WIREFRAME: Supporting Data-dependent Parallelism through Dependency Graph Execution in GPUs

AmirAli Abdolrashidi
University of California, Riverside
Riverside, CA
amirali.abdolrashidi@email.ucr.edu

Devashree Tripathy
University of California, Riverside
Riverside, CA
dtrip003@ucr.edu

Mehmet Esat Belviranli*
Oak Ridge National Laboratories
Oak Ridge, TN
belviranlime@ornl.gov

Laxmi Narayan Bhuyan
University of California, Riverside
Riverside, CA
bhuyan@cs.ucr.edu

Daniel Wong
University of California, Riverside
Riverside, CA
dwong@ece.ucr.edu

ABSTRACT

GPUs lack fundamental support for data-dependent parallelism and synchronization. While CUDA Dynamic Parallelism signals progress in this direction, many limitations and challenges still remain. This paper introduces *Wireframe*, a hardware-software solution that enables generalized support for data-dependent parallelism and synchronization. Wireframe enables applications to naturally express execution dependencies across different thread blocks through a *dependency graph* abstraction at run-time, which is sent to the GPU hardware at kernel launch. At run-time, the hardware enforces the dependencies specified in the dependency graph through a dependency-aware thread block scheduler. Overall, Wireframe is able to improve total execution time up to 65.20% with an average of 45.07%.

CCS CONCEPTS

• **Computer systems organization** → **Single instruction, multiple data**; Data flow architectures;

KEYWORDS

GPGPU, SIMD, Data Dependency, Thread Block Scheduling, Dataflow

ACM Reference format:
AmirAli Abdolrashidi, Devashree Tripathy, Mehmet Esat Belviranli, Laxmi Narayan Bhuyan, and Daniel Wong. 2017. WIREFRAME: Supporting Data-dependent Parallelism through Dependency Graph Execution in GPUs. In *Proceedings of MICRO-50, Cambridge, MA, USA, October 14–18, 2017,* 12 pages.
https://doi.org/10.1145/3123939.3123976

*Work performed during Ph.D. studies at University of California, Riverside

1 INTRODUCTION

GPUs have played a remarkable role in the evolution of scientific computing in the last decade. The massive parallelism offered by thousands of compute cores has led developers to redesign traditional CPU applications to run on the massively parallel hardware. Despite the rapid adaptation of GPGPU computing with an enlarging number of application classes, the GPU hardware has failed to evolve fast enough to account for the increasing complexity of such applications.

A major deficiency in the modern CUDA programming paradigm is a lack of fine-grain support for data-dependent parallelism and synchronization. Typically data dependencies require algorithms to be redesigned and mapped to intra-SM barriers (using `__syncthreads()`) or global barriers via implicit synchronization through consecutive kernel launches. This causes difficulty in programming GPGPUs due to mapping algorithms to these constraints, and more importantly, is responsible for significant inefficiencies in the hardware due to load imbalance and resource underutilization [11]. Recent studies [7, 37] have shown that, SMs can remain under-utilized and unnecessarily idle as the execution reaches near global barriers, even though there are TBs whose dependencies are already satisfied.

An intermediate level of inter-block synchronization can ease programmer burden by granting programmers flexibility to convey data-dependent synchronization at the thread block (TB) level. Unfortunately, existing GPGPU software and hardware assume that the TBs (in CUDA), or workgroups (in OpenCL), in a given kernel can be executed in any order, since there is no native support for synchronization between TBs.

Prior work [19, 36] has shown that it is possible to implement limited inter-TB synchronization in software via *persistent threads (PT)*. In this approach, the kernels are redesigned to run with limited number of TBs, whose total count is equal to the number of SMs. The threads in different TBs synchronize via global memory-based software barriers as they iterate through the data indices. However, the PT approach may cause deadlocks due to potentially unscheduled TBs and also may increase global memory access contention if the inter-TB synchronization is frequent.

In a step towards supporting data-dependent parallelism, CUDA dynamic parallelism (CDP) was introduced to support nested parallelism [1]. CDP enables parent kernels to launch child kernels, and then optionally synchronize on the completion of the latter.

CDP is mainly limited to certain application patterns with recursive nested parallelism and time-varying data-dependent nested parallelism, such as loops [34]. Moreover, CDP introduces additional kernel launch overhead due to in-memory context switching, and also significant effort is required for programmers to efficiently map workloads to dynamic parallelism kernels [13].

Prior work have proposed to avoid the overhead of kernel launches in CDP, by instead launching thread blocks in hardware [28, 33, 34], supporting nested parallelism for loops through code transformation [38] or consolidating kernel launch overheads [10, 13]. In the most recent CUDA version (9), Cooperative Threads (CUDA-CT) were introduced to enable explicit synchronization between threads within and across thread blocks, which enables an efficient implementation for global barriers [3]. Although CUDA-CT will partially remedy the problems caused by device-level kernel launches, the SM under-utilization problem mentioned above will remain due to bulk-synchronization mechanisms across multiple TBs still present.

In an attempt to enable "true" data-dependent parallelism on GPUs, several task-based software execution schemes have been proposed to enable a producer-consumer model between tasks (i.e. TBs) and SMs. These schemes resemble Dataflow execution models [14, 18], but the main computation units are SMs instead of CPU cores. Tzeng et al. [30] proposed a scheme where tasks with resolved dependencies are inserted in a centralized first-come, first-served (FCFS) queue and executed. [6] proposed a scheduler-worker-based solution based on distributed queues, where task dependencies are maintained by a scheduler thread block via an in-memory dependency matrix and updated on-the-fly as the tasks are processed by the worker TBs. However, the major drawback for all these software solutions is their reliance on expensive global memory atomics as well as busy-waiting to handle task insertion & retrieval operations and inter-SM communication.

Fundamentally, there is a lack of support for conveying generalized data-dependent parallelism and inter-SM synchronization. While task-based execution schemes rely on long-latency global memory, others focus on improving CDP-based kernels by compile- or run-time optimizations to achieve better thread utilization for a specific class of applications (i.e. nested parallelism). Yet none of the aforementioned studies provide a generalized solution for an arbitrary network of inter-block data dependencies. To this end, we propose *Wireframe*[1], a hardware-software approach which provides generalized support for hardware execution of task-based dependency graphs.

Wireframe is built on the abstraction of *Dependency Graph (DG)* execution, where individual thread blocks are represented as *tasks*. These dependency graphs can be generated either through programmer API (*DepLinks*), or compiler profiling [15–17, 27, 31]. The dependency graph is then enforced in the hardware through a *Dependency-Aware Thread block Scheduler (DATS)*.

In this work, we show that Wireframe can be utilized to support a generalized dependency graph-based execution approach to enable programmers to naturally convey data-dependent parallelism. In addition, we show that Wireframe can be used to support lightweight

[1]The name "Wireframe" stems from the similarities between the graphs utilized in our benchmarks with standard 3D wireframe terrain models used in computer-aided design.

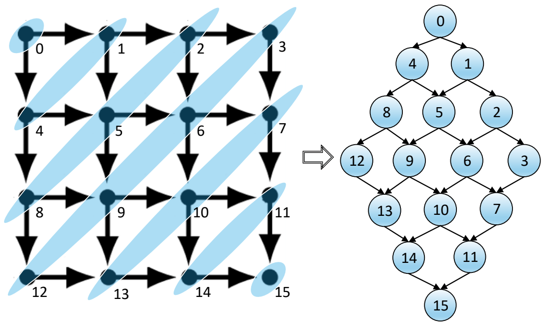


Figure 1: Wavefront pattern execution of thread blocks in an application kernel (left) and its equivalent dependency graph (right). The numbers represent the node IDs.

barrier and deadlock-free inter-block synchronizations through dependency graph primitives.

This paper makes the following contributions:

- In Section 2, we present a case for Wireframe and show how the dependency graph abstraction can be generalized for data-dependent parallelism and efficient synchronization.
- In Section 3, we present DepLinks to support programming data-dependent parallelism. We also present support for run-time dependency graph generation.
- Section 4 demonstrates hardware support for dependency graph execution through dependency-aware thread block scheduling (DATS). DATS enforces dependencies with a Dependency Graph Buffer (DGB) and maximizes ready nodes with Level-bounded thread block scheduling.
- Section 5 involves evaluation for Wireframe using a range of data-dependent workloads, measuring an average of 45% performance boost, with ∼2KB area overhead.

In Section 6, we discuss related literature and the paper finally concludes in Section 7.

2 MOTIVATION

In this section, we motivate a case for Wireframe. We will make use of Code Blocks 1-4 to motivate and drive this section. We use a basic wavefront pattern, a common data-dependent parallel pattern [7, 21, 35], as a running illustrative example due to its simple structure and clarity in conveying concepts in this paper. It should be stressed that our proposed technique is generic to all data-dependent parallel patterns and in no way limited to the examples presented here. Figure 1 displays a wavefront pattern. In wavefront parallelism, computation are typically dependent on neighbors, where data dependencies form diagonal 'waves' of computation (shown in blue). We define *task* as an abstract unit of computation. In this example, a task can be fine-grain and represent the computation of a single element in the wavefront, or it can be coarse-grain and represent a tile consisting of multiple elements. The dependencies between tasks in this workload is shown on the right, as a directed graph, which we call a *dependency graph*, with each node representing a thread block.

2.1 Data-dependent Parallelism

We will now demonstrate CUDA's current support for data-dependent parallelism, and highlight its limitations and challenges. An implementation of wavefront processing using global barriers is shown in

Code Block 1. Every wave computation maps to a kernel call, which processes the computation for that wave. As demonstrated in various prior work [6, 7, 29], this limitation of global barriers introduces significant overhead due to multiple kernel launches and requires programmers to map data-dependent parallelism to this rigid constraint. An alternative option so as to avoid multiple kernel launches is to enforce synchronization of waves within the thread block. This requires each wave to be processed entirely within a single thread block, which would severely under-utilize the GPGPU hardware.

Code Block 1: Global Barriers

```
int main() {
    for (int i=0; i<nWaves; i++) {
      kernel<<<GridSize, BlockSize>>>(args);
      cudaDeviceSynchronize();
    }
}
__global__ void kernel(args) { processWave(); }
```

In order to facilitate support for data-dependent nested parallelism, CUDA Dynamic Parallelism (CDP) was introduced. CDP enables device-side kernel launches, avoiding the overhead of host-side kernel launches. Every device-side thread has the ability to spawn a child kernel. CDP typically supports two common implementation methods - recursion and nesting. Code Block 2 shows an implementation of CDP using a recursive pattern. Here every wave is still processed by a single kernel and subsequent waves are handled by recursively launching another kernel until every wave has been processed. In lines 9-12, we have thread 0 spawning a single child kernel and wait for its completion. A main limitation of the recursive approach is the recursion depth limitation. In CDP, there is a maximum nesting depth of 24 [2], i.e 25 waves at most. This pattern works well for algorithms that can be mapped recursively, but otherwise inflexible.

Code Block 2: Dynamic Parallelism - Recursive

```
1  int main() {
2        kernel<<<GridSize, BlockSize>>>(0, args);
3        cudaDeviceSynchronize();
4  }
5  __global__ void kernel(i, args) {
6      if(i == nWaves) return;
7      processWave();
8      if(threadIdx == 0) {
9          kernel<<<GridSize,BlockSize>>>(i+1,args);
10         cudaDeviceSynchronize();
11     }
12     __syncthreads();
13 }
```

A more flexible implementation is shown in Code Block 3, where nested parallelism is used. In this approach [8], a parent kernel launches a child kernel for every wave. However, unlike recursive parallelism where the child will also spawn a child kernel of its own, the child returns, prompting the parent kernel to launch the next child kernel, which resolves the spawning depth limit issue in the recursive version. This approach is very similar to the global barriers implementation, but with the overhead of device-side kernel launch instead of host-side kernel launch.

Although this implementation is lower-overhead, the device-side kernel launches still incur non-trivial overhead [34] and there is also the limitation of coarse-grain synchronization across waves. This implicit synchronization introduced by kernel launches limits potential opportunities for nodes to run ahead and execute when ready. For example, during the 4th wave, if nodes 9 and 12 are ready, then node 13 is ready to execute, but has to stall until nodes 3 and 6 complete the wave. This limitation is mainly due to the 1-parent-*m*-child representation of CDP, where child kernels can only have a single parent. Thus the wavefront pattern has to be mapped to coarse-grain synchronization at wavefront boundaries.

Code Block 3: Dynamic Parallelism - Nested

```
int main() {
      parentKernel<<<GridSize, BlockSize>>>(args);
      cudaDeviceSynchronize();
}
__global__ void parentKernel(args) {
    for (int i=0; i<nWaves; i++) {
        if(threadIdx == 0) {
           childKernel<<<GridSize, BlockSize>>>(args);
           cudaDeviceSynchronize();
       }
        __syncthreads();
    }
}
__global__ void childKernel(args) { processWave(); }
```

In order to fully express the data-dependent parallelism of the wavefront pattern, we need a generalized approach to convey *n*-parent-*m*-child relationships. In our wavefront example, there are parent-child relationships with *2*-parent-*1*-child (e.g. node 9), *1*-parent-*2*-child (e.g. node 0), and *1*-parent-*1*-child (e.g. node 12). To this end, we present *DepLinks*, to support expression of generalized data-dependent parallelism. DepLinks is built on the abstraction of dependency graphs between tasks. In our framework, we partition a task as a single thread block (or CTA[2]) in hardware.

Code Block 4: Wireframe

```
#define parent1 dim3 (blockIdx.x-1, blockIdx.y, blockIdx.z);
#define parent2 dim3 (blockIdx.x, blockIdx.y-1, blockIdx.z);
void* DepLink() {
    WF::AddDependency(parent1);
    WF::AddDependency(parent2);
}
int main() {
   kernel<<<GridSize, BlockSize, DepLink>>>(args);
   cudaDeviceSynchronize();
}
__WF__ void kernel(args) {
    processWave();
}
```

Code Block 4 shows how wavefront parallelism can be expressed using DepLinks. In this scenario, we simply launch a kernel with a sufficient number of thread blocks to represent the entire dependency graph. One of the kernel launch options is a mapping function which defines the graph. This function consists of dependency links which are specified by *dim3* structures and its job is to specify the relative thread block on which any thread block is dependent. The dependency graph will then be generated by running the mapping function

[2]We use thread block and CTA interchangeably

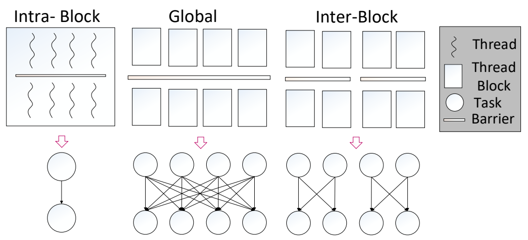


Figure 2: Synchronization barrier primitives using dependency graph abstraction: Intra-block (left), Global (middle) and Inter-block primitives (right).

on every available thread block. For instance, in our wavefront example, every node is dependent on its north and west neighbors. This dependency graph will then be passed to the GPGPU hardware to enforce data dependency at run-time. In the next section, we will discuss this process in detail. Due to the fine-grain data-dependency that we can convey, individual tasks can run ahead and execute when parent tasks are complete. In this execution pattern, tasks are not constrained to waves. Overall, Wireframe enables a natural and flexible way to convey data-dependent parallelism.

2.2 Barrier Synchronization Primitives

As mentioned before, another major challenge of data-dependent parallelism is the lack of support for flexible barrier synchronization. Inter-block synchronization can ease programmer burden by granting programmers flexibility to convey synchronization between TBs, which has limited support in CUDA 9 with Cooperative Groups. Our synchronization primitives have similar support as Cooperative Groups, but we will later showcase how Wireframe can further eliminate stalls due to barrier synchronization by supporting a programming paradigm to avoid barriers completely. In this section, we demonstrate how dependency graphs can be used to form primitives that enable flexible lightweight synchronization across thread blocks. Figure 2 shows the supported synchronization primitives.

Intra-block synchronization: As shown in Figure 2 (left), intra-block synchronization implements a barrier among threads inside of a single thread block. This is achieved with `__syncthreads()` in CUDA . In our dependency graph abstraction, intra-block synchronization can be conveyed through a *1*-parent-*1*-child relationship between tasks. Using this dependency graph representation actually imposes greater overhead than `__syncthreads()` due to using 2 thread blocks to achieve this task. Therefore, we still rely on intra-block synchronization using the standard `__syncthreads()` call.

Global synchronization: In Figure 2 (middle), a scenario is shown where we assume the kernel consists of 4 thread blocks. Traditionally, in order to globally synchronize all thread blocks, we require implicit synchronization through consecutive kernel calls. This suffers from significant overhead due to the need for host-side kernel launches. Using the dependency graph abstraction, we can represent global synchronization using a dependency graph where each individual task after the barrier is dependent on every task before the barrier. In this example, global synchronization is represented as a *4*-parent-*1*-child relationship. This lightweight global synchronization primitive completely eliminates the unnecessary host- and device-side kernel launches.

Inter-block synchronization: In Figure 2 (right), we illustrate inter-block synchronization with a scenario where thread blocks synchronize in pairs. This is similar to the global synchronization primitive where each individual task after the barrier is dependent on every task before the barrier, but constrained to a subset of thread blocks that are synchronizing. Supporting inter-block synchronization is a key component towards fully-supported data-dependent parallelism. What is unique about our approach is that this abstraction is deadlock-free. In prior work [36], a barrier is placed at the end of the thread block and wait for all other thread blocks to reach it. This results in some thread blocks staying in the SM, preventing other thread blocks from being scheduled in, and they will subsequently cause a deadlock because they never got scheduled to be finished. Unlike [36], our inter-block synchronization primitive does not result in deadlock because parent thread blocks are allowed to complete and exit the SM, with barrier dependencies checked before a new thread block is issued to an SM.

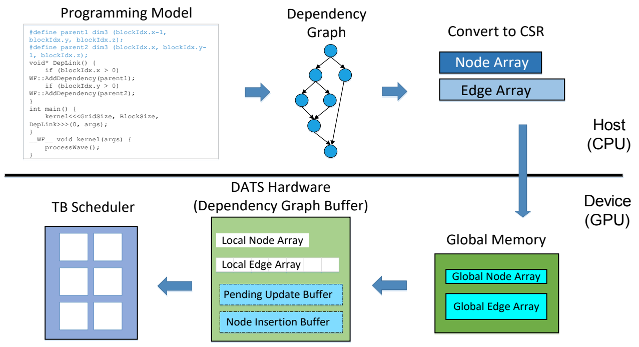
3 WIREFRAME

Figure 3 shows an overview of the Wireframe framework. Wireframe consists of three main parts: DepLinks extensions to the CUDA programming model, dependency graph generation, and dependency graph execution in hardware through our dependency-aware thread block scheduler (DATS). The programmer can express data-dependent parallelism and barrier synchronization through our CUDA programming model extensions. At kernel launch time, Wireframe would then retrieve the dependencies from the programmer via the API, create the dependency graph in Compressed Sparse Row (CSR) format and send it to the GPU hardware. Once the CSR is received by the hardware, the GPU will make use of the dependency graph to enforce data-dependent parallelism when scheduling TBs.

The interface between the software and hardware is simply an abstraction of task dependencies represented as CSR. Therefore, our framework is not tied to a specific programming interface. The dependency information between tasks can be in the order of hundreds of MBs, limiting prior dependency-based task scheduling to software run-times [6, 19, 30, 36] with significant overheads. Wireframe, to the best of our knowledge, is the first efficient hardware solution to support and manage dependency-based task scheduling with only 2KB hardware overhead.

Note that the focus of this paper is on efficient hardware support of statically generated dependency graphs. There are currently many efforts in various compilers and programming paradigms to convey task dependencies in CPUs [15–17, 27, 31]. For example, OpenMP [17, 27] contains extensions to define tasks and dependencies using the depend clause. This information is utilized to create a directed acyclic graph of the tasks. Till date, there is no software run-time-agnostic API for GPUs to convey task dependencies. This paper makes an argument for dependency awareness extensions to CUDA, and demonstrates the potential benefits.

As the main focus is on hardware support for dependency graph execution, we propose a simple API in order to convey static task dependencies. It will generate a CSR dependency graph, which enables easy interpolation with any future task dependency programming paradigms. Automatic generation of the graph is future work.



```
#define parent1 dim3 (blockIdx.x-1,
blockIdx.y, blockIdx.z);
#define parent2 dim3 (blockIdx.x, blockIdx.y-
1, blockIdx.z);
void* DepLink() {
    if (blockIdx.x > 0)
WF::AddDependency(parent1);
    if (blockIdx.y > 0)
WF::AddDependency(parent2);
}
int main() {
    kernel<<<GridSize, BlockSize,
DepLink>>>(0, args);
}
__WF__ void kernel(args) {
    processWave();
}
```

Figure 3: Overview of Wireframe; programmer supplied dependency constraints are translated into a dependency graph at run-time and conveyed to the GPU, where it is scheduled for execution through the DATS hardware.

3.1 DepLinks API

In this section, we present our DepLinks API and how it maps the TBs to the nodes in the dependency graph. Later, we discuss the in/out dependency concept from OpenMP which could also be used to profile and generate dependencies.

DepLinks requires API calls for scheduling policy assignment, inter-block synchronization, and assignment of parents for every thread block. In addition, DepLinks supports executing different kernels with different dependency graphs.

Code Block 5: Wireframe API Example

```
#define node1 dim3 (blockIdx.x-1, blockIdx.y, blockIdx.z);1
#define node2 dim3 (blockIdx.x, blockIdx.y-1, blockIdx.z);2
                                                          3
void* DepLink() {                                         4
       //Add dependency for every thread block            5
       WF::AddDependency(node1);                          6
       WF::AddDependency(node2);                          7
                                                          8
       //Set the policy for the hardware                  9
       WF::SetPolicy(WF::LVL,4);                         10
}                                                        11
                                                         12
__WF__ void kFunction(<args>)                            13
{                                                        14
   //Do kernel execution                                 15
}                                                        16
                                                         17
void main()                                              18
{                                                        19
   //Launch kernel kFunction()                           20
   dim3 dimGrid(4,4,1), dimBlock(16,16,1);               21
   kFunction<<<dimGrid, dimBlock, DepLink>>>(<args>);    22
}                                                        23
```

We demonstrate our API in Code Block 5. The code block implements a kernel, `kFunction` (line 22). The kernel calls are extended with a mapping function, `DepLink` (lines 4-11). The kernel maps a wavefront dependency graph similar to Figure 1. For every thread block in the kernel, it will call the mapping function to identify its parent dependency. In wavefront dependency, each node is dependent on its west (x-1) and north (y-1) neighbors. We have defined this in lines 1 and 2. However, the thread blocks do not always have identical dependency patterns. In that case, conditional statements could be utilized to differentiate dependencies related to different groups of blocks.

The `AddDependency()` call will map the dependencies to the thread block. In addition, in line 10, the thread block scheduling policy is specified as level-bound (LVL) with a range limit of 4, i.e. running TBs in the graph cannot be more than 4 levels apart. Overall, it is possible to declare wavefront dependency pattern in less than 10 lines of code.

Note that our API function implements boundary checking to handle invalid arguments. For example in the DepLink() function, for block ID (1,0,0), node1 will have negative elements in which the API will correctly handle and ignore. Similarly, block ID (0,0,0) will have no parent nodes.

Similar to OpenMP depend clause, we only need to specify each edge in a dependency graph. Using this simple, yet flexible, API we can also easily implement any synchronization barrier primitives shown in Figure 2.

Profiling-based generation: The OpenMP depend clause provides a list of dependent inputs and outputs for each task. This data flow information is then utilized to generate a DAG. Similarly, kernel calls in global barriers implementation follow a similar pattern, with input and output data to the kernel managed by cudaMemcpy. Therefore, it is feasible to extract data dependencies from the global barriers implementation by obtaining the data flow between the kernels without programmer intervention. Due to the indexing nature of TBs in CUDA programming, we can also profile the data flow between thread blocks to identify dependencies and generate the dependency graph. In prior work [15] parallel task-based dependencies were extracted from sequential programs using a similar technique.

3.2 Dependency Graph Generation

At kernel launch time, the program creates a static dependency graph based on the programmer-supplied dependencies.

In order to pass this information to the GPU in a compact manner, we chose to represent the dependency graph in modified compressed sparse row (CSR) format. The API gives us a list of nodes and edges from which we can generate the CSR with time complexity of $O(|V|+|E|)$. Our dependency graph CSR representation is shown in the upper half of Figure 6.

CSR consists of two arrays: a Node Array and an Edge Array. Every Node Array entry corresponds to a node, with three fields: Edge start, Parent count, and Level. How they are used is explained in the Section 4.2.1.

In Figure 6, the numbers in the Node Array correspond to the start indices in the Edge Array. For example, node 0 has child nodes 1 and 2, node 2 has child nodes 3 and 4, etc.

Our customized CSR array contains the number of nodes in the Node Array, the edge start and edge count for every node (in short, location of child nodes in the edge array and the number thereof), the number of edges and the nodes to which they lead.

A major challenge of using dependency graphs is their size. The size of the dependency graph is arbitrary and can be very large in the order of MBs. So the full CSR should be stored in the global memory (or constant memory if size permitting) of the GPU. However, the thread block scheduler requires dependency information from the

CSR in order to schedule thread blocks, which can be very slow with global memory access. To overcome this challenge, we will exploit spatial locality behaviors of actively executing nodes and their immediate child nodes in the dependency graph.

3.2.1 Dependency Graph Execution Properties. We observed that during the execution of data-dependent parallel applications, there exists spatial locality of actively executing nodes and immediately dependent nodes. In our dependency graph, there are no explicit or implicit barriers across different levels of the dependency graph. Due to the fine-grain dependency representation, it is possible for ready nodes to process ahead even when prior levels of the dependency graph are not fully processed. Despite this freedom, we observe that there exists a narrow 'window' of levels in which active nodes are executing. We demonstrate this in Figure 4. We ran the HEAT2D application with a dependency graph of 9216 nodes and 191 levels in GPGPU-Sim [5]. During run-time, we measured the level range of active tasks over the course of the application run. We observed that even though the dependency graph has 191 levels, the level range of the active nodes grows no more than 7. We found this behavior common in data-dependent parallel workloads.

Using this key observation, we can buffer only a small subset of the dependency graph in the thread block scheduler to effectively support a dependency graph of any size, while still enabling the thread block scheduler to quickly keep track of dependency statuses at run-time. We will discuss this hardware mechanism in detail in Section 4.

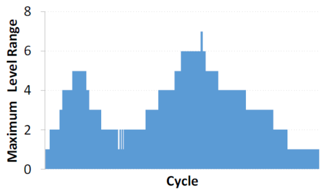


Figure 4: Level range during HEAT2D application.

3.2.2 Dependency Graph Node Renaming. By buffering subsets of the dependency graph, we are exploiting level locality. However, the current dependency graph and CSR format may not be amenable to buffering as CSR stores tasks in sequential node ID order (as defined by thread block IDs). In order to efficiently buffer the dependency graph, we need sequential ordering of levels and node IDs. As shown in Figure 5 (left), the dependency graph for the wavefront application in Figure 1 does not exhibit sequential level-by-level numbering. Therefore access to the CSR will result in non-contiguous global memory access, which also introduces major complexity issues when fetching nodes to buffer, as well as the management of the buffer. To overcome this, we perform a sequential level-by-level renaming transformation to the dependency graph as illustrated in Figure 5 (right).

Rather than changing the actual thread block IDs, we rename the node IDs. The dependency graph will be analyzed, every node's parents, children and level will be determined, and then each node will be assigned a 'virtual ID' (VID), which will be used exclusively by the thread block scheduler. The original thread block ID remains intact and is used as normal. The procedure is similar to breadth-first search and is performed at run-time. In the beginning, all nodes with no parents will be considered level 0. We then move down the graph and assign the child nodes recursively. For every child node with exactly one parent, the level of the child will be: $lvl_{child} = lvl_{parent} + 1$. For a general case where there are N parents, the level of the child will be: $level_{child} = 1 + \max\{level_{parent_i}\}, 1 \leq i \leq N$. When we move from every parent to a child, it increments the "parent counter" in the child node, which represents the number of parents for that node when this process is finished. This will be used in TB scheduling shown in the following sections.

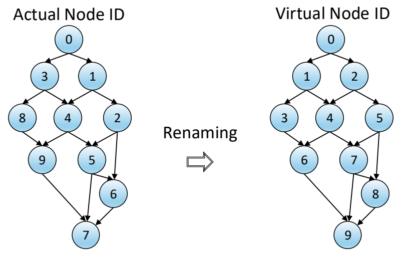


Figure 5: Illustrative example of node renaming

4 DEPENDENCY-AWARE THREAD BLOCK SCHEDULER (DATS)

In the previous sections, we described the DepLinks programmer interface and how dependency graphs are generated. In this section, we describe how the GPGPU hardware enforces dependencies through a dependency-aware thread block scheduler (DATS). As described in the last section, we use the CSR format to store the nodes and edges of a dependency graph, which is generated at run-time and transferred to the GPU's global memory. The CSR representation of the dependency graph in Figure 5 is illustrated in the global memory section of Figure 6.

4.1 GPGPU Architecture Overview

We target an NVIDIA Fermi-like architecture modeled after the GTX480. Our architecture comprises of 15 streaming multiprocessors (SM), where every SM in Fermi can execute up to 1536 threads or 8 thread blocks (CTAs). A thread block scheduler is responsible for issuing any ready thread block to an available SM. The technique that we present in this paper is agnostic to the GPGPU microarchitecture and is self-contained within the thread block scheduling mechanism. Kernel parameters are stored in the global memory. There is already a communication path between the global memory and the kernel management/distribution unit [34] to allow transfer of the CSR into the thread block scheduler. In this architecture, we assume only single kernel execution.

4.2 Dependency Graph Buffer

The Dependency Graph is stored in CSR format in the global memory. It is infeasible for the thread block scheduler to keep track of the graph node states in the global memory. Therefore we propose the addition of a *Dependency Graph Buffer (DGB)* into the thread block scheduler to buffer a subset of the CSR, sized large enough to keep the execution flowing and prevent it from stalling. In Section 3, we observed that there is spatial locality in the actively executing and immediate dependent nodes of the dependency graph. Therefore, we can buffer this active 'window' of nodes of the dependency graph in

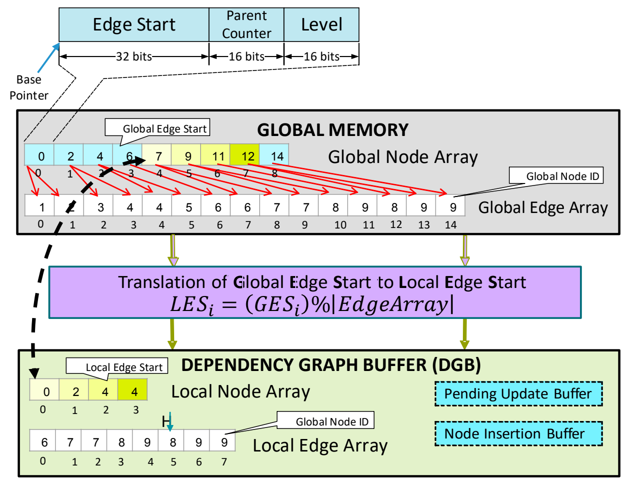


Figure 6: Connections between the global memory and local Dependency Graph Buffer for the graph in Figure 5.

the Dependency Graph Buffer in order to keep track of dependency states in a low-overhead manner.

The Dependency Graph Buffer is showcased at the bottom of Figure 6. The Dependency Graph Buffer consists of Local Node Array, Local Edge Array, Pending Update Buffer, Node Insertion Buffer, and Node State Table (not shown, details in Figure 7). The Local Node Array and Local Edge Array are implemented as circular buffers. The Node State Table is tightly coupled with the Local Node Array, where every node entry of the Local Node Array has a corresponding entry in the Node State Table. The node and edge values for the relevant dependency graph are stored in the global memory in the format shown at the top of Figure 6.

When moving portions of the global node/edge array into the local node/edge array, we re-index the global edge start to a local edge start. This is done using a simple modulus-based mapping function to minimize the size of the local node array entries. The local arrays will be loaded from memory in bursts of 128 bytes, the memory request size, to maximize memory load utilization.

We will now discuss the Dependency Graph Buffer in detail.

4.2.1 Node State Table.

The Node State Table is shown in Figure 7 and contains the following fields:

State: Signifies whether the node is Waiting (W), Ready (R), Processing (P), or Done executing (D). Initially all nodes are initialized to the Waiting state, except for the nodes with no dependencies which are set to Ready.

Parent Count: For every node, it shows number of unfinished parent nodes. It is computed in the host and transferred to and stored in the global CSR memory at run-time.

Level: The maximum distance of every node from a root, i.e. a node with no dependency. It is also computed at the host, to be used for thread block scheduling as discussed in Section 4.3.

Global Node #: The virtual node ID of the dependency graph, which indexes into a node in the Global Node Array.

Local Edge Start: The address of the first child of the node in the local edge array. If there were no children, it will be set to -1. The number of children can be determined by finding the difference of two consecutive edge starts.

4.2.2 Dependency Graph Buffer (DGB) Management.

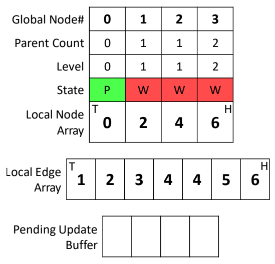
We will now demonstrate the operation of our DGB management mechanism. To illustrate the operation of our scheme, we make use of the DGB structure shown in Figure 7.

Transferring Nodes/Edges from Global Memory to DGB: The hardware fetches ‘chunks’ of nodes and edges from the global memory into the DGB. A chunk is defined as the number of the entries of the Local Node array which fit in a single memory request from the global memory (128B) to the DGB. In our running example, a chunk is 2 node entries, the Node Array size is 4, and Edge Array size is 12. During the transfer, the ‘edge start’ field of the local node array is re-indexed so they point to the local edge array directly. The Local Edge array, on the other hand, will keep the global node IDs so they can be used to update the parent counter of children nodes. To illustrate this, let’s look at global node ID 4. In Figure 6, this refers to index 4 of the Global Node Array, which contains a Global Edge Start of 7. The neighbor node (ID 5) has a Global Edge Start of 9, which means node 4 has 2 children. At index 7 of the Global Edge Array we see that node 4 has node 6 for a child, and at index 8, node 4 has child 7. Once node 4 is transferred to the DGB, as illustrated at the bottom of Figure 6, ID 4 has a transformed local edge start of 0, which points to index 0 of the local edge array. Index 0 and 1 of the local edge array contain the global node ID of children 6 and 7.

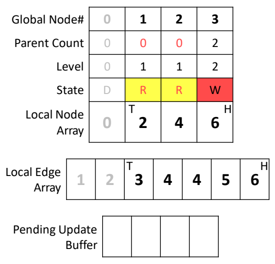
Translating Global to Local: The translation of the edge start from the global memory to the local memory is modulus-based: $LES_i = (GES_i)\%|LEA|$, where LES_i is the translated Local Edge Start for node i, GES_i is the Global Edge Starts for nodes i and $|LEA|$ is the size of the Local Edge array. Let us use Figure 6 as an example, where $|LEA| = 8$. Suppose that nodes 0 to 3 are already inserted in the DGB along with their edges, so they are both full. Then nodes 0 and 1 finish and, as a result, are invalidated in the DGB. Since the chunk size is 2, nodes 4 and 5 load into the DGB. As the Global Edge starts for nodes 4 and 5 are 7 and 9, their new Local Edge starts will be $LES_4 = 7\%8 = 7, LES_5 = 9\%8 = 1$ respectively.

The local node address for node i is also modulus-based. At the time of the node’s insertion from the global memory into the dependency graph buffer, the operation is performed in the following address: $LNID_i = i\%|LNA|$, where $LNID_i$ is the local address for node i at the time of its insertion and $|LNA|$ is the size of the Local Node array. Since both i and $|LNA|$ are known before the node’s insertion into the Local Node Array, the hardware can predict the future location of any node in the said array. In the event of any new node transfer, the hardware will compare the global node ID of the new node and the target location to check if the latter is indeed unused. If the location is occupied by a prior node, it would terminate the memory transfer and put the node in the node insertion buffer until the space becomes available.

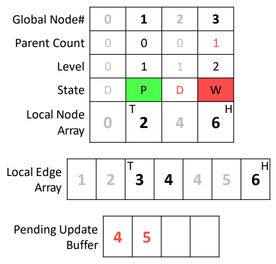
Handling Head Pointer Node Once the nodes are inserted into the graph buffer, ready nodes can be issued in any order. The only exception is the last node pointed by the head pointer. When a ready node completes, it decrements the parent counter of each children. In order to do so, we must know the number of children each node has by subtracting its local edge start from that of the next node’s local edge start. For the last node pointed by the head pointer, it cannot determine the number of children due to the absence of a next node. We handle this scenario by not scheduling the head pointer node, unless the node is childless, e.g. the last node.



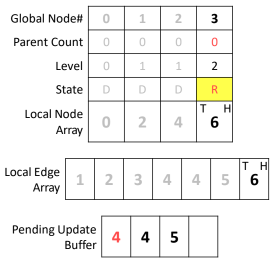
(a) Initial state: Global Node 0 is processing. Global Nodes 1, 2, 3 are in the waiting state.



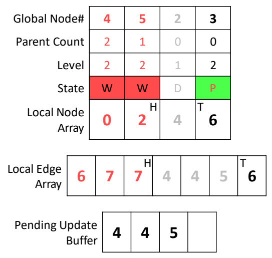
(b) Node completion: Global Node 0 is finished, parent counters for global nodes 1 and 2 decrement, nodes become ready. Tail pointer moves to next valid entry. Global nodes 1 and 2 will begin processing when TB scheduled.


Global Node# 0 1 2 3
Parent Count 0 0 0 1
Level 0 1 1 2
State D P D W
Local Node Array 0 T 2 4 6 H
Local Edge Array 1 2 T 3 4 4 5 6 H
Pending Update Buffer 4 5

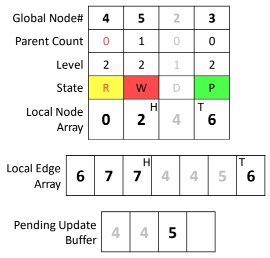
(c) Node completion w/o local child: Global node 2 completes execution and seeks to decrement the parent counters of its children, nodes 4 and 5. Since they are not in the node array yet, the updates are saved for later in the pending update buffer.


Global Node# 0 1 2 3
Parent Count 0 0 0 0
Level 0 1 1 2
State D D D R
Local Node Array 0 2 4 T H 6
Local Edge Array 1 2 3 4 4 5 T H 6
Pending Update Buffer 4 4 5

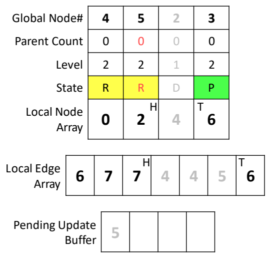
(d) Global node 1 also completes and decrements parent counter for global node 3 and 4. Global node 3 is now ready. Global node 4 is not in the graph buffer yet, the update will be saved in the update buffer. Tail pointer moves to next valid entry.


Global Node# 4 5 2 3
Parent Count 2 1 0 0
Level 2 2 1 2
State W W D P
Local Node Array 0 2 H 4 T 6
Local Edge Array 6 7 7 H 4 4 5 T 6
Pending Update Buffer 4 4 5

(e) Loading nodes: Enough spaces in the local node array prompt the graph buffer to load more content. Nodes 4 and 5 will be loaded into the graph buffer. Head pointer moves. Node 3 begins execution.


Global Node# 4 5 2 3
Parent Count 0 1 0 0
Level 2 2 1 2
State R W D P
Local Node Array 0 2 H 4 T 6
Local Edge Array 6 7 7 H 4 4 5 T 6
Pending Update Buffer 4 4 5

(f) Pending Update Buffer Removal: Node 4 parent counter decremented from pending update buffer entries. Global node 4 parent counter becomes 0, thus ready for execution.



(g) Global node 5, which will now receive the updates from the pending update buffer and become ready.

Figure 7: Management of Dependency Graph Buffer.

Node Completion Figure 7b illustrates the scenario where a node completes execution. This running example starts with nodes 0-3 in the DGB, with node 0 executing as shown in Figure 7a. Since node 0 has no parents, it has level 0 and will be the first to execute. When node 0 completes, we first fetch the children of node 0 (node 1 and node 2). Each child is accessed and their parent counter is decremented. Once a parent counter reaches 0, it indicates that all dependencies are met, and its state is updated to ready.

After decrementing the parent counters, the entry associated with the node which finished is invalidated as shown by the lighter text. Recall that the Local Node/Edge Arrays are circular buffers. As the tail entries are invalidated, we move the tail to the next valid entry. In this case, the tails moved to Local Node Array index 1 and Local Edge Array index 2. In addition, the execution of thread blocks can be completed out of order in the array as shown in Figure 7c, where node 2 has finished executing while node 1 is still being processed. We only move the tail if the tail's entry is invalidated.

Pending Update Buffer Insertion: Note that when node 2 finishes, the children nodes 4 and 5 are not in the node status table, and thus we cannot decrement their parent counters. To overcome this overflow, we add a *Pending Update Buffer (PUB)* to handle the situation where the child node is not in the Local Node Array. The PUB stores the global node ID of the child. This is illustrated in Figure 7c, where node 2 has finished executing and attempts to update the parent counters of its children, nodes 4 and 5. Since neither of those nodes is in the graph buffer yet, it will use the PUB to save the changes so they can be applied later. Note that if the buffer is full, the hardware cannot mark the node as complete if it has children. Therefore it has to wait until there is enough space before the node's execution can be finalized.

Loading Local Node/Edge Array Entries As shown in Figure 7d, node 1 will complete, and decrement the parent counter for its children nodes 3 and 4. Node 3's parent counter is now at 0 and its state is updated to ready. Node 4 is not in the Node State Table, and is thus put into the PUB. At this point, the entries for node 1 and 2 are invalidated and the tail advances to index 3 (node 3). At this point in time, there is enough empty space in the Local Node Array to load a new chunk of nodes. We can keep track of the available space using the distance between the head and tail pointers.

A memory request is issued to the global memory and the next chunk is fetched from it. From the head pointer in the Local Node

Array, we can generate a memory access to load the next node based on its ID: $BaseAddress_{Global} + NodeID \times NodeEntrySize$.

The Global Node Array entries contain the global edge start, which points to the Global Edge Array. Memory requests are iteratively issued to fetch the Global Edge Array entries with memory address location calculated similar to accessing Global Node Array.

The hardware loads a new chunk from the global memory into the node array where the head pointer is, followed by the associated data in the edge array, starting from the edge head pointer. This is depicted in Figure 7e. A node's insertion is only finalized if there are enough spaces for its edges in the Local Edge Array. Otherwise the node shall be put in a temporary *node insertion buffer* to wait and the loading process halts. The next nodes will not also be loaded until enough space for the node in question and its edges is available in the Local Edge Array, in which case loading will resume. If the node's insertion into the local memory is successful, the head pointer will then also move. Note that the edges to which the nodes will be pointing have been translated to their local counterparts beforehand as described earlier in this section.

Pending Update Buffer Removal: Figure 7f shows the scenario when nodes 4 and 5 are loaded into the Node State Table. At this point in time, the update buffer contains two updates for node 4 (one each from completion of node 1 and 2), and one update for node 5 (from node 2). When a node with a registered ID in the update buffer is loaded into the Node State Table, the parent counter update will be applied and the entry in the update buffer removed. For example, in Figure 7f, the two pending node 4 entries decrement the parent counter of node 4 to 0, changing the state of node 4 to ready. Similarly, the update to node 5 will also be applied, marking it as ready (Figure 7g).

4.3 Level-bound Thread Block Scheduling

Up until now, we have described how dependencies between thread blocks are enforced and managed in hardware. We will now discuss how ready thread blocks are scheduled to SMs. We first present the baseline thread block scheduling policy, and then motivate the need for a thread block scheduling policy for dependency graphs.

The baseline default policy is *Loose Round-robin (LRR)*. It first selects a ready node with the smallest ID, and cycles among all the SMs, selecting the next SM to issue to. If the intended SM is already full, the policy will move to the next available SM. This policy attempts to evenly distribute the workload among the available resources. However, this scheduler is very simplistic and does not account for dependency graph execution dynamics, which leads to performance hindrance.

We will again borrow the wavefront example from Section 2 to illustrate dependency graph execution dynamics. In coarse-grain synchronization scenarios (global barrier, CDP), each *level* of the dependency graph is executed until completion, one after another. As a result, nodes ready in subsequent levels cannot be scheduled and must wait until the preceding level is complete, limiting performance. Dependency graph execution allows any ready node to run ahead and execute without having to wait until the prior level is completed.

However, we observed that if nodes run ahead too far, it can end up hampering the performance. This is illustrated in Figure 8. Here 8 nodes (marked with 'D') have completed execution. On the left, we depict a potential scenario with the baseline LRR policy where

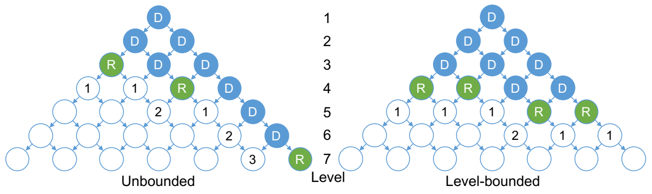


Figure 8: Effect of thread block scheduling on Dependency Graph node availability.

nodes can run ahead unbounded. There is significant run-ahead, with ready nodes ('R') spanning a *level range* of 4 (in levels 3, 4, and 7). Due to a single path running ahead, it can potentially limit the number of ready nodes. In the case of wavefront, there are significant data-dependencies with most nodes dependent on 2 parents from the previous level. If a dependency graph observes a high level range, it means that neighboring nodes may have more dependencies pending. For example, in Figure 8, the numerical values within the immediate neighboring nodes of completed and processing nodes represent the levels of dependencies that must be resolved before that node can run. Due to the run-ahead, neighboring nodes have a longer chain of dependencies with less nodes ready in the near future (only 3 nodes have 1-level dependency).

To this end, we propose *Level-bounded (LVL)* thread block scheduling, which extends the LRR thread block scheduler by bounding the level range to satisfy dependencies quicker. This results in greater ready node availability as shown in the figure. Under level bounding, we have 5 nodes with 1-level of dependency, and also 4 ready nodes. Intuitively this scheduler operates in the following manner: If a path runs ahead too far (reaches a level range limit), the Level-bounded scheduler will prevent that path from proceeding further and favor scheduling nodes from slower paths to allow the level range to narrow. Bounding the level range promotes completion of node dependencies, resulting in more ready nodes than the baseline unbounded scenario.

5 EVALUATION

5.1 Methodology

We evaluate Wireframe on GPGPU-Sim v3.2 [5]. We use the default NVIDIA GTX480 configuration with 15 SMs, each having 8 CTAs, 128KB register file and 16KB L1 cache size. The shared L2 cache size is 786KB. The warp scheduling policy follows a greedy-then-oldest (GTO) policy [24]. Our thread block scheduling technique can be run with any warp scheduler, but we find GTO to provide the best performance. We modeled the device-side kernel launch overheads by implementing the latency model proposed in [34]. We measured empirically and used the host-side kernel launch time of $30\mu s$. The baseline machine runs at a core clock of 700MHz, where each SM consists of 2 shader processors (SP), each containing 32 CUDA cores, 16 LDST units and 4 SFUs.

We utilize a selection of data-dependent heavy workloads. For each workload, we implement four versions: Global Barriers (Global), CUDA Dynamic Parallelism (CDP), DepLinks synchronization primitives (DepLinks), and Wireframe with the LRR scheduler (LRR) and Level-bound scheduler (LVL). For the level-bound scheduler, we use a level bound of 3. Note that DepLinks enables barrier synchronization primitive support through task graph representation and does not change the way TBs are assigned to SMs. LRR and LVL, on the other hand, do not enforce any barrier behaviors, but rather control the TB assignments, allowing nodes with satisfied dependencies to execute, enabling them to run-ahead instead of waiting for other nodes at their level to finish first.

We verified the output of each workload implementation against the original to ensure output correctness and that dependencies are satisfied safely. Unless otherwise stated, we partition the workload with up to 4K nodes in the dependency graph. We will later explore the impact of the size of dependency graph on performance. In addition, we used a Local Node Array size of 128 entries and a Local Edge Array size of 512 entries for LVL scheduler, and 512 entries and 2K entries for LRR scheduler, respectively. We set the size of the Pending Update Buffer to 64 entries.

5.2 Benchmarks

The benchmarks used are DTW (Dynamic Time Warping) [22], HEAT2D [25], HIST (Histogram) [23], INT_IMG (Integral Image) [9], SOR (Successive Over-Relaxation) [12] and SW (Smith Waterman) [26]. DTW is a common algorithm in time series analysis for measuring similarity between two time series with varying speeds. DTW takes in two time series, one of size 12K and one of size 8K. HEAT2D is a common solver for heat equations in two dimensions. At every iteration, the temperature of each point is dependent on neighboring points. We use a 2D grid of size 12K x 12K. HIST calculates the integral histogram over a 13MP bitmap image. INT_IMG is an image processing technique that generates the sum of values in a rectangular subset of a grid. We similarly use INT_IMG with a 13MP bitmap image. SOR is a linear system solver which is implemented using a generic 5-way stencil pattern. We use a random 2D matrix with 144M entries as input. SW is a common local sequence alignment algorithm. We input two 8K strings. We verified that the size of the data is sufficient to utilize the entire GPU (maximize hardware CTAs, cache, etc.) with each workload's data set size in the order of hundreds of MBs.

5.3 Evaluation Results

Performance: Figure 9 illustrates the speedup for all implementations with respect to Global Barriers. Speedup is the ratio of the total execution time and kernel launch overhead for a given technique, with respect to the baseline global implementation. It shows how much every technique addition, up to LVL, is responsible in improving the performance. In all scenarios, CDP and our proposed techniques outperform global barriers by removing costly host- and device-side kernel launches. CDP has an average speedup of 6.87%. DepLinks further remove device-side kernel launches and improve average speedup by 25.07%. Wireframe further enables task run-ahead. On average for Wireframe, LVL outperforms LRR (31.81% vs 29.81%). In certain scenarios, such as HIST, LVL performs slightly worse due to limited improvement to level range properties.

Memory Overhead: Figure 10 (left) shows the memory request overhead introduced by DATS. At most, DATS introduce 0.16% memory request overhead, with an average overhead of 0.12%. Despite making use of the global memory, Wireframe does not have a substantial negative impact to L2 cache performance as shown in Figure 11. In addition, the miss rate is consistent regardless of the programming model of Wireframe used (DepLinks vs LRR/LVL).

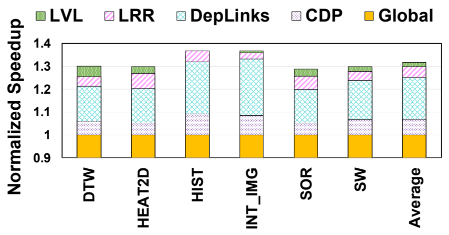


Figure 9: Normalized Speedup w.r.t Global Barriers

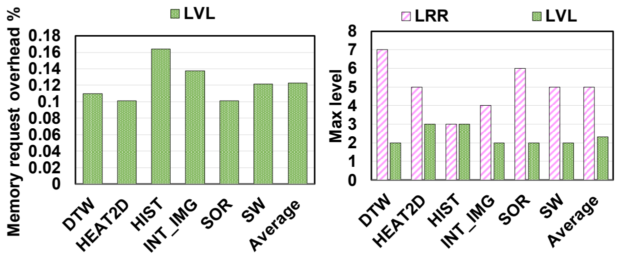


Figure 10: Memory request overhead (left) and maximum level range (right).

Level Range: Figure 10 (right) shows the impact of the LVL scheduler, with level bound of 3, on the maximum observed level range. We utilize a dependency graph of 9K as the benefits of level-bounding is more apparent with larger graphs. The observed level range can actually be less than the bound. For example, GTO warp scheduler focus on the warps of thread blocks on the lower levels (older TBs) so they can finish faster, resulting in a lower range than anticipated. In certain scenarios, the level range reduced drastically from 7 to 2 (DTW) and 6 to 2 (SOR), with an average level range dropped from 5 to 2.

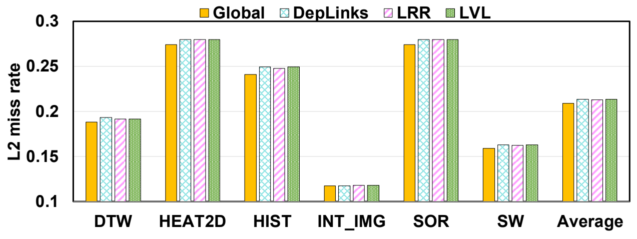


Figure 11: L2 miss rate

Dependency Graph Size: Figure 12 (left) shows the effect of dependency graph size on overall speedup for the level-bound scheduler normalized to the global barriers implementation. As the size of the graph increases, there is generally more levels, and greater opportunity for run-ahead. In addition, this is associated with removal of more global barriers. This can be observed as the average speedup increases as the graph size grows: 14.11% for 1K, 31.81% for 4K, and 45.07% for 9K dependency graph size, with a maximum speedup of 65.20%. Furthermore, Figure 12 (right) shows the computation time to kernel launch time ratio with a constant data size. Therefore the ratio decreases as the graph grows. Notwithstanding, on average, we have significantly more computational time than kernel launch time, with an average of 8x, 5x, and 3x more compute with graph size 1K, 4K, and 9K, respectively.

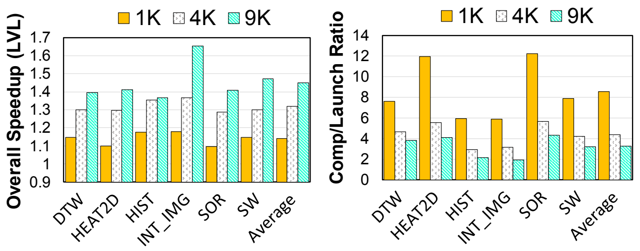


Figure 12: Overall speedup for different graph sizes (left) and compute-to-kernel-launch ratio (right)

5.4 Overheads

Sensitivity to Local Node/Edge Array Size: Figure 13 (left) shows the maximum pending update buffer usage (solid line) and the IPC (dotted line) with respect to varying the Local Node Array size for LRR and LVL using the SOR benchmark. We present SOR as it utilizes PUB the most. The LVL scheduler requires a notably smaller update buffer than the regular LRR scheduler. We use a node array size of 128 entries as it provides a high level of IPC for LVL, with a manageable PUB usage of almost 32 entries, which we pick as the best size for PUB. For a PUB usage of equal size, LRR scheduler requires Local Node Array size of 512.

Figure 13 (right) shows how the maximum pending update buffer usage changes as we decrease the Local Edge Array size. We set the Local Node Array size as before. For LRR, IPC falls as we use less than 512 entries. Thus we select 512 as the best edge array size. LRR meanwhile requires over 1K entries. LVL scheduler can significantly reduce the size of the Local Node/Edge Array needed.

Dependency Graph Buffer Size: The DGB requires very little space. Each local node array entry needs 2 bits for state, 16 bits for the global node ID, 16 bits for the parent counter, 16 bits for the level and 9 bits to address the local edge array, i.e. 58 bits in total, which rounds up to 8 bytes for each node in the Node State Table and Local Node Array. As for the Local Edge Array, it only needs 16 bits per element to store the target's global node ID, for a total of 1KB. In addition, we have a Pending Update Buffer of 32 entries of 2 bytes each and a single Node Insertion Buffer of 128B, for a total of 256B. In total, the DGB has a size of 2304 bytes, which is negligible in comparison to the size of register file per SM (128KB).

DGB Access Overheads: We can access and update the DGB quickly due to the small size. Any timing overheads would occur due to fetching chunks from memory. However, this operation is off the critical path as fetching of memory chunks can occur as TBs are executing on SMs. The only time there may be timing penalties due to our DGB mechanism is if there are no ready nodes due to nodes still being loaded from memory. Due to having a Local Node Array size of 128 entries, we observed that this scenario is rare as there are always plenty of other nodes to schedule. We observed the timing overheads of DGB to be less than 1%.

6 RELATED WORK

Synchronization: Existing GPGPU programming models have been designed with support for coarse granular synchronization primitives (commands and streams in CUDA, events and pipes in OpenCL, and pipelines in OpenACC) to enable flow control across

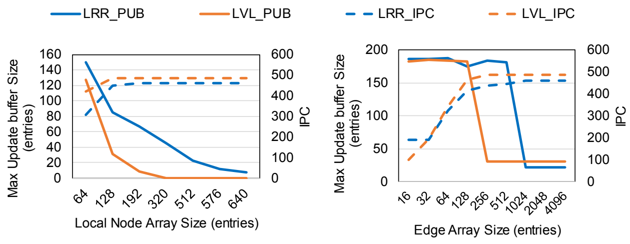


Figure 13: Effect of the local node size (left) and local edge array size (right) on the maximum update buffer size

multiple kernel launches and data transfers. However, these primitives are at the device level and only able to provide coarse granular dependency management between host-initiated calls. Such constructs fail to address the data-dependency requirements across the threads within the execution of a kernel.

A finer granular in-GPU synchronization across the TBs of a kernel enables better utilization of SMs by allowing the dependencies to be resolved locally. One of the major issues which is often encountered in in-GPU synchronization is the deadlock problem [36]. Consider a case where there are many thread blocks with a global barrier, all parenting a single child kernel, as shown in Figure 2 (middle). If the number of thread blocks exceed the total number of CTAs that can run concurrently on all the SMs, at a point some thread blocks could be running on the SMs and hit the global barrier, whereas the others have never been dispatched, and therefore cannot context switch, causing the GPU to enter a deadlock state. In [36], the deadlock is handled by using atomic operations and memory flags. However, this situation will not occur in Wireframe due to the absence of global barriers. A similar method is to transform algorithms to remove global barriers, such as PeerWave [7]. However, this requires significant programming effort and is not general purpose. However, all of these techniques are software-based and result in significant run-time overhead.

Reducing Kernel Launch Overhead: In [33], authors proposed a locality-aware thread block scheduler to schedule child nodes to maximize cache locality within dynamic parallelism (CDP). However, the maximum recursion depth it can reach in any workload is limited to 24 [1]. Wireframe, however, support more complex parent-child relationships which are configurable by the user. This makes the whole execution flow more manageable and efficient.

In [36], Xiao proposed an improvement of the subkernel launch using GPU lock-based and lock-free synchronizations. In [10], G. Chen et al. emphasize on re-using parent thread to operate on the data to be processed by the child kernel. In [28], Tang et al. coordinates dynamically-generated child kernels to reduce launch overheads and schedules both parent and child kernels to improve launch overhead hiding. In our work, we represent data dependencies through DepLinks rather than implicitly through kernels or barriers, and therefore completely avoid kernel launch overheads.

Dataflow Scheduling: In addition to the GPU-related work mentioned above, the problem that Wireframe targets has generally been addressed in the architecture literature as "Dataflow scheduling". Etsion et al. [14] have developed a superscalar, out-of-order task pipeline to execute dataflow programming models. Gupta et al. [18] utilized run-times to exploit parallel dataflow execution out of serial programs on multicores. Wang et al. [32] have implemented a task-level dataflow execution engine on FPGAs. More recently, Avron et al. [4] studied hardware task scheduling performance on Plural many-core-architecture. All these works present state-of-the-art examples for supporting data-flow based task execution on various platforms, however none of them addresses the problem for GPUs.

Thread Block Scheduling: One of the works considering the CTA behavior for the scheduling decisions is OWL [20], in which the authors tackle the hardware under-utilization issue by prioritizing certain thread block groups and improving the cache hit rates. To the best of our knowledge, our scheduler is the first to target data-dependent parallelism.

7 CONCLUSION

In this work, we propose a general-purpose data-dependent parallelism paradigm, Wireframe, which dramatically improves the performance on GPGPU by eliminating the need for global barriers and careful assignment of thread blocks as per the scheduling policy. Wireframe has shown an average speedup of up to 45.07% across multiple benchmarks.

ACKNOWLEDGMENT

This work is partly supported by NSF Grants CCF-1423108, CCF-1513201. The authors would like to thank the anonymous reviewers for their invaluable comments and suggestions.

REFERENCES

[1] 2012. Dynamic Parallelism in CUDA. http://developer.download.nvidia.com/assets/cuda/docs/TechBrief_Dynamic_Parallelism_in_CUDA_v2.pdf. (2012).
[2] 2016. CUDA Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/. (2016). Accessed: 09-27-2016.
[3] 2017. CUDA 9 Features Revealed: Volta, Cooperative Groups and More. https://devblogs.nvidia.com/parallelforall/cuda-9-features-revealed/. (2017).
[4] Itai Avron and Ran Ginosar. [n. d.]. Hardware Scheduler Performance on the Plural Many-Core Architecture. In *Proceedings of the 3rd International Workshop on Many-core Embedded Systems (MES '15)*. ACM, New York, NY, USA, 48–51.
[5] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 163–174.
[6] Mehmet E. Belviranli, Chih-Hsun Chou, Laxmi N Bhuyan, and Rajiv Gupta. 2014. A paradigm shift in GP-GPU computing: task based execution of applications with dynamic data dependencies. In *Proceedings of the sixth international workshop on Data intensive distributed computing*. ACM, 29–34.
[7] Mehmet E. Belviranli, Peng Deng, Laxmi N. Bhuyan, Rajiv Gupta, and Qi Zhu. 2015. PeerWave: Exploiting Wavefront Parallelism on GPUs with Peer-SM Synchronization. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS '15)*. ACM, New York, NY, USA, 25–35.
[8] Lars Bergstrom and John Reppy. 2012. Nested data-parallelism on the GPU. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 247–258.
[9] Berkin Bilgic, Berthold KP Horn, and Ichiro Masaki. 2010. Efficient integral image computation on the GPU. In *Intelligent Vehicles Symposium (IV), 2010 IEEE*. IEEE, 528–533.
[10] Guoyang Chen and Xipeng Shen. 2015. Free launch: optimizing GPU dynamic kernel launches through thread reuse. In *Microarchitecture (MICRO), 2015 48th Annual IEEE/ACM International Symposium on*. IEEE, 407–419.
[11] Long Chen, Oreste Villa, Sriram Krishnamoorthy, and Guang R Gao. 2010. Dynamic load balancing on single-and multi-GPU systems. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 1–12.
[12] Peng Di, Hui Wu, Jingling Xue, Feng Wang, and Canqun Yang. 2012. Parallelizing SOR for GPGPUs using alternate loop tiling. *Parallel Comput.* 38, 6 (2012), 310–328.
[13] Izzat El Hajj et al. 2016. KLAP: Kernel Launch Aggregation and Promotion for Optimizing Dynamic Parallelism. In *MICRO'16*.
[14] Yoav Etsion, Felipe Cabarcas, Alejandro Rico, Alex Ramirez, Rosa M Badia, Eduard Ayguade, Jesus Labarta, and Mateo Valero. 2010. Task superscalar: An out-of-order task pipeline. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 89–100.
[15] Alcides Fonseca, Bruno Cabral, João Rafael, and Ivo Correia. 2016. Automatic Parallelization: Executing Sequential Programs on a Task-Based Parallel Runtime. *International Journal of Parallel Programming* 44, 6 (2016), 1337–1358.
[16] Priyanka Ghosh, Yonghong Yan, and Barbara Chapman. 2012. *Support for dependency driven executions among openmp tasks*. IEEE.
[17] Priyanka Ghosh, Yonghong Yan, Deepak Eachempati, and Barbara Chapman. 2013. A prototype implementation of OpenMP task dependency support. In *International Workshop on OpenMP*. Springer, 128–140.
[18] Gagan Gupta and Gurindar S Sohi. 2011. Dataflow execution of sequential imperative programs on multicore architectures. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 59–70.
[19] Kshitij Gupta, Jeff A Stuart, and John D Owens. 2012. A study of persistent threads style GPU programming for GPGPU workloads. In *Innovative Parallel Computing (InPar), 2012*. IEEE, 1–14.
[20] Adwait Jog et al. 2013. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 395–406.
[21] Leslie Lamport. 1974. The parallel execution of DO loops. *Commun. ACM* 17, 2 (1974), 83–93.
[22] Meinard Müller. 2007. *Dynamic Time Warping*. Springer Berlin Heidelberg, 69–84.
[23] Mahdieh Poostchi, Kannappan Palaniappan, Filiz Bunyak, Michela Becchi, and Guna Seetharaman. 2012. Efficient GPU implementation of the integral histogram. In *Asian Conference on Computer Vision*. Springer, 266–278.
[24] Timothy G Rogers, Mike O'Connor, and Tor M Aamodt. 2012. Cache-conscious wavefront scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 72–83.
[25] Jason Sanders and Edward Kandrot. 2010. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional.
[26] Edans Flavius de O Sandes and Alba Cristina MA de Melo. 2013. Retrieving smith-waterman alignments with optimizations for megabase biological sequences using GPU. *IEEE Transactions on Parallel and Distributed Systems* 24, 5 (2013), 1009–1021.
[27] Maria A Serrano, Alessandra Melani, Roberto Vargas, Andrea Marongiu, Marko Bertogna, and Eduardo Quinones. 2015. Timing characterization of OpenMP4 tasking model. In *Proceedings of the 2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. IEEE Press, 157–166.
[28] Xulong Tang, Ashutosh Pattnaik, Huaipan Jiang, Onur Kayiran, Adwait Jog, Sreepathi Pai, Mohamed Ibrahim, Mahmut T. Kandemir, and Chita R. Das. 2017. Controlled Kernel Launch for Dynamic Parallelism in GPUs. In *2017 IEEE 23rd International Symposium on High Performance Computer Architecture (HPCA)*.
[29] David Tarjan, Kevin Skadron, and Paulius Micikevicius. [n. d.]. The art of performance tuning for CUDA and manycore architectures. ([n. d.]).
[30] Stanley Tzeng, Brandon Lloyd, and John D Owens. 2012. A GPU Task-Parallel Model with Dependency Resolution. *Computer* 45, 8 (2012), 0034–41.
[31] Philippe Virouleau, Pierrick Brunet, François Broquedis, Nathalie Furmento, Samuel Thibault, Olivier Aumage, and Thierry Gautier. 2014. Evaluation of OpenMP dependent tasks with the KASTORS benchmark suite. In *International Workshop on OpenMP*. Springer, 16–29.
[32] Chao Wang, Junneng Zhang, Xi Li, Aili Wang, and Xuehai Zhou. 2016. Hardware Implementation on FPGA for Task-Level Parallel Dataflow Execution Engine. *IEEE Transactions on Parallel and Distributed Systems* 27, 8 (2016), 2303–2315.
[33] Jin Wang et al. 2016. LaPerm: Locality Aware Scheduler for Dynamic Parallelism on GPUs.. In *International Symposium of Computer Architecture (ISCA)*.
[34] Jin Wang, Norm Rubin, Albert Sidelnik, and Sudhakar Yalamanchili. 2016. Dynamic thread block launch: A lightweight execution mechanism to support irregular applications on gpus. *ACM SIGARCH Computer Architecture News* 43, 3, 528–540.
[35] Michael Wolfe. 1986. Loops skewing: The wavefront method revisited. *International Journal of Parallel Programming* 15, 4 (1986), 279–293.
[36] Shucai Xiao and Wu-chun Feng. 2010. Inter-block GPU communication via fast barrier synchronization. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 1–12.
[37] Shengen Yan, Guoping Long, and Yunquan Zhang. 2013. StreamScan: Fast Scan Algorithms for GPUs Without Global Barrier Synchronization. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*. ACM, New York, NY, USA, 229–238.
[38] Yi Yang and Huiyang Zhou. 2014. CUDA-NP: realizing nested thread-level parallelism in GPGPU applications. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 93–106.