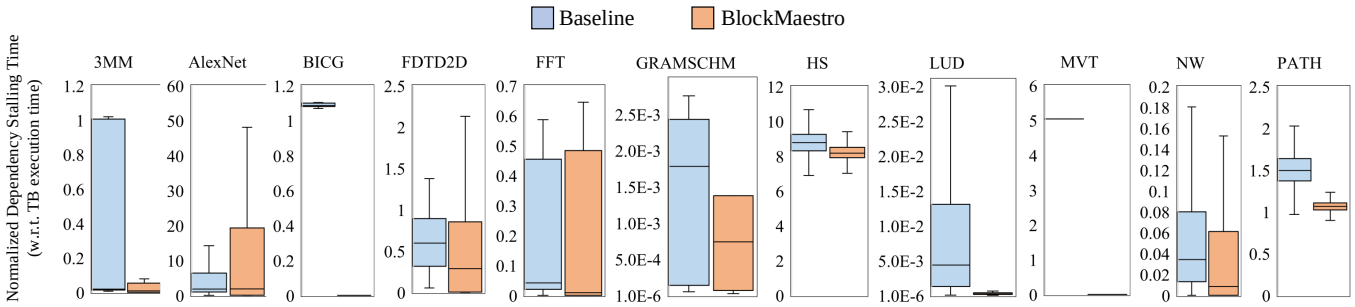


Fig. 11. Dependency stall distribution normalized to TB execution time.

Sterams benefits since independent kernels that can concurrently execute exists. However, CUDA Streams will not be able to be used with concurrently executing non-independent kernels. These results demonstrate that BlockMaestro can gain the benefit of executing independent concurrent kernels across streams automatically, while also extracting benefits for more complex dependency patterns.

C. Overheads

Inter-connectivity Analysis: In Figure 12, we demonstrate the effect of the degree of dependency that exist between inter-kernel thread blocks and kernel size on the speedup of a microbenchmark based on *VectorAdd* with two equal-size kernels. In this application, there is a simple 1-to-1 dependency pattern between the two kernels by default. Each line represents the workload size (number of TBs in one kernel). During each workload, we increase each TB's dependency degree by artificially introducing dependencies between the kernels in the form of an n-group fully connected pattern. For example, a degree of 4 signifies that the first 4 TBs from `K1` are dependent on the first 4 TBs from `K2`, etc., resulting in a 4-to-1 dependency pattern.

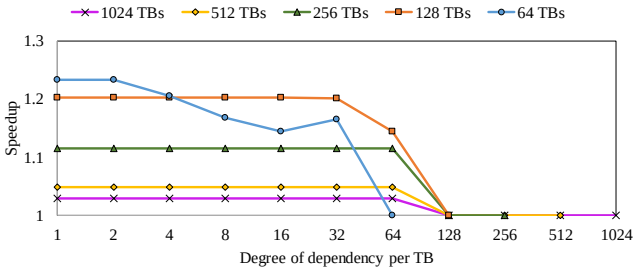


Fig. 12. Interconnectivity analysis for BlockMaestro. The x-axis shows the size of each TB's dependency group.

It can be seen that the benefits we can get from dependency resolution begin to quickly deteriorate once the average dependency degree passes a certain threshold, in our case $deg = 32$. After this point, the speedup benefits reflect that of a fully-connected dependency graph.

In addition, the speedup we get even before this threshold decreases as the number of TBs in the kernels grow and ceases to exist by the time the workload size is 2048. With more TBs running in a kernel, the most resources a kernel require and limits the opportunity for pre-launched kernels to run-ahead. We leverage our insights on how the inter-connectivity of the dependency graph to minimize hardware overheads.

Area overhead: BlockMaestro mainly introduces a dependency list buffer and a parent counter buffer. Since the dependency list buffers actively running TBs, we require $28 \times 32 = 896$ entries. We similarly set the number of parent count buffer entries to the same. For each dependency list buffer entry, we choose to store 4 child TBs per entry. We aggressively choose a narrower entry since most workloads can be described by a dependency pattern. Thus, the encoding can derive child TB IDs. For the rarer scenario where we cannot encode, we will utilize the 4 child TBs per entry. If we require a wider entry (as it exist in global memory) we can simply split the wider entry across multiple entries in the dependency list buffer.

Each index into the dependency list buffer and parent counter buffer (representing the TB ID) is 32 bits + 2 bits for kernel identification. Each child TB ID in the dependency list buffer is 32 bits since kernel identification can be computed. Since we see diminishing return with greater levels of inter-connectivity (greater than 64-to-1), we use 6 bits for the parent counter. Anything higher and we conservatively encode to fully connected without much loss to speedup. In total, we require a storage overhead of about 22KB, in addition to control logic.

Memory Request Overhead: Figure 13 shows the impact of BlockMaestro on the memory requests. Buffering the dependent list information from the memory can incur a request overhead in the order of $O(V)$. As it is seen, BlockMaestro's average memory request overhead is only about 1.36%.

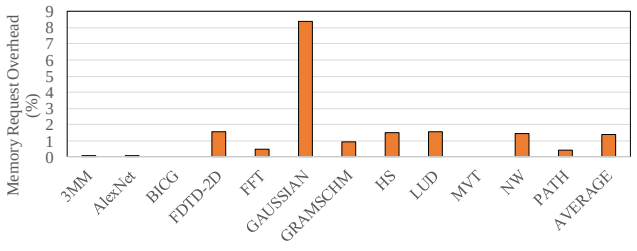


Fig. 13. Memory request overhead for BlockMaestro.

Bipartite Dependency Graph Storage Overhead: Table III displays the amount of storage used for the entire run of each application normalized with respect to the case where no encoding is used, i.e., plain storage. As it is observed, the average storage is reduced by 34.7%. (Note that BICG and MVT are excluded here since their kernels are independent and therefore there is no memory storage used for them even without encoding.)

TABLE III
NORMALIZED TOTAL STORAGE OF BIPARTITE DEPENDENCY GRAPHS FOR THE ENTIRE APPLICATION RUN W.R.T. PLAIN STORAGE.

	Storage		**Storage**
3MM	0.210	AlexNet	0.012
BICG	-	FDTD-2D	1
FFT	1	GAUSSIAN	1.77E-04
GRAMSCHM	0.375	HS	1
LUD	0.938	MVT	-
NW	1	PATH	1
Average	**0.653**		

D. Comparative Results

Comparison to Task-based Execution Models and Dynamic Parallelism: Figure 14 showcases a comparison with CUDA Dynamic Parallelism (CDP) [42], a "Tasks as Kernels" execution model, and Wireframe [4], a "Tasks as TBs" execution model. Wireframe requires the programmer to specify task dependencies using a proprietary API and relies on hardware dependency resolution. Essentially, Wireframe represents multi-kernel workloads into a single mega-kernel with tasks mapped to a TB. CDP represents each task as a device-side kernel launch, avoiding much of the overhead of host-side kernel launches. For a direct comparison with prior works, we have used the benchmarks in [4]; six applications with wavefront dependency pattern of 4K tasks. In other words, each kernel has an overlapped dependency pattern with its predecessor, and the number of TBs gradually grows until the middle of the dependency graph, where it starts to decline.

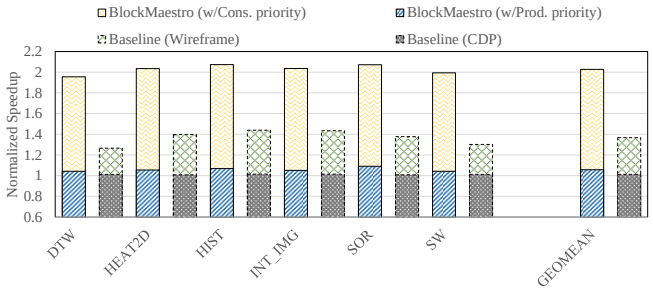


Fig. 14. Comparison with existing "Task as Kernel" (CDP) and "Task as TBs" (Wireframe [4]) task-based execution models.

The widely used CDP kernel launch latency model [53] is based on Kepler and estimates a CDP kernel launch overhead of 20μs, significantly greater than modern host-side kernel launch times (5μs) [27]. Therefore, we model CDP's kernel launch latency as 3μs by removing the kernel launch API call overhead (2μs) [27] from the host-side kernel launch time.

Figure 14 shows the normalized speedup of our comparison normalized to CDP. BlockMaestro with producer priority achieves only 5.8% speedup. Wireframe achieves a geomean speedup of 36.8% due to its ability for tasks to run-ahead up to three waves (levels of dependencies). This enables more tasks to run and utilize the GPU. To that end, we evaluate BlockMaestro with consumer priority to enable tasks to run-ahead. With this, we observe speedup of 2x. We found that Wireframe's reliance on size-constrained hardware task management buffers (i.e., pending update buffers) can actually limit the amount of tasks that can run. Since BlockMaestro's dependency resolution can update task states stored in global memory, our execution is not constrained to increase GPU utilization, but at the cost of slightly higher memory traffic as shown in Figure 13. This successfully demonstrates the benefits that BlockMaestro is able to achieve the benefits of task-based execution models without programmer intervention.

V. RELATED WORK

Task Dependency: CUDA Dynamic Parallelism [42] enables device-side kernel launches to support dynamic kernel launches. This amortizes kernel launch overheads and allows tasks to dynamically spawn on the GPU. However, there are significant drawbacks such as limited levels of recursion [4]. Since CUDA 10, CUDA Graphs [43] allows the user to define a dependency graph between different kernels, perform optimizations on the whole graph during its instantiation, and execute it many times. CUDA Graphs can reduce the kernel launch overhead. However, it still does not address the GPU under-utilization during the execution of dependent kernels.

AMD has also added support the expression of dependencies among GPU "task groups" for years [5]. The authors in [46] use an asynchronous task-based paradigm to express three well-known applications as directed acyclic graphs (DAG) [15] on the Heterogeneous System Architecture (HSA) [23]. In [47], the same authors note the problem of queue oversubscription from parallel tasks, and propose a mechanism to prioritize the critical path in the task graph. In our work, the priority is to finish the TBs from the producer kernel first in a step to potentially add more consumer kernel TBs to the ready TB pool. Adaptive Task Aggregation (ATA) [26] has been proposed as a software solution to reduce the overhead of irregular applications, specifically sparse solvers, through fine-grain task scheduling. The tasks can be assigned to a compute unit even before their parent tasks are complete, hence avoiding the launch overhead.

Task Scheduling: There have been various works on GPU task scheduling. The authors in [4] introduce Wireframe, a hybrid solution for data-dependent workloads which handles TB scheduling in the hardware and eliminates the need for in-kernel synchronizations. However, there is significant programmer burden, and it is designed with single-kernel applications in mind. Juggler [10] employs a software-based

runtime using persistent threads (PT) for single-kernel GPU workloads with data dependencies, trading synchronizations with scheduling through a DAG. The authors in [38] propose overlapped kernel execution through a modified host code paradigm, obtaining the memory access information using compiler-generated profiler kernels and storing them on the GPU's reference count table a TB scheduler with the goal of maximizing parallelism. In [30], the authors seek to overcome the memory bottleneck in GPU applications by proposing a reuse-aware thread block scheduler to exploit data reuse between the kernels with producer-consumer data dependencies in mind, the majority being dependencies between TBs with the same ID from the two kernels ("self-dependencies"), as well as using work stealing to minimize load imbalance. PAVER [51] presents a hybrid TB scheduling method by measuring data locality among each kernel's TBs, scheduling them based on a heuristic method to reduce cache thrashing, and performing task stealing to reduce load imbalance towards the end of each kernel. Note that our work does not target load imbalance directly. However, by enabling and managing the scheduling of TBs from an additional kernel, the SMs have more TBs to run, reducing under-utilization.

VI. Conclusion

In this work, we have proposed BlockMaestro, a software-hardware solution in order to hide the effect of kernel launch overheads as much as possible and manage the execution of thread blocks in a more fine-grained manner by tracking their data dependencies in hardware in order to enforce correctness. Our solution also increases GPU utilization during a GPU kernel execution while incurring a small memory overhead. By using this paradigm, we have observed an average speedup of 51.76% (up to 2.92x) in various applications.

Acknowledgments

This work is partly supported by National Science Foundation under Grants CCF-1815643, CNS-1955650 and CNS-2047521. We would like to thank the anonymous reviewers for their invaluable comments and suggestions. We also extend our gratitude to Qiumin Xu for providing their implementation of Warped-Slicer [56] which was utilized as our baseline for concurrent kernel execution.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283.

[2] M. Abdel-Majeed, D. Wong, and M. Annavaram, "Warped gates: Gating aware scheduling and power gating for gpgpus," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 111–122.

[3] M. Abdel-Majeed, D. Wong, J. Kuang, and M. Annavaram, "Origami: Folding warps for energy efficient gpus," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2925426.2926281

[4] A. Abdolrashidi, D. Tripathy, M. E. Belviranli, L. N. Bhuyan, and D. Wong, "Wireframe: Supporting data-dependent parallelism through dependency graph execution in gpus," in *50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 50)*, 2017, pp. 600–611.

[5] AMD, "Atmi (asynchronous task and memory interface)," https://github.com/RadeonOpenCompute/atmi, 2016, accessed: 2020-06-11.

[6] H. Asghari Esfeden, F. Khorasani, H. Jeon, D. Wong, and N. Abu-Ghazaleh, "Corf: Coalescing operand register file for gpus," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2019, pp. 701–714.

[7] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The design of openmp tasks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 3, pp. 404–418, 2009.

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

[9] M. E. Belviranli, P. Deng, L. N. Bhuyan, R. Gupta, and Q. Zhu, "Peerwave: Exploiting wavefront parallelism on gpus with peer-sm synchronization," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015, pp. 25–35.

[10] M. E. Belviranli, S. Lee, J. S. Vetter, and L. N. Bhuyan, "Juggler: a dependence-aware task-based execution framework for gpus," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2018, pp. 54–67.

[11] S. Chatterjee, M. Grossman, A. Sbîrlea, and V. Sarkar, "Dynamic task parallelism with a gpu work-stealing runtime system," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2011, pp. 203–217.

[12] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. Ieee, 2009, pp. 44–54.

[13] G. Chen and X. Shen, "Free launch: optimizing gpu dynamic kernel launches through thread reuse," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 407–419.

[14] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[15] N. Christofides, *Graph theory: An algorithmic approach (Computer science and applied mathematics)*. Academic Press, Inc., 1975.

[16] Cray, "El capitan," https://www.cray.com/company/customers/lawrence-livermore-national-lab, 2020, accessed: 2020-04-14.

[17] Cray, "Frontier," https://www.cray.com/company/customers/oak-ridge-national-laboratory, 2020, accessed: 2020-04-14.

[18] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (shoc) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, 2010, pp. 63–74.

[19] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[20] I. El Hajj, J. Gómez-Luna, C. Li, L.-W. Chang, D. Milojicic, and W.-m. Hwu, "Klap: Kernel launch aggregation and promotion for optimizing dynamic parallelism," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[21] H. A. Esfeden, A. Abdolrashidi, S. Rahman, D. Wong, and N. Abu-Ghazaleh, "Bow: Breathing operand windows to exploit bypassing in gpus," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 996–1008.

[22] N. Farooqui, A. Kerr, G. Diamos, S. Yalamanchili, and K. Schwan, "A framework for dynamically instrumenting gpu compute applications within gpu ocelot," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, 2011, pp. 1–9.

[23] H. Foundation, "Hsa platform system architecture specification 1.0 (jan 2015)," http://www.hsafoundation.com/standards/, 2015, accessed: 2020-08-20.

[24] T. Gautier, J. V. Lima, N. Maillard, and B. Raffin, "Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 2013, pp. 1299–1308.

[25] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to gpu codes," in *2012 Innovative Parallel Computing (InPar)*. Ieee, 2012, pp. 1–10.

[26] A. E. Helal, A. M. Aji, M. L. Chu, B. M. Beckmann, and W.-c. Feng, "Adaptive task aggregation for high-performance sparse solvers on gpus," in *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2019, pp. 324–336.

[27] T. H. Hetherington, M. Lubeznov, D. Shah, and T. M. Aamodt, "Edge: Event-driven gpu execution," in *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2019, pp. 337–353.

[28] R. D. Hornung and J. A. Keasler, "The raja portability layer: overview and status," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2014.

[29] K. Hou, H. Wang, W.-c. Feng, J. S. Vetter, and S. Lee, "Highly efficient compensation-based parallelism for wavefront loops on gpus," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pp. 276–285.

[30] M. Huzaifa, J. Alsop, A. Mahmoud, G. Salvador, M. D. Sinclair, and S. V. Adve, "Inter-kernel reuse-aware thread block scheduling," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 17, no. 3, pp. 1–27, 2020.

[31] H. Jeon, H. A. Esfeden, N. B. Abu-Ghazaleh, D. Wong, and S. Elango, "Locality-aware gpu register file," *IEEE Computer Architecture Letters*, vol. 18, no. 2, pp. 153–156, 2019.

[32] S. Karimi-Bidhendi, A. Arafati, A. L. Cheng, Y. Wu, A. Kheradvar, and H. Jafarkhani, "Fully-automated deep-learning segmentation of pediatric cardiovascular magnetic resonance of patients with complex congenital heart diseases," *Journal of Cardiovascular Magnetic Resonance*, vol. 22, no. 1, pp. 1–24, 2020.

[33] S. Karimi-Bidhendi, F. Munshi, and A. Munshi, "Scalable classification of univariate and multivariate time series," in *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2018, pp. 1598–1605.

[34] A. Karki, C. P. Keshava, S. M. Shivakumar, J. Skow, G. M. Hegde, and H. Jeon, "Tango: A deep neural network benchmark suite for various accelerators," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE Press, 2019.

[35] A. M. Kaushik, A. M. Aji, M. A. Hassaan, N. Chalmers, N. Wolfe, S. Moe, S. Puthoor, and B. M. Beckmann, "Optimizing hyperplane sweep operations using asynchronous multi-grain gpu tasks," in *2019 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2019, pp. 59–69.

[36] F. Khorasani, H. A. Esfeden, N. Abu-Ghazaleh, and V. Sarkar, "In-register parameter caching for dynamic neural nets with virtual persistent processor specialization," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 377–389.

[37] F. Khorasani, H. A. Esfeden, A. Farmahini-Farahani, N. Jayasena, and V. Sarkar, "Regmutex: Inter-warp gpu register time-sharing," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 816–828.

[38] G. Kim, J. Jeong, J. Kim, and M. Stephenson, "Automatically exploiting implicit pipeline parallelism from multiple dependent kernels for GPUs," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, ser. PACT '16, 2016.

[39] M. LeBeane, B. Potter, A. Pan, A. Dutu, V. Agarwala, W. Lee, D. Majeti, B. Ghimire, E. Van Tassell, S. Wasmundt *et al.*, "Extended task queuing: Active messages for heterogeneous systems," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 933–944.

[40] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker, "Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, pp. 94–110, 2020.

[41] S. Minaee, A. Abdolrashidi, H. Su, M. Bennamoun, and D. Zhang, "Biometric recognition using deep learning: A survey," *arXiv preprint arXiv:1912.00271*, 2019.

[42] NVIDIA, "Dynamic parallelism in cuda," http://developer.download.nvidia.com/assets/cuda/docs/TechBrief_Dynamic_Parallelism_in_CUDA_v2.pdf, 2012.

[43] NVIDIA, "Getting started with cuda graphs," https://devblogs.nvidia.com/cuda-graphs/, 2018, accessed: 2019-11-17.

[44] NVIDIA, "Nvidia turing gpu architecture," https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf, 2018, accessed: 2019-11-13.

[45] A. Paszke, S. Gross, S. Chintala, and G. Chanan, "Pytorch," *Computer software. Vers. 0.3*, vol. 1, 2017.

[46] S. Puthoor, A. M. Aji, S. Che, M. Daga, W. Wu, B. M. Beckmann, and G. Rodgers, "Implementing directed acyclic graphs with the heterogeneous system architecture," in *Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit*, 2016, pp. 53–62.

[47] S. Puthoor, X. Tang, J. Gross, and B. M. Beckmann, "Oversubscribed command queues in gpus," in *Proceedings of the 11th Workshop on General Purpose GPUs*, 2018, pp. 50–60.

[48] K. Ranganath, A. Abdolrashidi, S. L. Song, and D. Wong, "Speeding up collective communications through inter-gpu re-routing," *IEEE Computer Architecture Letters*, vol. 18, no. 2, pp. 128–131, 2019.

[49] 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*. IEEE Computer Society, 2012, pp. 72–83.

[50] R. Stevens, J. Ramprakash, P. Messina, M. Papka, and K. Riley, "Aurora: Argonne's next-generation exascale supercomputer," ANL (Argonne National Laboratory (ANL), Argonne, IL (United States)), Tech. Rep., 2019.

[51] D. Tripathy, A. Abdolrashidi, L. N. Bhuyan, L. Zhou, and D. Wong, "Paver: Locality graph-based thread block scheduling for gpus," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 18, no. 3, 2021. [Online]. Available: http://dx.doi.org/10.1145/3451164.

[52] D. Tripathy, H. Zamani, D. Sahoo, L. N. Bhuyan, and M. Satpathy, "Slumber: static-power management for gpgpu register files," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, 2020, pp. 109–114.

[53] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili, "Dynamic thread block launch: A lightweight execution mechanism to support irregular applications on gpus," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3S, pp. 528–540, 2015.

[54] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili, "Laperm: Locality aware scheduler for dynamic parallelism on gpus," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 583–595.

[55] D. Wong, N. S. Kim, and M. Annavaram, "Approximating warps with intra-warp operand value similarity," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 176–187.

[56] Q. Xu, H. Jeon, K. Kim, W. W. Ro, and M. Annavaram, "Warped-slicer: efficient intra-sm slicing through dynamic resource partitioning for gpu multiprogramming," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 230–242.

[57] H. Zamani, Y. Liu, D. Tripathy, L. Bhuyan, and Z. Chen, "Greenmm: energy efficient gpu matrix multiplication through undervolting," in *Proceedings of the ACM International Conference on Supercomputing*, 2019, pp. 308–318.

[58] H. Zamani, D. Tripathy, L. Bhuyan, and Z. Chen, "Saou: safe adaptive overclocking and undervolting for energy-efficient gpu computing," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, 2020, pp. 205–210.

[59] Z. Zheng, C. Oh, J. Zhai, X. Shen, Y. Yi, and W. Chen, "Versapipe: a versatile programming framework for pipelined computing on gpu," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2017, pp. 587–599.