TECHNIQUES FOR OPTIMIZING DYNAMIC PARALLELISM ON GRAPHICS PROCESSING UNITS

BY

IZZAT EL HAJJ

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

ABSTRACT

Dynamic parallelism is a feature of general purpose graphics processing units (GPUs) whereby threads running on a GPU can spawn other threads without CPU intervention. This feature is useful for programming applications with nested parallelism where threads executing in parallel may each identify additional work that can itself be parallelized. Unfortunately, current GPU microarchitectures do not efficiently support using dynamic parallelism for accelerating applications with nested parallelism due to the high overhead of grid launches, the limited number of grids that can execute simultaneously, and the limited supported depth of the dynamic call stack.

The compiler techniques presented herein improve the performance of applications with nested parallelism that use dynamic parallelism by mitigating the aforementioned microarchitectural limitations. Horizontal aggregation fuses grids launched by threads in the same warp, block, or grid into a single aggregated grid, thereby reducing the total number of grids launched and increasing the amount of work per grid to improve occupancy. Vertical aggregation fuses grids down the call stack with their descendant grids, again reducing the total number of grids launched but also reducing the depth of the call stack and removing grid launches from the application's critical path.

Evaluation of these compiler techniques shows that they result in substantial performance improvement over regular dynamic parallelism for benchmarks representing common nested parallelism patterns. This observation has held true for multiple architecture generations, showing the continued relevance of these techniques.

This work shows that to make dynamic parallelism practical for accelerating applications with nested parallelism, compiler transformations can be used to aggregate dynamically launched grids, thereby amortizing their launch overhead and improving their occupancy, without the need for additional hardware support.

To Mama, Baba, Hind, Mohamad, Mona, and Amal

ACKNOWLEDGMENTS

All praise is due to God.

I would like to thank my advisor, Professor Wen-mei Hwu, who has been an exemplary role model for me in all aspects, whether in his intelligence and integrity as a researcher, his passion and clarity as a teacher, or his humility and compassion as a person. Wen-mei always puts people first and is concerned about their well-being before being concerned about their technical progress, always willing to listen and always willing to help. He continues to inspire us who work with him to strive for high standards of academic excellence and character through the example that he sets.

I would also like to thank Dr. Dejan Milojicic from Hewlett Packard Labs who has been the equivalent of a second advisor to me. Getting to know Dejan was a turning point in my career because of the opportunities he opened up for me and the confidence he instilled in me by believing in me. Dejan's humility, openness, empathy, and genuine care for others make him a great mentor, a great collaborator, and a great human being.

I would also like to thank the rest of my committee, Professors Steven Lumetta and Deming Chen, for going above and beyond their duty in their responsiveness and the thoroughness of their feedback. My work is much stronger because of their contributions.

I would like to thank the IMPACT Research Group for their feedback and companionship throughout this journey and for being a fun bunch of people. I especially would like to thank Liwen Chang for his brilliance and his eagerness to help others, and Juan Gómez-Luna for his dedication and remarkable work ethic. I would also like to thank Marie-Pierre Lassiva-Moulin for the positive energy she provides.

I am not who I am without my parents, my siblings, my wife, and my daughter. I thank them wholeheartedly for their love and support. They are my biggest blessing.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 BACKGROUND AND MOTIVATION 4
- 2.1 GPU Architecture and Programming Model 4
- 2.2 Nested Parallelism 6
- 2.3 Dynamic Parallelism 8
- 2.4 Dynamic Parallelism Overhead 9

CHAPTER 3 HORIZONTAL AGGREGATION 11
- 3.1 Warp and Block Granularity 13
- 3.2 Grid Granularity 19
- 3.3 Tradeoffs between Levels of Granularity 23
- 3.4 Additional Optimizations 23
- 3.5 Handling of Control Divergence 29

CHAPTER 4 VERTICAL AGGREGATION 31
- 4.1 Launch Promotion 33
- 4.2 Vertical Aggregation 36
- 4.3 Overlapping Parents and Children 38
- 4.4 Combining Vertical Aggregation and Overlapping 39
- 4.5 Advantages and Tradeoffs 41

CHAPTER 5 METHODOLOGY AND IMPLEMENTATION 42
- 5.1 Benchmarks 42
- 5.2 Compiler Implementation 44
- 5.3 Evaluation Platform 44
- 5.4 Profiling 45

CHAPTER 6 HORIZONTAL AGGREGATION EVALUATION . . . 46
- 6.1 Performance without Optimizations 46
- 6.2 Scalarization of Uniform Arguments 51
- 6.3 Scalarization of Uniform Configurations 53
- 6.4 More Efficient Aggregation Logic at Grid Granularity 53
- 6.5 Aggregation of Dynamic Memory Allocation 55
- 6.6 Performance with Optimizations 55

CHAPTER 7 VERTICAL AGGREGATION EVALUATION 59
7.1 Performance 59
7.2 Profiling 62

CHAPTER 8 CASE STUDY: TRIANGLE COUNTING 63
8.1 Application Overview 63
8.2 Baseline Implementation 63
8.3 Dynamic Parallelism Implementation 65
8.4 Performance Evaluation 67

CHAPTER 9 RELATED WORK 76
9.1 Applications using Dynamic Parallelism 76
9.2 Dynamic Parallelism Optimizations 80
9.3 Parallelization and Optimization of Recursion and Producer-Consumer Chains 83
9.4 Other Trends in GPU Computing 83

CHAPTER 10 CONCLUSION 87

REFERENCES 89

CHAPTER 1

INTRODUCTION

Over the past decade, since the introduction of CUDA [1], graphics processing units (GPUs) have proven effective as general purpose accelerators. The most straightforward targets for GPU acceleration have been applications with parallel computation patterns that are flat and regular in nature, and are thus a good match for GPU architectures. However, as GPUs become more mainstream as processors for general purpose acceleration, there is an increased interest in using them to accelerate a wider range of computation patterns. Of particular interest are computation patterns with nested parallelism.

Nested parallelism is a computation pattern in which an application has multiple levels of parallelism that can be exploited. The application typically creates multiple workers to span the first level of parallelism, then each of those workers does a certain amount of serial work, discovers more parallel work, and creates multiple workers to span that next level of parallelism. This process can happen recursively across multiple levels of nesting.

One example of applications that exhibit nested parallelism is graph traversal [2] where the first level of parallelism consists of nodes at the current frontier and the second level of parallelism consists of those nodes' neighbors. Another example is mesh refinement trees [3] where bounding boxes at one level of parallelism are recursively divided into multiple sub-divisions at subsequent levels of parallelism based on the amount of activity happening in those boxes. Finally, some applications with inter-worker dependences can be expressed using nested parallelism for convenience and ease of programming. For example, producer-consumer chains in which a consumer worker must wait for a producer worker to partially execute before it can begin its execution can be expressed as though the consumer worker is nested in the producer worker. Such producer-consumer chains are common in data sliding algorithms [4] such as multi-dimensional array padding/unpadding as well as stream compaction.

To enable intuitive programming of nested parallelism computation patterns on GPUs, a feature called *dynamic parallelism* [5, 6] was introduced whereby threads running on the GPU can spawn other threads without CPU intervention. Dynamic parallelism improves developer productivity and code maintainability [2, 7, 8, 9] by allowing developers to express nested parallelism computation patterns more naturally. For example, in graph traversal algorithms, a thread visiting a node in the current frontier can launch other threads to visit that node's neighbors. Without dynamic parallelism, the thread would have to loop over those neighbors sequentially, capturing less parallelism, or use other queuing-based techniques, which can be complicated and error prone. Similarly, in data sliding algorithms, a producer can launch consumers when their dependences have been satisfied rather than the programmer having to manually write complicated synchronization code using atomic operations.

While dynamic parallelism is an attractive feature for ease of programming, it has seen limited adoption in practice due to its poor performance [7, 8, 10]. One limitation is that dynamic parallelism on current hardware incurs high overhead when grids are launched from a GPU [7, 8]. Another limitation is that there can be a limited number of grids that can execute at the same time [7, 10]. These limitations are exacerbated when threads in a parent grid (which are typically large in number) each launch a small child grid. In this case, child grids are launched without enough work to amortize the launch overhead, and the grid count limit is reached before the GPU is fully utilized. Another limitation of dynamic parallelism is the bound on the depth of the call stack, which is problematic for computation patterns with deep recursion [7], such as those that express producer-consumer chains using nested parallelism. All these limitations have led application developers to resort to other programming techniques to avoid the use of dynamic parallelism [8, 11, 12, 13].

There have been multiple efforts to address the inefficiency of dynamic parallelism, both in hardware and software. Proposed hardware approaches include coalescing dynamically launched thread blocks with existing grids on the fly [10], co-locating child grids with their parents for better locality [14], and providing feedback on the profitability of a launch before launching [15]. However, these hardware proposals are not available on current hardware.

Proposed software approaches for addressing the inefficiency of dynamic parallelism include overprovisioning parent grids with slave threads [8] or reusing parent threads/blocks [16] to execute the nested parallel work. These approaches eliminate dynamic grid launches entirely by using mega-kernels that inline the code of all kernels participating in the call hierarchy into a single one that is supplied with maximal resources. The latter approach also uses persistent threads to keep parent threads/blocks active so that they can be reused in the future. In contrast, the compiler techniques presented herein avoid the use of mega-kernels and persistent threads by continuing to use dynamic grid launches for executing nested parallel work, but controlling the granularity of those launches.

The compiler techniques presented herein aim to reduce the overhead of device-side grid launches that use dynamic parallelism by fusing child grids together to amortize their launch overhead. Horizontal aggregation fuses grids launched by threads in the same warp, block, or grid into a single aggregated grid, thereby reducing the total number of grids launched and increasing the amount of work per grid to improve occupancy. Vertical aggregation fuses grids down the call stack with their descendant grids, again reducing the total number of grids launched but also reducing the depth of the call stack and removing grid launches from the application's critical path.

These compiler techniques have been implemented in a real compiler and have been evaluated on three recent generations of NVIDIA GPU architectures: Kepler, Maxwell, and Pascal. They have been evaluated using benchmarks representing common nested parallelism patterns. The evaluation shows that these techniques result in substantial performance improvement over regular dynamic parallelism and continue to be relevant across architecture generations.

This work shows that to make dynamic parallelism practical for accelerating applications with nested parallelism, compiler transformations can be used to aggregate dynamically launched grids, thereby amortizing their launch overhead and improving their occupancy, without the need for additional hardware support.

CHAPTER 2

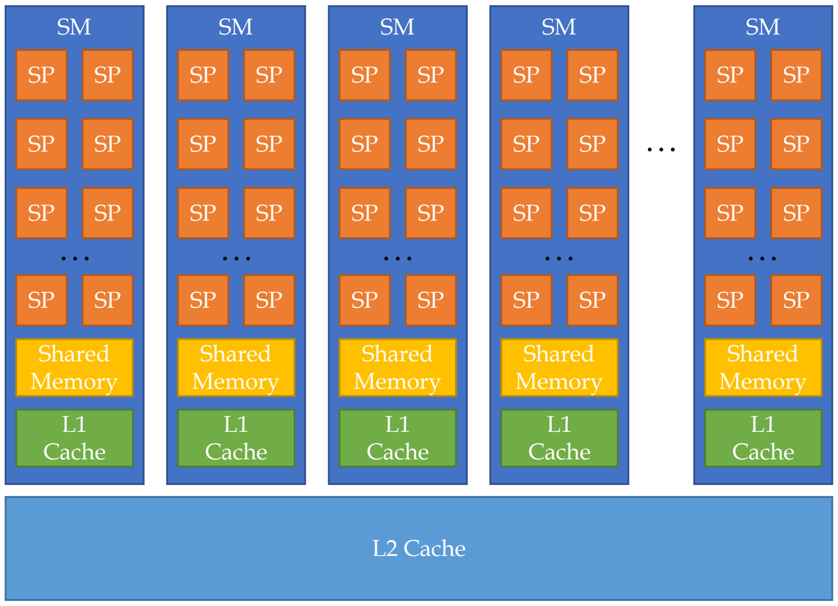
BACKGROUND AND MOTIVATION

Over the past decade, graphics processing units (GPUs) have gained adoption as general purpose accelerators. As GPUs become more mainstream, there is an increased interest in using them to accelerate a wider range of applications such as applications exhibiting nested parallelism. This interest has led to the introduction of a feature called dynamic parallelism. This chapter gives an overview of the basic GPU architecture and programming model (Section 2.1), describes nested parallelism (Section 2.2), describes dynamic parallelism and how it is used to accelerate applications with nested parallelism (Section 2.3), and finally discusses the overhead of dynamic grid launches to motivate the need for compiler optimizations to reduce this overhead (Section 2.4).

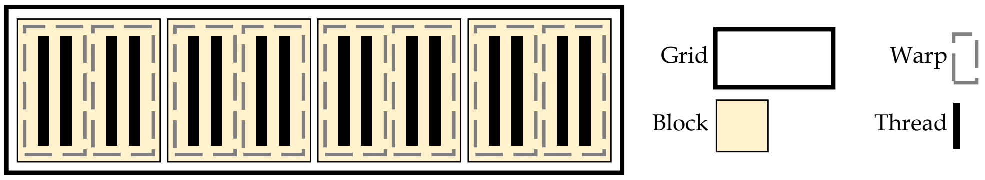
2.1 GPU Architecture and Programming Model

Mainstream CPUs contain a few heavyweight cores which are significantly optimized for single-threaded performance with techniques such as out-of-order execution, branch prediction, and other speculative execution techniques. In contrast, GPUs consist of a large number of lightweight execution cores with coarser-grain control, and are thus optimized for massive parallelism at the expense of single-threaded performance. This section describes the GPU architecture and programming model in detail.

Figure 2.1(a) illustrates a simplified diagram of a typical GPU architecture. The most basic unit of a GPU is the streaming processor (SP), which executes a single thread of instructions. Multiple SPs are grouped into a streaming multiprocessor (SM). The number of SPs per SM varies significantly across architectures: 32, 192, 128, and 64 for Fermi, Kepler, Maxwell, and Pascal architectures, respectively. An SM contains a *shared memory* structure and



(a) GPU Architecture Overview



(b) CUDA Kernel Overview

Figure 2.1: GPU Architecture and Programming Model

an L1 cache that are private to the SM and shared across all SPs in the SM. The shared memory is managed by the programmer, while the L1 cache is managed by the hardware. The GPU is comprised of many SMs that all share an L2 cache. The number of SMs per GPU also varies significantly across architectures: 16, 15, 24, and 56 for Fermi, Kepler, Maxwell, and Pascal architectures, respectively.

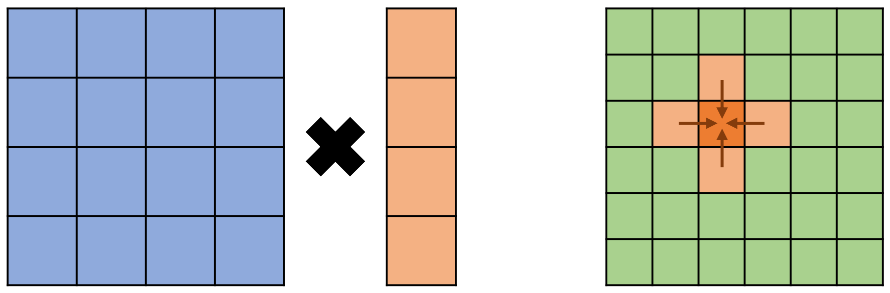
The structure of a CUDA program running on a GPU is illustrated in Figure 2.1(b). The smallest unit of work in a CUDA program is a thread, which executes a sequential stream of instructions on a single SP. Threads are grouped into warps such that threads in the same warp logically execute in lock-step (SIMD) on different SPs of the same SM. There are typically 32 threads per warp, which has been the case for all architecture generations to date, but that value is a characteristic of the hardware and could change in the future. At the next level, warps are grouped into thread blocks.

Threads/warps in the same thread block execute on the same SM, can share memory via the shared memory structure, and can perform local barrier synchronizations. Different warps of the same block are co-scheduled on the same SM, which helps hide memory access latency. The number of threads per block is chosen by the application but is limited by the hardware. The maximum number of threads allowed per block depends on the architecture but has been 1,024 threads/block for all recent architecture generations. Finally thread blocks are grouped into grids, the largest unit of work, which runs on the entire GPU. Different thread blocks may execute on different SMs, therefore threads in different thread blocks cannot access the same shared memory structure. The only way for threads in different thread blocks to communicate is through global memory.

Threads in different thread blocks cannot perform barrier synchronization because there is no guarantee that they will all be scheduled simultaneously. The only way to synchronize across all thread blocks is by terminating the grid and launching a new one. This lack of scheduling control limits the types of parallelism that can be expressed on the GPU. Some works have demonstrated how barrier synchronization across all thread blocks can be implemented using atomic operations on global memory [17, 18] and similar techniques have also been recently added to CUDA [19]. However, these techniques require the grid to use (and reuse) a limited number of thread blocks to ensure that they are all simultaneously resident on the GPU in order to synchronize.

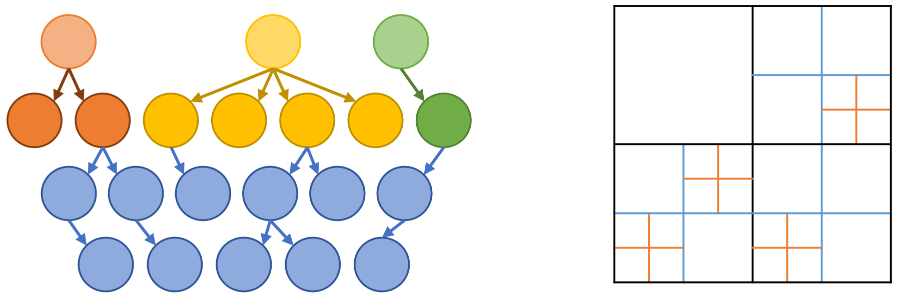
2.2 Nested Parallelism

The grid structure of CUDA programs makes them a good fit for regular computation patterns with a single level of parallelism, i.e., flat parallelism. Figure 2.2(a) illustrates examples of computation patterns with flat parallelism. One example is dense linear algebra operations such as dot-products, dense matrix-vector multiplication, and dense matrix-matrix multiplication. To process these operations, it is typically sufficient to launch a single grid with as many threads as there are elements in the input or output vectors or matrices. Another example of flat parallelism is stencil operations, for which it is typically sufficient to launch a single grid with a thread to process each



Dense Linear Algebra *Stencil*

(a) Computation Patterns with Flat Parallelism



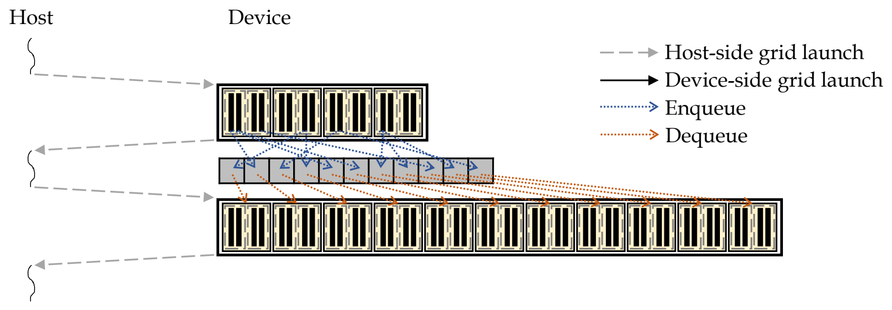
Graph Traversal *Mesh Refinement Trees*

(b) Computation Patterns with Nested Parallelism

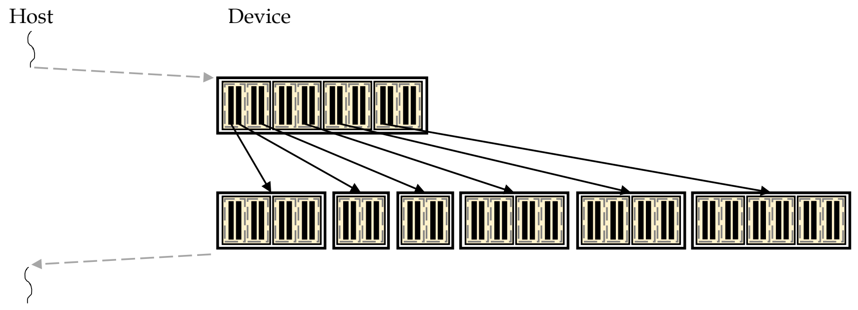
Figure 2.2: Examples of Flat and Nested Parallelism

grid point in the domain.

On the other hand, CUDA grids are unsuitable for computation patterns with multiple levels of parallelism, i.e., nested parallelism. Applications with nested parallelism typically create multiple workers to span one level of parallelism, then each of those workers does a certain amount of serial work, discovers more parallel work, and creates multiple workers to span the next level of parallelism. This process can happen recursively across multiple levels of nesting. Figure 2.2(b) illustrates examples of computation patterns with nested parallelism. One example is graph traversal [2] operations: to capture the parallelism present in these operations, a worker must typically be created for every node in the current frontier of the search, and each worker then creates more workers to visit the neighbors of that node in parallel. Another example of nested parallelism is mesh refinement trees [3]. To capture all the parallelism present in these operations, a worker must typically be created for every bounding box at one level of the tree. Each worker then assesses the amount of activity happening in its box and, if needed,



(a) Programming model without dynamic parallelism



(b) Programming model with dynamic parallelism

Figure 2.3: CUDA Dynamic Parallelism Overview

divides its box into multiple sub-divisions and creates a worker to process each sub-division. This process may continue recursively.

2.3 Dynamic Parallelism

Dynamic parallelism is a feature that was introduced with CUDA 5.0 whereby threads running on a GPU can create other threads without CPU intervention [5]. Similar features were also introduced for other programming models around the same time [6]. Dynamic parallelism enables intuitive programming of nested parallel computation patterns on GPUs, thereby improving developer productivity and code maintainability.

Without dynamic parallelism, there are several ways in which developers can deal with nested parallelism in their applications. The simplest way is to serialize the work of the second level of parallelism within the first. For example, in graph traversal, a thread visiting a node can process its neighbors serially. The drawback of this approach is that it does not extract all the

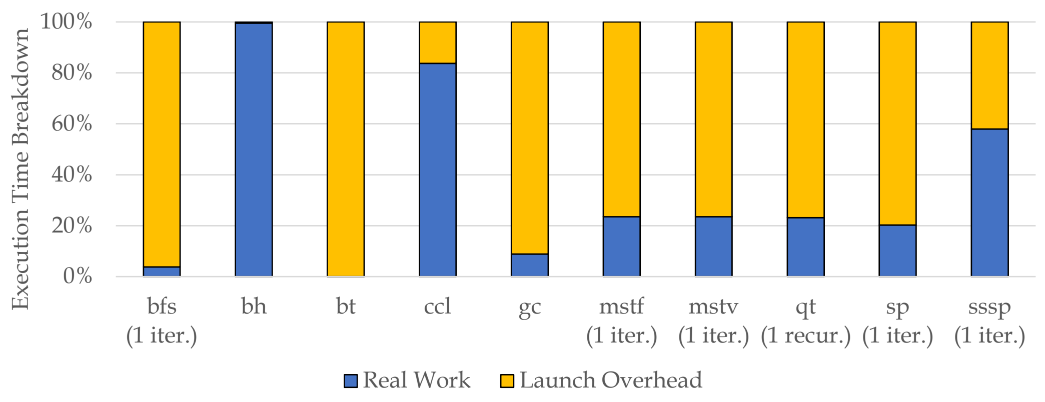


Figure 2.4: Launch Overhead

parallelism available in the application and can suffer from control divergence if adjacent threads have significantly different amounts of nested work. Another way to deal with nested parallelism without dynamic parallelism is to have threads add their nested tasks to queues, then launch a subsequent grid from the CPU based on the number of tasks in the queue. An illustration of this approach is shown in Figure 2.3(a). The drawback of this approach is that it is not an intuitive way to express nested parallelism which makes the code hard to write, read, and maintain.

Dynamic parallelism presents a simple way for developers to express nested parallelism in applications. Threads that discover nested parallel work can simply launch a new grid of threads to execute that work in parallel as shown in Figure 2.3(b). For example, in graph traversal, a thread visiting a node in a graph can launch a new grid with as many threads as it has neighbors after visiting its node and identifying the number of neighbors it has. Similarly, in mesh refinement trees, a thread/block processing a bounding box can launch a new grid with as many threads/blocks as there are sub-divisions after it has discovered whether or not the bounding box needs to be sub-divided for finer resolution.

2.4 Dynamic Parallelism Overhead

In addition to enhancing the ease of programming, one would expect dynamic parallelism to also improve resource utilization (hence, performance) for applications with nested parallelism because it enables launching just the right number of threads needed at each level of parallelism. However, it has been shown that dynamic parallelism has high overheads in practice [7, 8] which has resulted in low adoption.

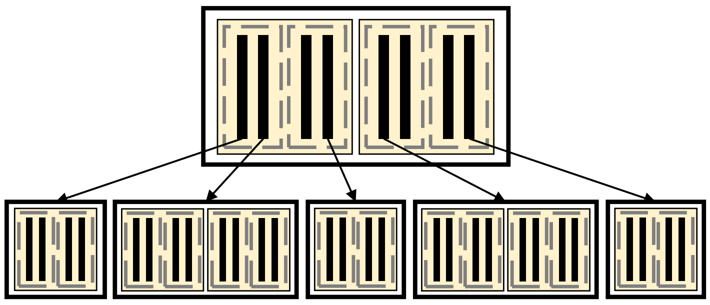
Figure 2.4 shows the breakdown of execution time for some benchmarks that use dynamic parallelism (see Chapter 5 for experimental details). These results demonstrate that workloads using dynamic parallelism can be dominated by the launch overhead. The compiler techniques presented herein aim at reducing this high overhead incurred by dynamic grid launches to make dynamic parallelism more usable in practice.

CHAPTER 3

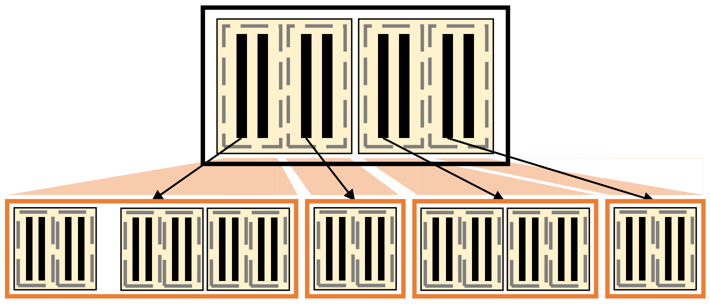
HORIZONTAL AGGREGATION

The primary compiler technique proposed in this work for mitigating the high launch overhead of dynamic parallelism is horizontal aggregation. We use the term *aggregation* in general to refer to a transformation whereby grids that were originally launched by different threads are fused together into a single grid that is launched by a single representative thread. Aggregation can be thought of in various dimensions – threads, call stack, and code sequence – as well as hybrid combinations of these dimensions. Aggregation across threads is the aggregation of grids launched by different threads in the same grid executing the same line of code. This dimension is referred to as *horizontal aggregation*, which is the subject of this chapter. Aggregation across the call stack is the aggregation of grids launched by a thread with the grids to be launched by its children threads in the presence of recursion. This dimension is referred to as *vertical aggregation* and is the subject of Chapter 4. Aggregation across the code sequence is the aggregation of grids launched at different lines of code in a single kernel. This dimension of aggregation is outside the scope of this work.

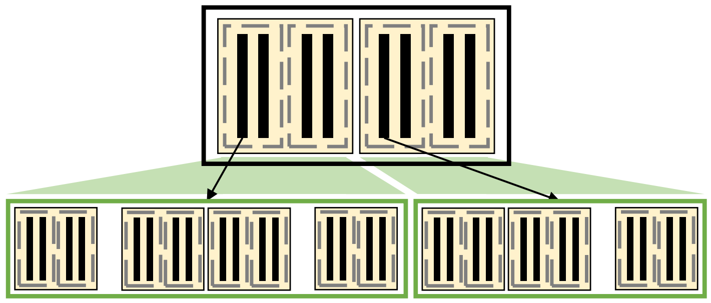
Horizontal aggregation can be performed at various levels of *granularity*, which is the scope of parent threads in the same parent grid across which the launched child grids are aggregated. For example, horizontal aggregation at warp granularity means that grids launched by threads in the same warp are aggregated into a single grid which is launched by one of the threads in that warp. On the other hand, grids launched by threads in different warps remain in separate aggregated grids. This work explores horizontal aggregation at three different levels of granularity: warp, block, and grid. Warp and block granularity is described in Section 3.1, while grid granularity is described in Section 3.2.



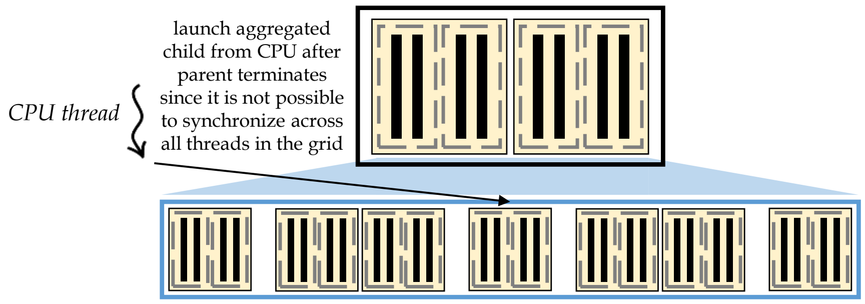
(a) Example of Regular Dynamic Parallelism without Aggregation



(b) Warp-Granularity Horizontal Aggregation



(c) Block-Granularity Horizontal Aggregation


launch aggregated child from CPU after parent terminates since it is not possible to synchronize across all threads in the grid
CPU thread

(d) Kernel-Granularity Horizontal Aggregation

Figure 3.1: Horizontal Aggregation for Different Levels of Granularity

3.1 Warp and Block Granularity

Horizontal aggregation at warp (or block) granularity fuses grids launched by threads in the same warp (or block) into a single aggregated grid launched by a single thread. The transformations necessary for these two levels of granularity are similar, so they are explained together with differences indicated where necessary.

3.1.1 Transformation Overview

Figure 3.1 illustrates the impact of horizontal aggregation at each level of granularity using a toy example. Figure 3.1(a) shows an example of regular dynamic parallelism without any aggregation applied. Five child grid launches are performed by five different threads of the parent grid. Figure 3.1(b) illustrates the transformation that takes place when horizontal aggregation is applied at warp granularity. The first warp in the parent grid originally had two threads each launching a child grid. In the transformed version, the two child grids are aggregated into one aggregated grid, which is launched by one of the two threads in the parent warp. This transformation effectively reduces the number of kernel launches by up to a factor of the warp size. Warps of size two are used in this toy example for simplicity, but warps are typically 32 threads, as explained in Section 2.1.

Figure 3.1(c) illustrates the transformation that takes place when horizontal aggregation is applied at block granularity. Here, only one thread per block launches a kernel on behalf of all the threads in the block. For example, the first block in the original kernel performs three different grid launches by different threads. These three grids are converted into one aggregated grid launched by one thread. This transformation effectively reduces the number of grid launches by up to a factor of the block size, which can be up to 1,024, as explained in Section 2.1.

Figure 3.1(d) illustrates the transformation that takes place when horizontal aggregation is applied at grid granularity. This transformation is discussed in Section 3.2.

```
kernel<<<gD,bD>>>(args)
```

(a) Original Kernel Call

```
__global__ void kernel(params) {
    kernel body
}
```

(b) Original Kernel

```
allocate arrays for args, gD, and bD
store args in arg arrays
store gD in gD array, and bD in bD array
new gD = sum of gD array across warp/block (includes implicit barrier)
new bD = max of bD array across warp/block (includes implicit barrier)
if(threadIdx == launcher thread in warp/block) {
   kernel_agg<<<new gD, new bD>>>
                    (arg arrays, gD array, bD array)
}
```

(c) Transformed Kernel Call (called in a kernel)

```
__global__ void kernel_agg(param arrays, gD array, bD array) {
    calculate index of parent thread
    load params from param arrays
    load actual gridDim/blockDim from gD/bD arrays
    calculate actual blockIdx
    if(threadIdx < actual blockDim) {
        kernel body  (with kernel launches transformed and with
                       using actual gridDim/blockDim/blockIdx)
    }
}
```

(d) Transformed Kernel (called from a kernel)

Figure 3.2: Code Generation for Aggregation at Warp and Block Levels of Granularity

3.1.2 Code Transformation of Kernel Call

The code transformation necessary to perform horizontal aggregation at warp (or block) granularity is shown in Figure 3.2. Pseudocode is used to keep the explanation simple. Figure 3.2(a) shows the original dynamic parallelism kernel call as called from a parent kernel. The kernel is configured with the grid dimension `gD` (number of thread blocks in the launched grid) and the block dimension `bD` (number of threads in a block) as well as passed a set of arguments `args` (line 01). Note that `gD` and `bD` can be arbitrary expressions so their values are not known at compile time.

Figure 3.2(c) shows how kernel calls such as the one in Figure 3.2(a) are transformed to achieve horizontal aggregation at warp (or block) granularity. The first step in the transformed code is for the warp (or block) to allocate global arrays to store the arguments and configurations to be passed to the aggregated kernel (line 05). Since dynamic memory allocation on the device using `cudaMalloc` can be expensive, this overhead is avoided by preallocating a memory pool on the host and passing it to the parent kernel. The parent kernel can therefore perform lightweight allocation of global arrays using atomic operations to grab memory from that pool. The size of the pool can be provided as a compiler option if the default size is too large or too small.

Next, each thread stores its arguments and configurations in the allocated arrays (lines 06-07). The reason the aggregated kernel needs to be passed these arguments and configurations in arrays is that different threads in the original parent kernel may have been passing different arguments and configurations to their child kernels and all these values need to be passed to the aggregated kernel. In practice, not all arguments and configurations vary across parent threads, which presents an opportunity for optimizations, as discussed in Sections 3.4.1 and 3.4.2.

Next, the number of thread blocks in the aggregated grid is calculated (line 08). The number of thread blocks in the aggregated grid is the sum of the number of thread blocks in each of the original grids launched by each thread participating in the aggregation. Likewise, the number of threads per block in the aggregated grid is calculated (line 09). The number of threads per block in the aggregated grid is calculated as the maximum number of threads per block in all child grids launched. The maximum is used is to ensure that all blocks in the aggregated child grid have enough threads to execute their work. However, using the maximum means that some blocks may have more threads than they originally did. To address this situation, these threads are masked out, as shown later.

Finally, one of the threads in the warp (or block) executes a single aggregated kernel launch on behalf of the others (line 10). A synchronization across the threads in the warp (or block) is needed before the launch to ensure that all the threads have completed the preparation of the arguments and configurations. This synchronization is already achieved implicitly during the warp-wide (or block-wide) summation/max. In the aggregated kernel call, the new configurations are used (line 11), arguments are replaced with argument arrays, and arrays containing the configurations for each original child are added (line 12).

3.1.3 Code Transformation of Called Kernel

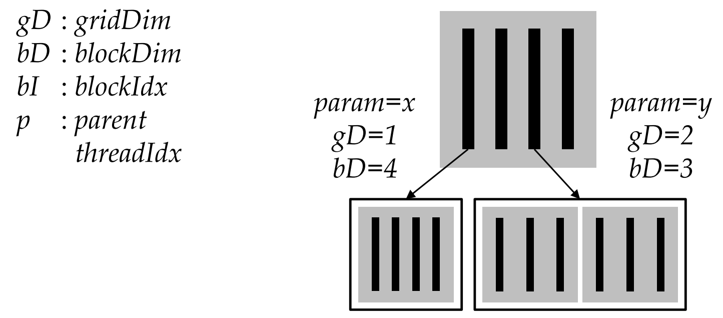
In addition to transforming kernel calls to collect parameters and calculate the new configurations of the aggregated grid, a version of the launched kernel must also be created that identifies its original caller, unpacks its parameters, and calculates its actual configurations accordingly. Figure 3.2(b) shows the original example kernel launched in Figure 3.2(a). The kernel receives a set of parameters `params` and has a kernel body that performs a set of operations using the various parameters as well as the reserved indexing variables `threadIdx`, `blockIdx`, `blockDim`, and `gridDim`.

Figure 3.2(d) shows how the kernel in Figure 3.2(b) is transformed into an aggregated version. First, the function signature of the kernel is transformed (line 14) so that all parameters are converted into parameter arrays that contain the parameters passed by each of the original parent threads. Moreover, configuration arrays are added to the parameter list to pass the number of blocks and threads per block of each of the original non-aggregated grids.

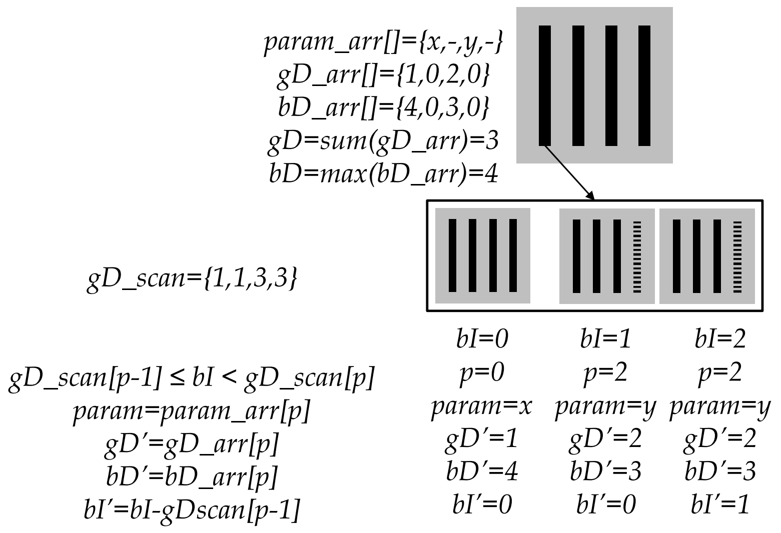
Next, code is added before the kernel body for the block to identify which thread in the parent warp (or block) was its original parent (line 15). For the block to identify its original parent, it needs a scanned (prefix sum) version of the `gD` (gridDim) array to find at which point in the aggregated grid the blocks of each parent thread are. Since all child blocks need to scan the same `gD` array, the scan is actually performed once by the parent warp (or block) before the array is passed to the aggregated child kernel (not shown in Figure 3.2). Conveniently, the scan can be performed while the reduction is taking place to calculate the total number of thread blocks making it incur little additional overhead. With the scanned array of grid dimensions, the child thread block searches the array to identify between which two values its own `blockIdx` falls, thereby identifying the thread index of its original parent thread. The search is performed as follows. First, an initial guess is made that assumes that the child thread blocks in the aggregated grid are evenly distributed across parent threads. This initial guess will be correct in cases where all parent threads launch the same number of child blocks. If the initial guess is incorrect, then an actual search is performed. If the number of threads available in the child thread block exceeds the size of the parent warp/block, then each thread checks one element of the scanned array. The thread which finds the correct position broadcasts the result to the rest of the threads in the thread block using shared memory. If the number of threads available in the child thread block is less than the size of the parent warp/block, then a p-ary [20] search is used.

After identifying its original parent, the block is then able to locate its actual parameters and configurations and load them (lines 16-18). Since the array containing the number of thread blocks in each original grid is scanned, the actual `gD` configuration is recovered by subtracting adjacent scan elements. The block also calculates its actual `blockIdx` within the original non-aggregated grid by subtracting its `blockIdx` in the aggregated grid from the starting point of the original grid in the aggregated grid. A check against the recomputed actual `blockDim` is made to mask out threads that did not exist in the original child grid (line 19). This accounts for the variation in the number of threads per block in the grids before aggregation and the usage of the maximum for all grids after aggregation.

Finally, in the kernel body, all uses of `blockDim` and `blockIdx` are replaced with the actual values computed previously (lines 20-21). Moreover, all kernel launches are also transformed into aggregated kernel launches which supports having multiple levels of nesting.



(a) Original Launch



(b) Transformed Launch

Figure 3.3: Aggregation Logic Example

3.1.4 Aggregation Logic Example

Figure 3.3 shows a toy example of how horizontal aggregation works in practice. In the original kernel launch in Figure 3.3(a), the parent warp (or block) has 4 threads. Parent thread 0 launches a child grid with 1 thread block of 4 threads and passes the value `x` the parameter `param`. Parent thread 2 launches a child grid with 2 thread blocks of 3 threads and passes the value `y` the parameter `param`.

In the transformed version in Figure 3.3(b), the following changes take place. Arrays of length 4 (equal to the number of original parent threads) are allocated for the parameter `param` as well as the kernel launch configurations `gD` and `bD`. Parent threads 0 and 2 store their `param` values `x` and `y` respectively in their corresponding slot in the array for that parameter.

Similarly, they store their values for `gD` and `bD` in the configuration arrays. The sum of the values in the `gD` array is calculated to be 3, which is the number of thread blocks to be launched in the aggregated grid. The maximum of the values in the `bD` array is calculated to be 4, which is the number of threads per block to be launched in the aggregated grid. The `gD` array is actually scanned while it is being summed resulting in the array `gD_scan` where every entry in the array indicates the number of child thread blocks up to and including the parent thread for that position.

In the launched kernel, the first step is for a block to identify the index of its parent thread in the parent warp (or block) denoted by `p`. To do so, the block searches the `gD_scan` array for the index `p` that satisfies the condition $gD\_scan[p-1] \leq bI < gD\_scan[p]$ where bI is the index of the block in the aggregated array. For example, block 1 identifies that its index 1 is greater than or equal to $gD\_scan[1]$ and less than $gD\_scan[2]$; therefore, its parent thread must have been thread 2.

After having identified its parent thread, the thread block loads the actual parameters and configurations from the arrays to be used by the child threads. It also calculates its real block index by subtracting the number of thread blocks up to and including the previous parent thread's thread blocks. For example, block 2, whose parent thread is 2, subtracts $gD\_scan[1] = 1$ from its block index to obtain its actual block index which is 1. Finally, the thread block masks out threads that did not exist in the original child grid. For example, all thread blocks in the aggregated child grid have 4 threads because the children of parent thread 0 (i.e., block 0) need 4 threads to execute. However, the children of parent thread 2 (i.e., blocks 1 and 2) only need 3 threads so they mask out 1 thread based on the recalculated block dimension.

3.2 Grid Granularity

Horizontal aggregation at grid granularity is when grids launched by threads in the same parent grid are fused together into a single grid launched from the host. While aggregation at warp granularity and aggregation at block granularity are fairly similar, aggregation at grid granularity has some significant differences, which are highlighted in this section.

3.2.1 Transformation Overview

Figure 3.1(d) illustrates the transformation that takes place when horizontal aggregation is applied at grid granularity. At this granularity, all the original child grids are aggregated into a single grid. Since no global synchronization on the GPU is allowed across threads in different blocks, a single thread in the parent grid cannot be chosen to launch the grid on behalf of the others because it is not possible for that thread to know when the others are ready. For this reason, the aggregated grid is instead launched from the host after the parent grid terminates. For this transformation to be possible, it must be legal for the grid launch to be postponed until later in the execution. For such postponing to be legal, the transformation requires that the parent kernels do not explicitly synchronize to wait for their child kernels to finish (by default, kernel launches are asynchronous which means that parent execution can continue before the child grid completes). For this reason, aggregation at grid granularity is only supported when there is no explicit synchronization on child grids by the parent grid.

3.2.2 Code Transformation of Kernel Call

Figure 3.4 shows the code transformation applied to perform horizontal aggregation at grid granularity for the same example as in Figure 3.2. Many parts of the transformation are similar to aggregation at warp and block granularity, but there are some key differences.

The main difference in the transformation of the kernel call is that the call is moved to the host function that calls the parent instead of being called from within the kernel. This transformation is shown in Figure 3.4(c), wherein the kernel launch in Figure 3.4(a) is replaced by just the allocation of arrays and storing of parameters and configurations (lines 05-07). After that, once the kernel returns to the host, a new kernel is called to scan the `gD` array and calculate the total sum to get the number of thread blocks in the aggregated grid (line 08). The scan operation must be done by a separate kernel call because it must wait for threads in all thread blocks to finish storing their configurations which requires a global synchronization that is achieved via kernel termination. Similarly, another kernel is called to calculate the maximum number of threads (line 09). Finally, a launchpad

```
kernel<<<gD,bD>>>(args)
```

(a) Original Kernel Call

```
__global__ void kernel(params) {
    kernel body
}
```

(b) Original Kernel

```
On device in place of child kerne launch:
allocate arrays for args, gD, and bD
store args in arg arrays
store gD in gD array, and bD in bD array

On host after parent kernel launch:
new gD = sum of gD array across grid
new bD = max of bD array across grid

kernel_launchpad(new gD, new bD
                  arg arrays, gD array, bD array)

```

(c) Transformed Kernel Call (called in a kernel)

```
__global__ void kernel_agg(param arrays, gD array, bD array) {
    calculate index of parent thread
    load params from param arrays
    load actual gridDim/blockDim from gD/bD arrays
    calculate actual blockIdx
    if(threadIdx < actual blockDim) {
        kernel body  (with kernel launches transformed and with
                       using actual gridDim/blockDim/blockIdx)
    }
}
void kernel_launchpad(gD, bD, param arrays, gD array, bD array) {
    kernel_agg<<<gD, bD>>>(arg arrays, gD array, bD array)
    postponed kernel launches from kernel body (if any)
}
```

(d) Transformed Kernel (called from a kernel)

Figure 3.4: Code Generation for Aggregation at Grid Granularity

function is called which launches the aggregated child grid from the host (line 11-12).

3.2.3 Code Transformation of Called Kernel

The transformation of the called kernel for horizontal aggregation at grid granularity is similar to that at warp and block granularity. However, one key difference is that this kernel is launched using a launchpad function from the host rather than from another kernel. If the child kernel body itself has any kernel calls that are to be aggregated, those kernel calls can be moved to the launchpad function. The usage of a launchpad function is important for handling multiple levels of nesting or recursion. Without the usage of launchpad functions, the compiler must extract all nested calls in the call hierarchy at the call site of the uppermost parent call, which is difficult, especially in the case where there is recursion and the depth of the call hierarchy is not statically known. This problem does not exist at warp and block granularity because the child kernel call is transformed within the parent kernel, not moved outside the parent kernel like with grid granularity.

3.2.4 Limitations

Unlike horizontal aggregation at warp and block granularity, there are some limitations faced at grid granularity due to the inability to perform a grid-wide synchronization to perform the aggregation on the device. The first limitation, which has been mentioned in Section 3.2.1, is that horizontal aggregation at grid granularity requires that the parent thread does not explicitly synchronize (using `cudaDeviceSynchronize()`) on the kernel launch that is being aggregated. The reason is that if the parent thread explicitly synchronized on the kernel launch, then the kernel launch cannot be postponed after the synchronization, and therefore it cannot be taken outside the kernel to be launched from the host.

The second limitation is when child kernel launch takes place inside of a loop. In this case, the number of launches that each parent thread performs may not be known. Handling such a case requires that the number of loop iterations in each thread be tracked so that it can be recovered on the host to

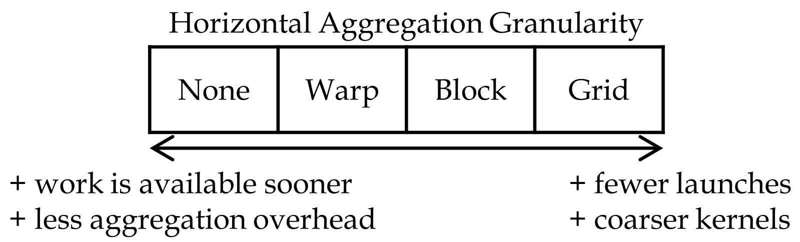

Horizontal Aggregation Granularity
None
Warp
Block
Grid
+ work is available sooner
+ less aggregation overhead
+ fewer launches
+ coarser kernels

Figure 3.5: Aggregation Granularity Tradeoffs

identify how many aggregated child grids to launch. While this is possible, it is not currently supported.

3.3 Tradeoffs between Levels of Granularity

Figure 3.5 shows the tradeoffs associated with varying the level of granularity at which horizontal aggregation is performed. Coarser granularity has the advantage of having fewer launches, hence fewer cycles wasted due to launch overhead, and larger grids, hence better resource utilization due to the availability of more thread blocks to schedule at a time. For example, at grid granularity, there will only be a single child grid launch per kernel call, whereas at warp granularity, there will be as many child grid launches as there are warps in the parent grid with at least one thread in that warp performing a dynamic launch.

However, finer granularity has the advantage of work being available sooner because there are fewer parent threads to wait for, and lower aggregation overhead due to smaller scan and max operations that need to be performed and a smaller array that needs to be searched to identify the original parent thread.

3.4 Additional Optimizations

This section describes a few additional optimizations that enhance horizontal aggregation. While the techniques described in this section are not essential for correctness, they result in substantial performance improvements in some cases and are therefore important for the overall performance of this technique. These optimizations include: scalarization of uniform arguments, scalarization of uniform configurations, sparse in-kernel scan of block dimensions at grid granularity, and aggregation of dynamic memory allocation.

3.4.1 Scalarization of Uniform Arguments

Recall that when grids are aggregated, the arguments passed to the original grids are stored in arrays to be passed to the aggregated grid as shown in Figure 3.2 line 06. However, it is often the case that some of these arguments are uniform across all parent threads. In this case, storing duplicate values of these configurations and arguments is a waste of space and, more importantly, a waste of time spent storing them to global memory in the parent threads and then loading them from global memory in the child grids. Instead, as an optimization, no arrays are allocated for uniform arguments, and they are passed as scalar values to the child grid.

To support this optimization, it is necessary for the compiler to analyze the code to identify which arguments are uniform and which ones are not. This analysis is done via standard iterative dataflow analysis techniques that iterate over the code until convergence. The following are the main rules applied by the analysis:

- Constant literals are uniform.
- Reserved words `blockDim` and `gridDim` are uniform.
- Reserved words `threadIdx` and `blockIdx` are non-uniform.
- A unary, binary, or ternary operation is uniform if all its operands are uniform, and non-uniform otherwise.
- An array access is uniform if the base pointer and index are both uniform, and non-uniform otherwise.
- An object member variable access is uniform if the base object is uniform, and non-uniform otherwise.
- A function call return value is conservatively considered non-uniform unless the function is a pure function with uniform arguments.
- The LHS of an assignment statement is uniform if its RHS is uniform and the assignment takes place in a non-divergent context.

- A control structure is non-divergent if its condition is uniform and divergent otherwise.

Note that the analysis also gives information about when code is in a divergent or non-divergent context. This information is useful for handling control divergence as described in Section 3.5. Such analyses on variable uniformity and control structure divergence are useful in research and tools that involve transformation and optimization of GPU code [21, 22, 23, 24].

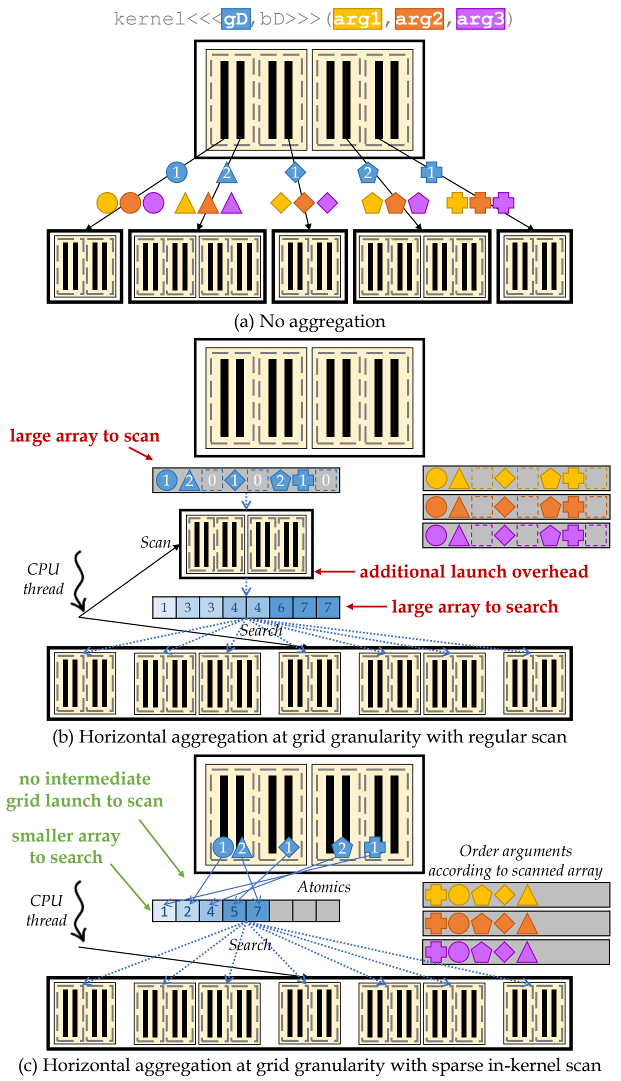
3.4.2 Scalarization of Uniform Configurations

Recall that when grids are aggregated, the configurations passed to the original grids are stored in arrays to be passed to the aggregated grid as shown in Figure 3.2 line 07. This passing of arrays can be avoided if the configurations are uniform, similar to what is done for uniform arguments as discussed in Section 3.4.1. Moreover, if the block dimension configuration is uniform, the maximum operation over the child block dimensions is unnecessary and can be eliminated. In this case, there is no need to pass any parameter for the child block dimension since the child can obtain its block dimension from the `blockDim` reserved word.

3.4.3 Sparse In-kernel Scan of Block Dimensions at Grid Granularity

One of the most expensive operations in the aggregation logic is the scan operation which is used to calculate the grid dimension of the aggregated grid and also to provide an array which the child thread blocks can search to identify their original parent. While the cost of these operations remains reasonable at warp and block granularity, it can become prohibitive at grid granularity for two reasons:

1. At grid granularity, the scan operation must be performed in a separate grid to ensure global synchronization which incurs the overhead of an additional grid launch and does not benefit from the grid dimension values being in the cache as is the case at warp and block granularity.


kernel<<<gD,bD>>>(arg1,arg2,arg3)
large array to scan
Scan
CPU thread
additional launch overhead
large array to search
Search
no intermediate grid launch to scan
smaller array to search
Atomics
Order arguments according to scanned array

(a) No aggregation

(b) Horizontal aggregation at grid granularity with regular scan

(c) Horizontal aggregation at grid granularity with sparse in-kernel scan

Figure 3.6: Sparse In-kernel Atomic-based Scan of Block Dimensions at Grid Granularity

2. While the widths of the scanned array for warp and block granularity are 32 and $\leq$ 1,024 respectively, the width can be much greater at grid granularity since a parent grid may contain many more threads. This larger array results in a larger scan operation after the parent terminates and a larger search operation by each child block, which can incur high latency.

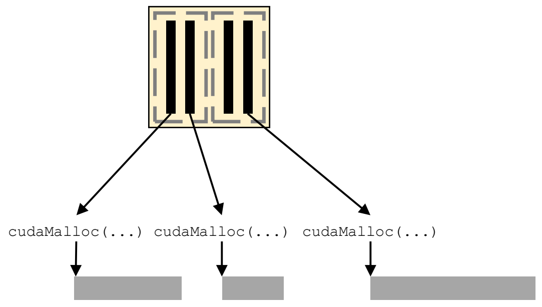
These drawbacks are illustrated in Figure 3.6(a).

To mitigate this overhead, a different technique is used at grid granularity to scan the grid dimension values as illustrated in Figure 3.6(b). Instead of performing a grid-wide scan after all threads have stored their values, atomic operations can be used to calculate the total number of thread blocks, as well as the number of thread blocks up to a certain thread for each parent thread. Using this approach, parent threads will likely not access the atomic operation in the same order as their order in the parent thread block so the “scanned” array will not be in that order. However, this change in order is acceptable because what matters is that whatever order parent threads use to store the cumulative grid dimensions, this same order is used to store arguments and configurations in their respective arrays. Using this technique, the “scan” operation can be performed on the fly in the parent grid without the need for a separate grid launch.

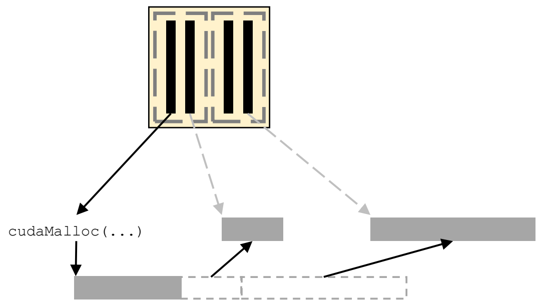
An additional advantage of performing the scan operation in this way is that zeros can be eliminated from the array by the parent thread simply not grabbing an entry in the array if it does not perform a launch (as opposed to storing a zero). By doing so, the array becomes much smaller when the child launches are sparse, resulting in a faster search operation by the blocks in the aggregated child grid. Child launches are sparse when the launch is guarded by some condition that is not true for all parent threads.

3.4.4 Aggregation of Dynamic Memory Allocation

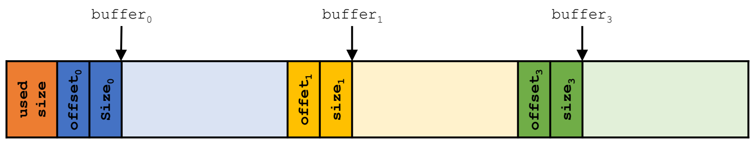
In the same way that grid launches performed by every thread incur high overhead and aggregating these launches amortizes that overhead, dynamic memory allocation by every thread also incurs high overhead and can be aggregated. When every thread in the grid calls `cudaMalloc` for a small amount of memory as shown in Figure 3.7(a), they put a lot of pressure on the device to service these requests. Instead, calls to `cudaMalloc` are aggregated as shown in Figure 3.7(b) by: (1) summing the total requested memory allocation across all threads in the scope of aggregation, (2) using one thread to allocate that total memory on behalf of the others, then (3) redistributing the allocated memory to all threads. It is only possible to aggregate `cudaMalloc` at warp and block granularity since `cudaMalloc` is a blocking call and blocking calls cannot be aggregated at grid granularity as explained in Section 3.2.4. This observation is not a limitation since `cudaMalloc` can be aggregated at a different granularity than other launches in the same kernel so it will not block grid granularity aggregation from being



(a) Dynamic Memory Allocation without Aggregation



(b) Dynamic Memory Allocation with Aggregation



(c) Aggregated Buffer Metadata

Figure 3.7: Aggregation of Dynamic Memory Allocation

applied elsewhere.

One important consideration when aggregating calls to `cudaMalloc` is ensuring that the corresponding calls to `cudaFree` behave correctly. Simply partitioning the allocated buffer across threads is not viable because the corresponding calls to `cudaFree` will either attempt to free in the middle of the buffer which will fail, or free the entire buffer which would deallocate data that may still be in use by other threads. For this reason, the aggregated call to `cudaMalloc` not only partitions the allocated buffer across threads, but also stores some metadata in the buffer that is used by a transformed version of `cudaFree` to perform the deallocation correctly.

Figure 3.7(c) shows the metadata stored in the buffer containing the aggregated allocations. Upon allocation, metadata is initialized at the beginning of each sub-buffer in the aggregated buffer containing the `size` of the sub-buffer as well as its `offset` from the beginning of the buffer. Metadata is

```
A
if(condition) {
    B
    kernel<<<gD, bD>>>(arg1, arg2, arg3);
    C
}
D
```

(a) Original Divergent Kernel

```
A
__gD = 0
if(condition) {
    B
    __gD = gD;
    C
}
kernel<<<__gD, bD>>>(arg1, arg2, arg3);
D
```

(b) Divergence Elimination by Postponing Kernel Launch

Figure 3.8: Handling of Control Divergence

also initialized at the beginning of the entire buffer containing the `used size` which is the amount of space in the buffer that is still being used (initially, all of it). When `cudaFree` is called on each of the sub-buffers, the transformed call to `cudaFree` looks up the `offset` of the sub-buffer being freed, uses it to identify the location of the `used size` variable, and atomically decrements that variable by `size`. If `used size` reaches zero, then all the sub-buffers in the buffer have been freed, so the thread that last atomically decremented the counter then proceeds to free the entire buffer.

3.5 Handling of Control Divergence

For horizontal aggregation at warp and block granularity, all threads in the warp or block must be active to participate in the aggregation operations. These operations include the scan and maximum operations necessary to compute the aggregated grid's configurations. Moreover, the designated launcher thread must always be active to allocate from the memory pool as well as launch the aggregated grid on behalf of the other threads. This requirement does not exist when aggregation is applied at grid granularity since the scan and maximum operations are done in separate kernels or using atomics and the aggregated launch is performed from the host after the kernel terminates.

The requirement that all threads are active to participate in aggregation operations is violated in the presence of control divergence. For this reason, the parent kernel is transformed to ensure that the child kernel is called at a point in the program when all threads are active. Therefore, the child kernel call is postponed to a later point in the parent kernel where there is no control divergence. These points in the program can be identified using the uniform variable analysis and divergence analysis described in Section 3.4.1. The transformation also requires that the kernel's launch configurations and arguments are preserved for use at the later launch point. An example of this transformation is shown in Figure 3.8. Since launches are asynchronous by default, performing this transformation is legal provided that there is no explicit synchronization (using `cudaDeviceSynchronize()`) by the parent thread between the launch's original and new location and that the launch is not taken outside of a loop.

CHAPTER 4

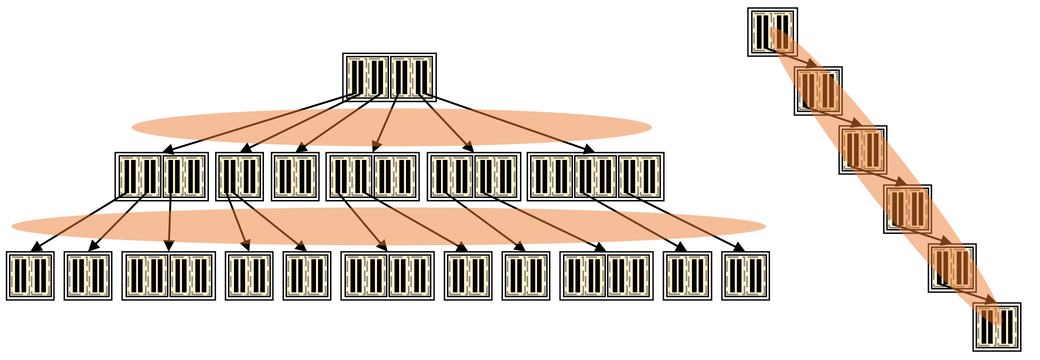
VERTICAL AGGREGATION

The horizontal aggregation transformation is good for wide and short call trees where many threads in the same parent grid launch child grids simultaneously. An example of such a call tree is shown in Figure 4.1(a). However, for tall and narrow call trees, such as that shown in Figure 4.1(b), horizontal aggregation is not useful because there are not many launches at each level of the tree to be aggregated. For this reason, another type of aggregation is proposed, which is vertical aggregation.

Vertical aggregation is a transformation whereby kernel launches are aggregated across different levels of the call tree. In other words, thread blocks of child grids are aggregated with the thread blocks of their parent grids. This requires target kernels to be recursive since aggregation must be applied to launches of the same kernel. In this work, a specific type of recursive kernel is targeted, namely data sliding kernels [4], which are producer-consumer kernels whereby each grid consists of a single thread block that is launched by its producer and that subsequently launches its consumer resulting in the most extreme version of a tall and narrow call tree such as that shown in Figure 4.1(b).

It is possible to implement data sliding kernels without dynamic parallelism using traditional atomic operations for inter-block wait and release flags [4]; however, these kinds of programs are difficult to write. Dynamic parallelism offers a more intuitive way to express these kinds of patterns. However, using dynamic parallelism has several issues: (1) there are many grid launches, (2) there are many small grids in flight, and (3) there is a deep call stack. Vertical aggregation aims to address these issues.

Throughout the explanation in this chapter as well as in the evaluation in Chapter 7, the transformation for vertical aggregation is broken down into multiple steps to better understand the workings of the transformation as well as the breakdown of its performance benefit. These steps are pro-



(a) Wide and short trees suitable for horizontal aggregation
(b) Tall and narrow trees suitable for vertical aggregation

Figure 4.1: Horizontal vs. Vertical Aggregation

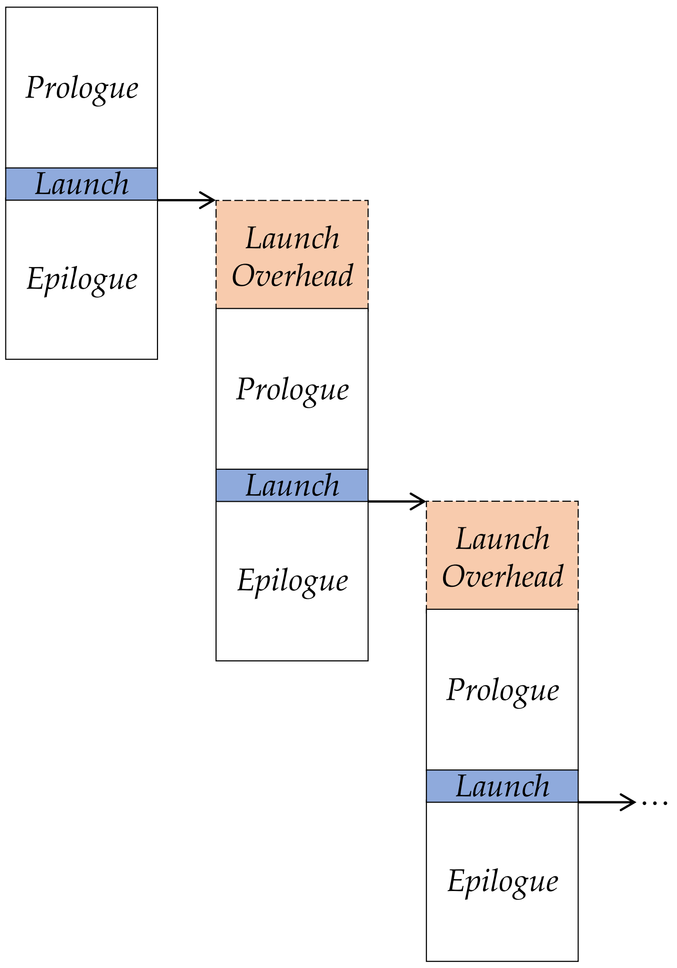


Figure 4.2: Original Program

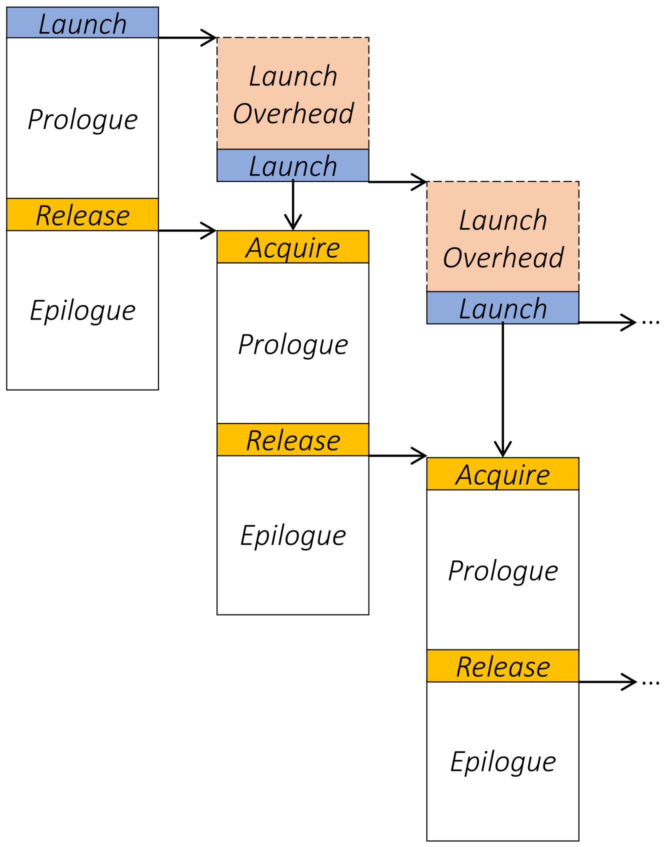


Figure 4.3: Promotion Transformation

motion (Section 4.1), aggregation (Section 4.2), overlap (Section 4.3), and aggregation with overlap (Section 4.4). Each step in the transformation is explained with reference to the baseline program shown in Figure 4.2. In this program, every thread block is considered to execute a prologue, launch the next thread block, and then execute an epilogue.

4.1 Launch Promotion

Promotion is the first step in the transformation whereby recursive kernel calls are hoisted to the beginning of the kernel body. Figure 4.3 illustrates the transformation that takes place when promotion is applied to the example in Figure 4.2. In the transformed code, the kernel call is moved to the beginning of the kernel before the prologue. However, to preserve correct ordering in between the parent and child, code is inserted to *release* a flag that the child will *acquire* to know when it is safe to execute. In all but the first kernel instance (which is launched from the host), code is also inserted before the prologue to acquire the release from the parent. The main advantage of this step is that the launch overhead is removed from the critical path and replaced with a more lightweight release acquire chain. The launch is thus set up by the device runtime and the child thread blocks are scheduled on the SMs while the parent thread block executes the prologue and before it reaches the release operation.

Figure 4.4 shows in pseudocode the code transformation that takes place to achieve kernel launch promotion. The target pattern of the original kernel is shown in Figure 4.4(a). The pattern is such that the kernel executes some prologue code, then selects a single launcher thread to perform a kernel launch, then executes an epilogue code. The kernel launch must be configured to have a single thread block. Moreover, the parameters/arguments of the kernel can be classified into two categories. The first category is the available parameters/arguments which are those that are available at the beginning of the kernel before the prologue. The second category is the postponed parameters/arguments which are those that are not available until after the prologue.

The kernel in Figure 4.4(a) is transformed to two different versions, one called from the host and one called recursively from the device, shown in Figures 4.4(b) and (c) respectively. In the kernel called from the host in Figure 4.4(b), the transformation is as follows. First, code is inserted before the prologue for the launcher thread to perform the launch prematurely. The launcher thread starts by allocating buffers (line 10) to be passed to the child grid where the postponed arguments will be stored later once they become available. The launcher thread also allocates a flag that is used by the parent to communicate with the child to release it when its data is ready (line 11). The child is then launched prematurely (lines 12-13) with the available arguments kept as they are and the postponed arguments replaced with their corresponding buffers. Performing a premature launch requires that any launch condition (not shown in the figure) also be promotable to the beginning of the kernel before the epilogue. In cases where the condition is not promotable because it is dependent on the prologue, the launch can be

$args_{avail}$	:	*arguments available at the beginning of the kernel*
$args_{post}$	:	*arguments whose values are “postponed” because they are not available at the beginning of the kernel*
$params_{avail}$	:	*parameters corresponding to available arguments*
$params_{post}$	:	*parameters corresponding to postponed arguments*

```
__global__ void kernel($params_{avail}$, $params_{post}$) {
    prologue
    if(launcher thread) {
        kernel<<<1,nThreads>>>($args_{avail}$, $args_{post}$)
    }
    epilogue
}
```

(a) Original Kernel

```
__global__ void kernel($params_{avail}$, $params_{post}$) {
    if(launcher thread) {
        allocate postponed arg buffers
        allocate child flag
        kernel_from_kernel<<<1,nThreads>>>
            ($args_{avail}$, postponed arg buffers, child flag)
    }
    prologue
    if(launcher thread) {
        store $args_{post}$ in postponed arg buffers
        memory fence
        set child flag to release child
    }
    epilogue
}
```

(b) Transformed Kernel (called from host)

```
__global__ void kernel_from_kernel($params_{avail}$,
                                   postponed param buffers, flag) {
    if(launcher thread) {
        allocate postponed arg buffers
        allocate child flag
        kernel_from_kernel<<<1,nThreads>>>
            ($args_{avail}$, postponed arg buffers, child flag)
    }
    wait to acquire flag
    load $params_{post}$ from postponed param buffers
    prologue
    if(launcher thread) {
        store $args_{post}$ in postponed arg buffers
        memory fence
        set flag to release child
    }
    epilogue
}
```

(c) Transformed Kernel (called from kernel)

Figure 4.4: Code Generation for Basic Promotion

performed speculatively and an abort signal can be sent instead of a release signal to abort trailing launches.

After the child grid is launched prematurely, the prologue is executed normally. At the end of the prologue, the values of the postponed arguments become available so the launcher thread stores these variables in the postponed argument buffers previously allocated (line 17). A memory fence is then executed to ensure that these values become visible to the child grid before it is released (line 18). Finally, the flag is set to be released (line 19) which signals to the child that it is safe to start executing. The release is implemented in CUDA as a non-cached store while the acquire is implemented as a polling loop with a volatile load. In OpenCL 2.0 [25], it is also possible to use built-in support for release and acquire. After the child is released, the execution of the epilogue continues normally.

The second version of the transformed kernel is the one intended to be launched from the device. The transformation is shown in Figure 4.4(c). The transformation is largely similar to that in Figure 4.4(b) with a few essential differences. One difference is that postponed parameters are replaced with parameter buffers and a flag parameter is also added to the parameter list (line 24). Another difference is that between the premature child launch and the prologue, a wait is inserted to make sure that a child thread block waits for its parent thread block to release it before executing its prologue (lines 31). After the thread block is released, it also loads the postponed parameters from the parameter buffers.

4.2 Vertical Aggregation

After kernel launches are promoted, they pave the way for two orthogonal optimizations: vertical aggregation and overlap of independent portions of the prologue. This section discusses how promotion enables vertical aggregation to be applied.

Figure 4.5 illustrates the transformation that takes place when aggregation is applied to the promoted kernel launch. The main difference between this version and the previous version with just promotion illustrated in Figure 4.3 is that the chain of single thread block launches is replaced with a single launch of a large pool of descendant thread blocks. The launching

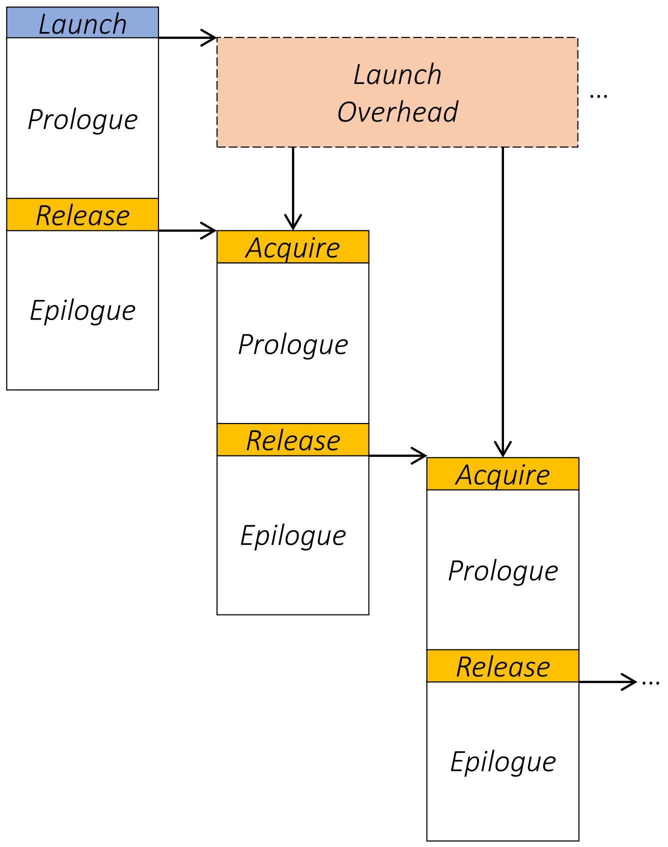


Figure 4.5: Promotion with Vertical Aggregation Transformation

thread block also allocates buffers and flags for all of its descendants in the aggregated grid as opposed to just its immediate descendant.

The size of the aggregated descendant pool defines the granularity of vertical aggregation. Large pools imply more aggregation, therefore fewer launches and larger kernels with better occupancy. They also imply longer chains of thread blocks since the depth of the original call stack is divided by the granularity of aggregation. If the pool of descendants is exhausted, the last descendant in the pool launches a new pool. If the last thread block in the chain is reached and the pool is not yet exhausted, then the final descendant is responsible for sending an abort message for all tail thread blocks in the pool to terminate. However, until the abort message is sent, these thread blocks will occupy resources unnecessarily. Larger pools result in larger tails, hence more unnecessary resource usage and longer tail abortion time.

One important consideration during aggregation is that parent thread blocks in the aggregated pool of descendants must be scheduled before their child thread blocks. Otherwise, deadlock may occur where descendants are scheduled before their ancestors and are waiting for their ancestors to release them, while the ancestors cannot be scheduled because their descendants are occupying the GPU resources. Since software has no control over the order in which thread blocks are scheduled, the static block indices cannot be used. Instead, the index of the thread block within the pool of descendants is assigned dynamically after the thread block is scheduled [26], which ensures that parent thread blocks are always scheduled before their children.

There are multiple benefits of applying the vertical aggregation step. First, the number of grid launches is reduced which results in less launch overhead incurred. Second, the sizes of the grids are increased which results in better utilization of the device. Third, the architectural limitation of the depth of descent is mitigated because the depth of the call stack is divided by a factor equal to the aggregation granularity (size of the descendant pool).

4.3 Overlapping Parents and Children

Another optimization enabled by promotion of kernel launches is the overlap of independent portions of the prologue. This optimization is orthogonal to aggregation so it is described independently in this section. In the next section, aggregation and overlap are combined together.

The overlap optimization is based on the observation that portions of the prologue of a child kernel can be executed independently from the parent. The prologue can thus be divided into two portions: an independent prologue and a dependent prologue. Informally, this independent prologue must satisfy two conditions: (1) it must not use any postponed parameters and (2) it must not have any data dependence with the prologue of the parent. If these two conditions are met, then this portion of the prologue can be executed in parallel with the parent before it releases the child.

The transformation that takes place to achieve the overlap step is shown in Figure 4.6. In this transformation, the prologue is divided into two regions – the independent region and dependent region – and the independent region is hoisted before the acquire logic. If the independent region of the j^{th} block

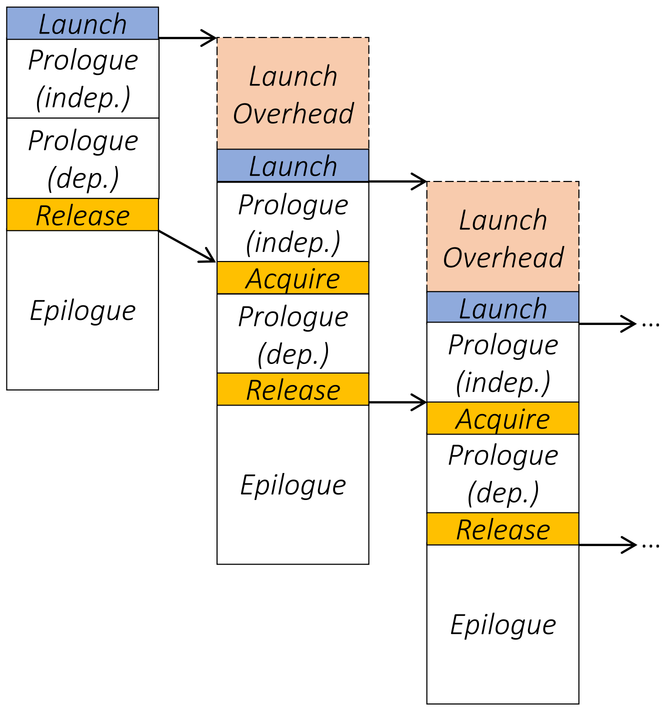


Figure 4.6: Promotion with Overlap Transformation

in the chain is denoted as $P_{i,j}$, the dependent region is denoted as $P_{d,j}$, and the entire prologue is denoted as P_j, then for this transformation to legal, it must satisfy the conditions that: P_j does not write to any memory referenced by $P_{i,j+n}$, and $P_{i,j+n}$ does not write to any memory referenced by P_j where $n > 0$. A simple programmer annotation is used to indicate the boundary between the independent and dependent regions of the prologue. However, dataflow analysis can also be used to detect these regions in many cases.

4.4 Combining Vertical Aggregation and Overlapping

As mentioned earlier, aggregation and overlap are orthogonal optimizations. Putting the three steps together – promotion, aggregation, and overlap –

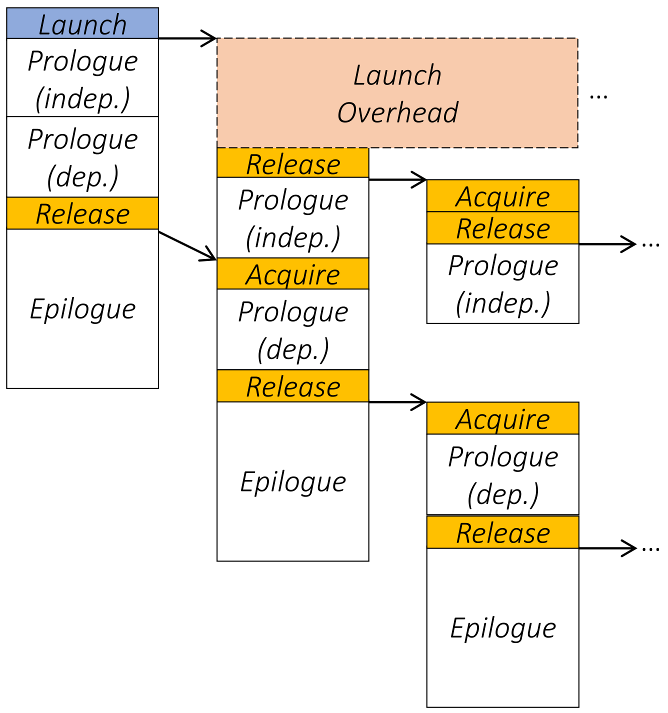


Figure 4.7: Promotion with Vertical Aggregation and Overlap

gives the complete vertical aggregation transformation proposed in this work. This transformation is illustrated in Figure 4.7.

The topmost parent thread block launches a pool of descendant thread blocks and releases its immediate child after it completes its own prologue. The subsequent children synchronize with each other using two release-acquire chains. In the first release-acquire chain, a thread block checks if it must launch a descendant. If yes, it either sends a release signal to the next child in the descendant pool or launches a new pool if the pool has been exhausted. If no, it sends an abort signal to its descendants. If the descendant receives a release, it proceeds. If it receives an abort, it forwards the abort to the rest of its descendants and terminates.

Once a thread block has been released by the first release-acquire chain, it executes the independent part of the prologue and that waits to be released

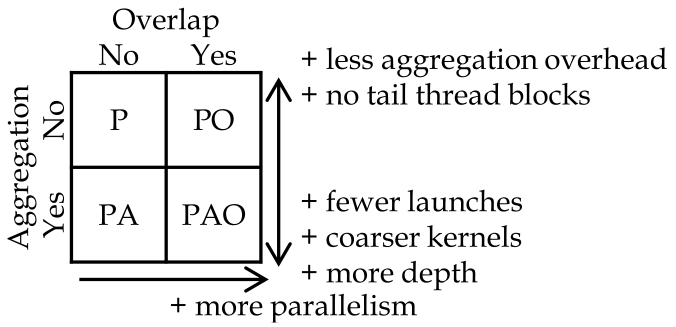


Figure 4.8: Tradeoffs of Vertical Aggregation Steps

by its parent as part of the second chain. Once released by the parent, it executes the dependent part of the prologue, sends a release signal to its child (except for the final thread block in the chain), and then executes its epilogue and terminates.

4.5 Advantages and Tradeoffs

The advantages of promotion compared to no promotion are that it removes the launch from the critical path and replaces it with lightweight release acquire chains. The disadvantages are that it holds onto processing resources for a longer amount of time as child thread blocks busy-wait until they are released by their parents. While these resources are not used by the transformed kernel itself for anything else, they may be useful to other kernels in co-run scenarios.

The advantages and tradeoffs of vertical aggregation and overlap are summarized in Figure 4.8. The advantages of aggregation are that it reduces the number of kernels, increases their granularity, and reduces the depth of the call stack as explained previously. The disadvantage is that it results in additional aggregation overhead as well as unnecessary tail thread blocks that waste resources. The advantage of overlap is that it extracts more parallelism from the long serial dependence chain.

Figure 4.8 also defines acronyms for the different versions of the transformation (P, PA, PO, PAO) based on what combinations of optimizations are applied. The same acronyms are used in the evaluation in Chapter 7.

CHAPTER 5

METHODOLOGY AND IMPLEMENTATION

5.1 Benchmarks

5.1.1 Horizontal Aggregation Benchmarks

Table 5.1 shows the benchmarks and datasets used in the evaluation of the horizontal aggregation transformation. Some of the benchmarks (*bt*, *ccl*, *qt*) use CUDA Dynamic Parallelism (CDP) in the original version of the benchmark. The rest were converted to using CDP from original versions that used nested loops to loop over work sequentially within each thread. For example, in the *bfs* benchmark, the loop over the neighbors of a node is converted to a kernel launch with one thread in the launched grid processing each neighbor.

For all benchmarks, private streams are used by each thread to ensure that launches by threads in the same thread block are not placed in the same stream and serialized which is the default semantic [33]. Using private streams makes the baseline CDP versions faster than serializing in the default stream (geometric mean 1.90× on Kepler and 1.83× on Maxwell). Using private streams is also essential for the benchmarks to be amenable to horizontal aggregation; otherwise, aggregation parallelizes kernels that should be serialized according to the default semantic. Moreover, `cudaDeviceSetLimit` is used to adjust the fixed-size pool of the pending launch buffer appropriately for the baseline CDP version. Without the right size, the cost of overflowing the launch buffer pool penalizes the execution time of the baseline versions [34]. By using private streams and setting the fixed-size pool appropriately, the baseline CDP versions are made as efficient as they can be to ensure a fair comparison.

Table 5.1: Horizontal Aggregation Benchmarks

Name	**Description**	**Dataset**	**Block Sizes**
bfs	Breadth First Search [27]	Random, 10000 nodes 1000 degree	parent=1024 child=32
bh	Barnes Hut Tree [28]	4096 bodies, 4 time-steps	parent=256 child=256
bt	Bezier Lines Tessellation [29]	25600 lines	parent=64 child=32
ccl	Connected Component Labelling [30]	8 frames, 4 host streams	parent=2 child=256
gc	Graph Coloring [31]	1 4096 0.01 bcsstk13.mtx [32]	parent=256 child=256
mstf	Minimum Spanning Tree (find) [28]	rmat12.sym.gr [28]	parent=1024 child=1024
mstv	Minimum Spanning Tree (verify) [28]	rmat12.sym.gr [28]	parent=1024 child=1024
qt	Quadtree [29]	40000 points, 12 depth 1 min.node	parent=128 child=128
sp	Survey Propagation [28]	random-42000-10000-3.cnf [28] 10000 literals	parent=384 child=64
sssp	Single-Source Shortest Path [28]	rmat12.sym.gr [28]	parent=128 child=128

5.1.2 Vertical Aggregation Benchmarks

Different benchmarks are used for evaluating horizontal and vertical aggregation because horizontal aggregation is intended for short and fat trees, while vertical aggregation is intended for long and narrow trees, specifically single thread block chains. Table 5.2 shows the benchmarks and datasets used in the evaluation of vertical aggregation. Benchmarks which require communication between adjacent thread blocks were selected and implemented using CDP.

Each benchmark is tested on three datasets: small, medium, and large. The small dataset is selected to create just two recurrences. The medium dataset is selected to create 25 recurrences (the maximum that can be handled by the GPUs). The large dataset is selected to be the maximum problem size the device can handle or the maximum recurrence that aggregation at granularity 128 can handle (whichever is smaller).

Table 5.2: Vertical Aggregation Benchmarks

Name	Description	Dataset	Block Sizes

los	Line of Sight [29]	small=511
		medium=6399
		large=49407
			parent=256
			child=256

pd	Padding [4]	small=120×120
		medium=450×450
		large=4600×4600
			parent=512
			child=512

pt	Partition [35]	small=16384
		medium=204800
		large=25174016
			parent=512
			child=512

sc	Stream Compaction [36]	small=16384
		medium=204800
		large=25174016
			parent=512
			child=512

unq	Unique [35]	small=16384
		medium=204800
		large=25174016
			parent=512
			child=512

upd	Unpadding [4]	small=120×120
		medium=450×450
		large=4600×4600
			parent=512
			child=512

5.2 Compiler Implementation

The compiler prototype is implemented as a source-to-source (CUDA-to-CUDA) compiler in the Clang infrastructure. Clang version 3.8.0 was used. Although this version does not compile code that uses dynamic parallelism to LLVM IR, the semantic checker was modified to accept kernel calls inside kernel functions for the sake of source-to-source transformation.

5.3 Evaluation Platform

The evaluation was performed on three recent architecture generations of GPUs: Kepler, Maxwell, and Pascal. The Kepler GPU is an NVIDIA Tesla K40c coupled with an 8-core Intel Core i7 950 (3.07 GHz) and uses CUDA SDK 7.5. The Maxwell GPU is an NVIDIA GeForce GTX 980 coupled with an 8-core Intel Core i7 950 (3.07 GHz) and uses CUDA SDK 9.1. The Pascal GPU is an NVIDIA Titan Xp coupled with an 8-core Intel Core i7-7700K (4.20 GHz) and uses CUDA SDK 9.1.

5.4 Profiling

The execution time breakdowns in Figure 6.2 were obtained by incrementally deactivating parts of the code and measuring the resulting time difference. Code regions are deactivated using conditionals that are always false but that cannot be proven so by the compiler. This approach prevents the compiler from performing dead code elimination in the active regions, which would give misleading timing results. For iterative benchmarks with data-dependent convergence criteria (*bfs*, *mstf*, *mstv*, *sp*, *sssp*), only the longest-running iteration was profiled because deactivating code for one iteration changes the behavior of later iterations. Likewise, for recursive kernels (*qt*), only the longest running recurrence was profiled.

For the profiling results in Figure 7.2, performance counters are used from the CUDA Profiler [37] to measure achieved occupancy and executed instruction count.

CHAPTER 6

HORIZONTAL AGGREGATION EVALUATION

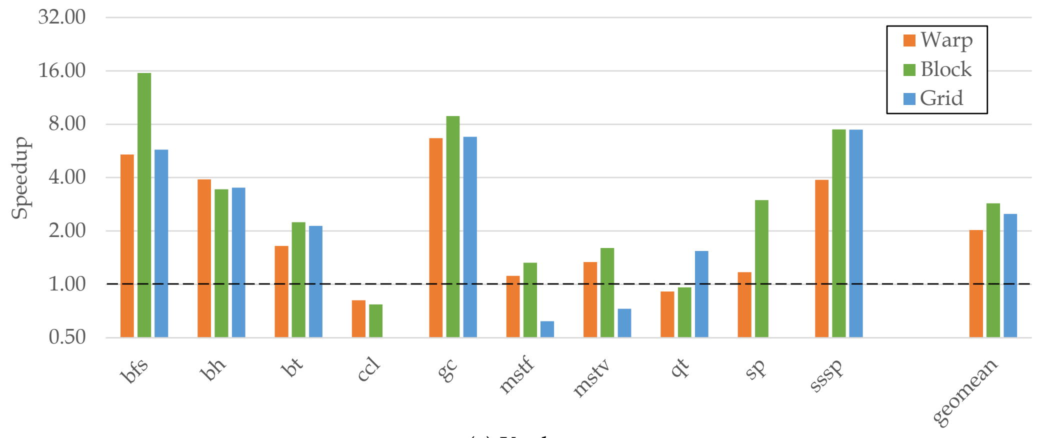
This chapter evaluates the performance improvement from horizontal aggregation at all three levels of granularity: warp, block, and grid. Throughout the chapter, results are shown for Kepler, Maxwell, and Pascal architectures to demonstrate the continued relevance of the horizontal aggregation transformation across architecture generations.

To isolate the impact of individual optimizations on performance, the evaluation is carried out incrementally. First, performance is evaluated without any optimizations in Section 6.1. Next, the incremental performance improvement is evaluated for each optimization including scalarization of uniform arguments (Section 6.2), scalarization of uniform configurations (Section 6.3), the use of more efficient aggregation logic at grid granularity (Section 6.4), and the aggregation of dynamic memory allocation (Section 6.5). Finally, the results are shown with all optimizations included in Section 6.6.

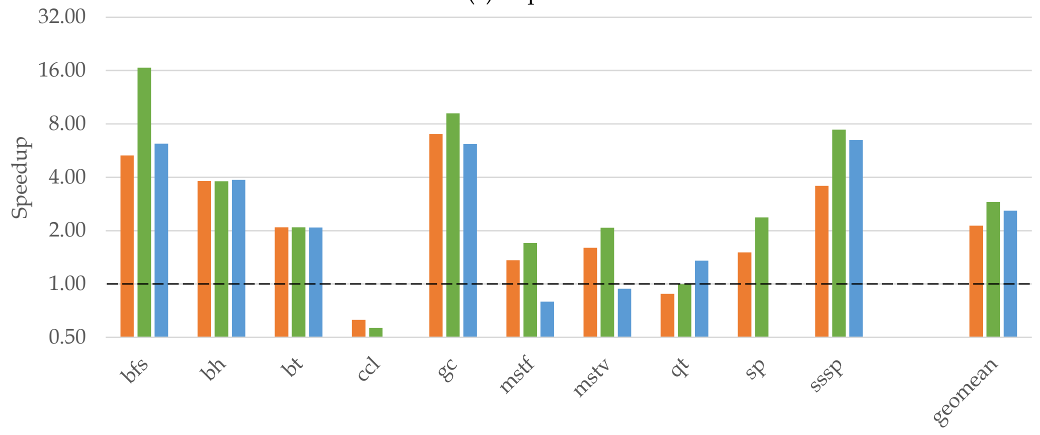
Note that throughout this chapter, two benchmarks do not have versions at grid granularity: *ccl* and *sp*. The reasons are that in *ccl*, the parent thread explicitly synchronizes with the child grid using `cudaDeviceSynchronize()` which prevents the launch from being postponed to the host, whereas in *sp*, the parent thread launches the child grid inside a loop, which cannot be handled. These cases in which grid granularity horizontal aggregation cannot be applied were discussed in Section 3.2.4.

6.1 Performance without Optimizations

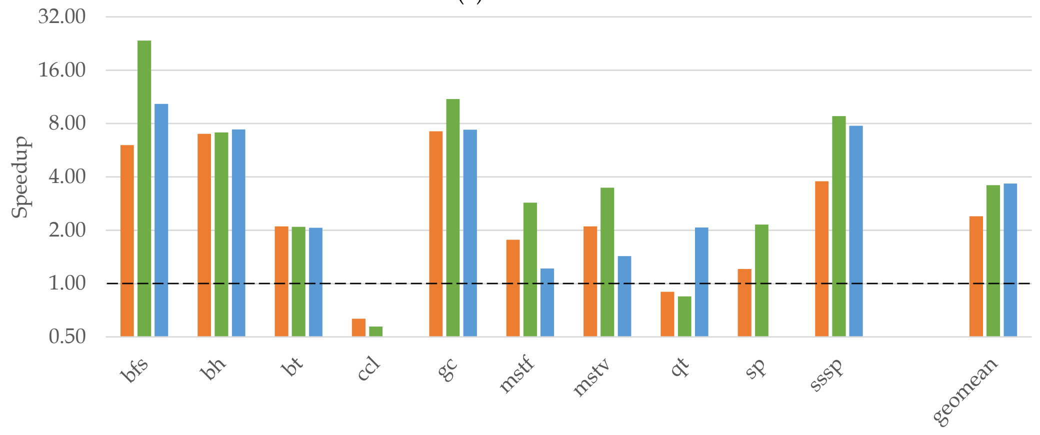
Figure 6.1 shows the overall speedup of unoptimized horizontal aggregation over regular dynamic parallelism for the three levels of granularity across the three architecture generations. Most benchmarks show speedup over the baseline dynamic parallelism versions for all levels of granularity and all



(a) Kepler



(b) Maxwell



(c) Pascal

Figure 6.1: Speedup of Unoptimized Horizontal Aggregation over Regular Dynamic Parallelism

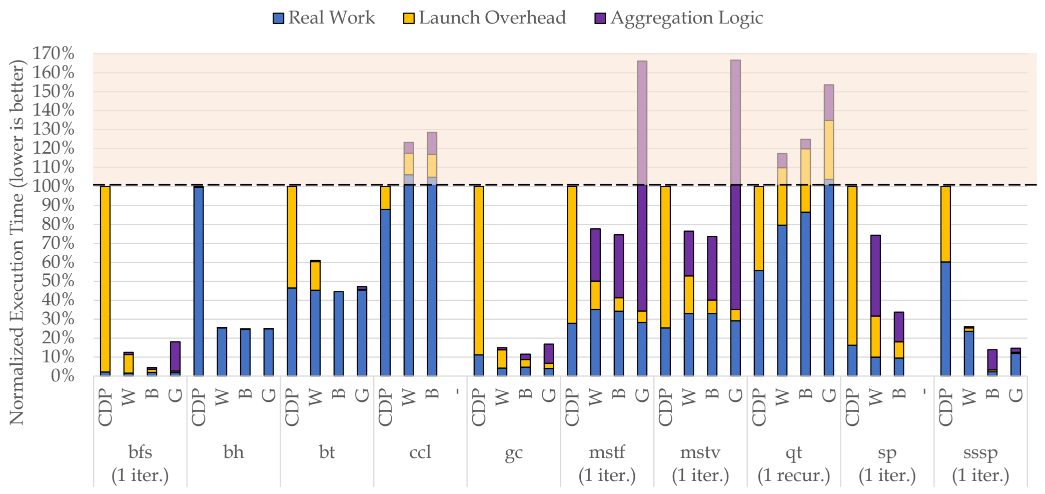


Figure 6.2: Breakdown of Execution Time for Unoptimized Horizontal Aggregation and Regular Dynamic Parallelism

architectures. The advantage of horizontal aggregation holds across all three architecture generations, and is the best on the most recent, which shows that dynamic parallelism overhead continues to be a problem and that horizontal aggregation continues to be a relevant technique.

Figure 6.2 shows the breakdown of the execution time for each benchmark over the original dynamic parallelism (CDP) version as well as horizontal aggregation at each granularity. These results are from the Kepler architecture. In the following paragraphs, the performance of each benchmark is discussed in the context of these profiling results.

bfs, bt, gc, sp, sssp. All five of these benchmarks show significant performance improvement for all levels of aggregation granularity and across all architectures. For all of them, the profiling results show that the original CDP execution time was dominated by the kernel launch overhead, which diminishes as the aggregation granularity increases. Moreover, the amount of time spent doing real work also decreases, which reflects the improvement in occupancy due to having coarser grids with more work available at once. The time spent in the aggregation logic increases with granularity, which is expected since the scan and search operations become longer. At grid granularity, the significant increase in aggregation logic time offsets the marginal reduction in launch overhead which results in a net loss compared to block granularity. This high overhead at grid granularity is optimized in Section 6.4. Of the five benchmarks, *bt* witnesses the least improvement because it continues to be dominated by dynamic memory allocation overhead (which is part of the blue bar). This overhead is mitigated in Section 6.5.

mstf, mstv. These two benchmarks behave similarly to the previous five. The main difference is that the aggregation logic for these benchmarks is much higher. At grid granularity, the overhead is so high that it offsets the benefit of horizontal aggregation resulting in a net slowdown compared to the original dynamic parallelism versions. This slowdown is only for the Kepler and Maxwell architectures, whereas in Pascal it becomes faster because Pascal is a more powerful GPU. This overhead is discussed and mitigated in Section 6.2.

bh. This benchmark contains long running child grids, which amortize the launch overhead. For this reason, the profiling results show that the launch overhead does not dominate performance. Nevertheless, the benchmark still shows significant performance improvement via reduction of the time spent doing real work. This reduction is due to the improvement in occupancy due to having coarser grids, which again highlights the dual benefit of horizontal aggregation in not only reducing the launch overhead but also improving resource utilization.

ccl. This benchmark consists of a parent grid that has a single thread block with two threads. Therefore, there is not much to be gained from aggregation at any granularity. For this reason, this benchmark experiences a slowdown because aggregation logic is being added without any significant reduction in launch overhead. Note that since horizontal aggregation is a compiler transformation, and not a hardware modification, programmers can be selective in applying it. The programmer can simply disable the optimization when it is not beneficial.

qt. In this benchmark, only a single thread in the thread block performs a grid launch. For this reason, there is not much benefit to be expected from aggregation at warp and block granularity similar to *ccl*. Instead, aggregation overhead is unnecessarily incurred resulting in a slowdown. On the other hand, aggregation at grid granularity results in a speedup in Figure 6.1 (despite the profiled recurrence in Figure 6.2 having a slowdown). This benchmark is recursive, which demonstrates the ability of horizontal aggregation to deal with recursion and multi-depth call trees.

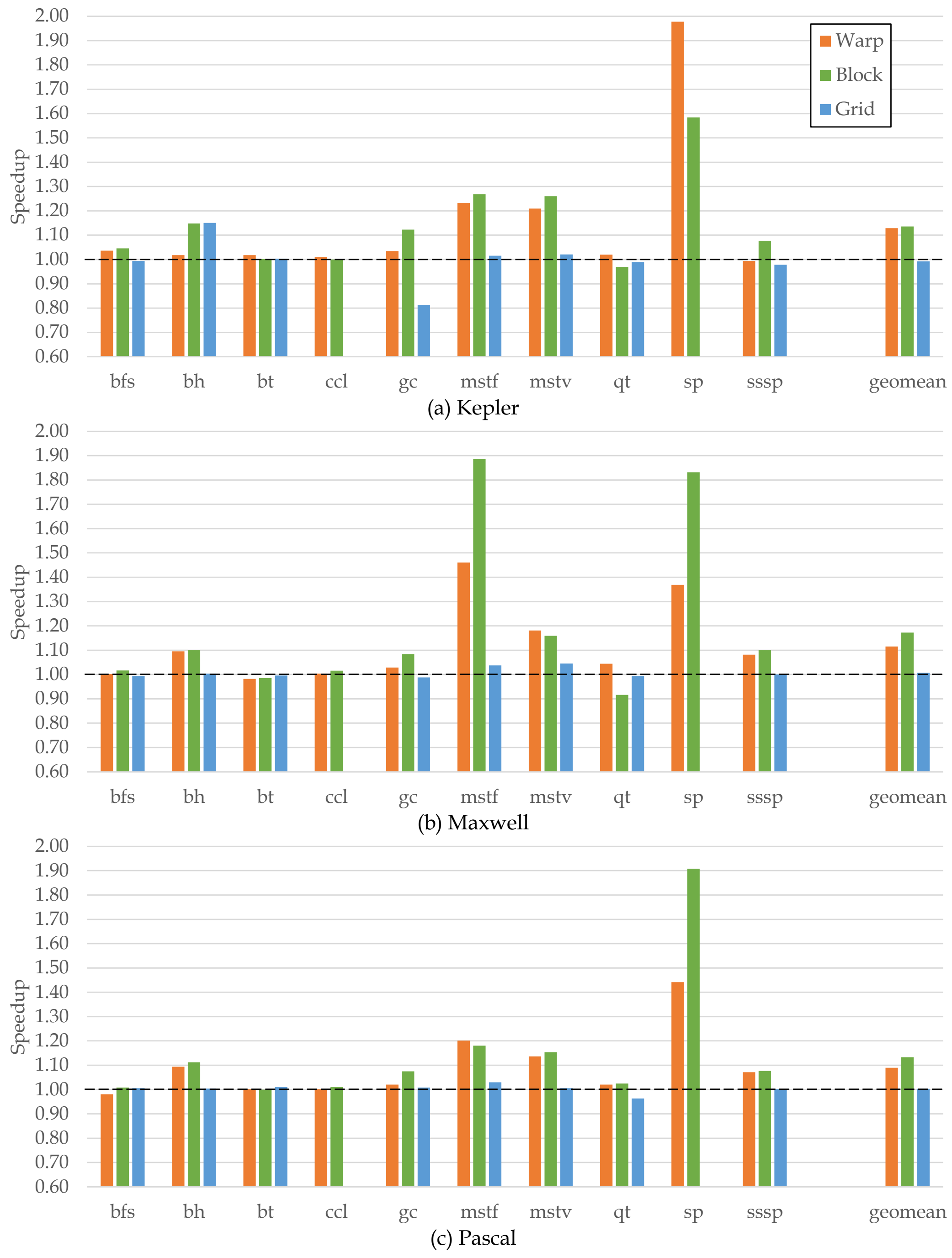


Figure 6.3: Incremental Speedup from Scalarization of Uniform Arguments

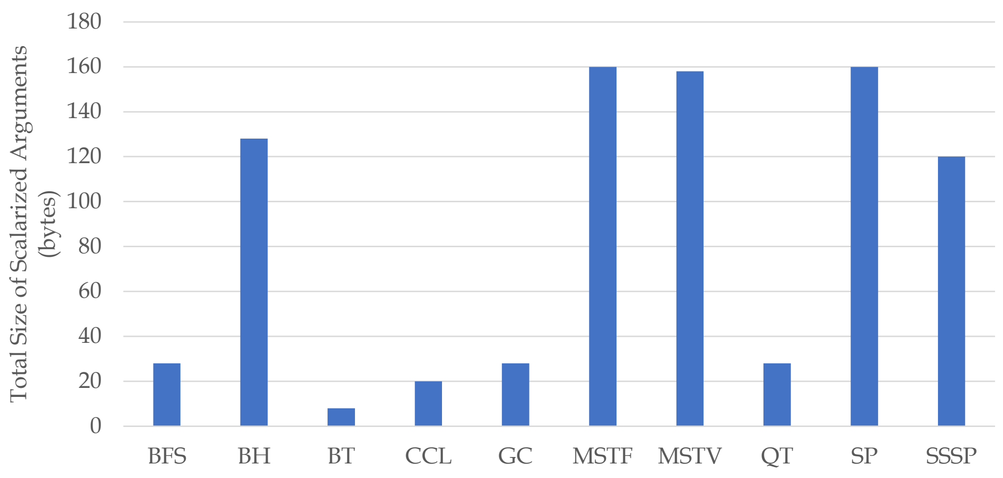


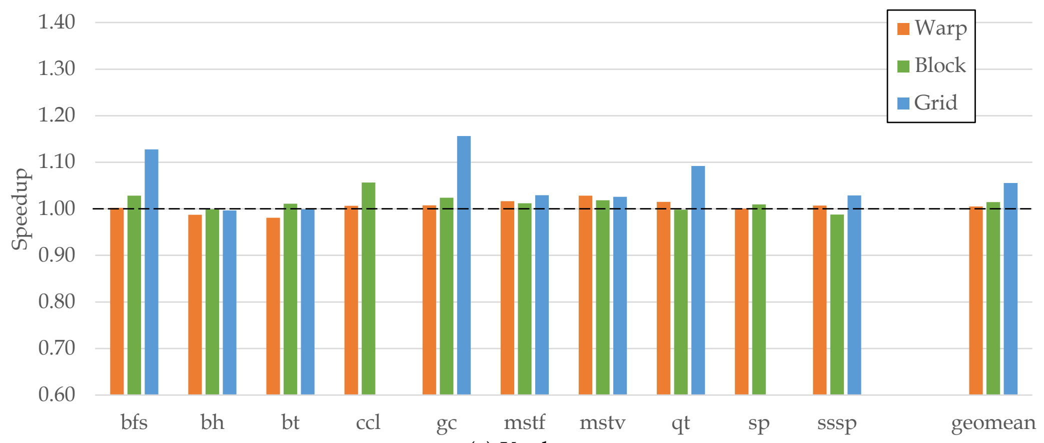
Figure 6.4: Size of Scalarized Parameters for Each Benchmark

6.2 Scalarization of Uniform Arguments

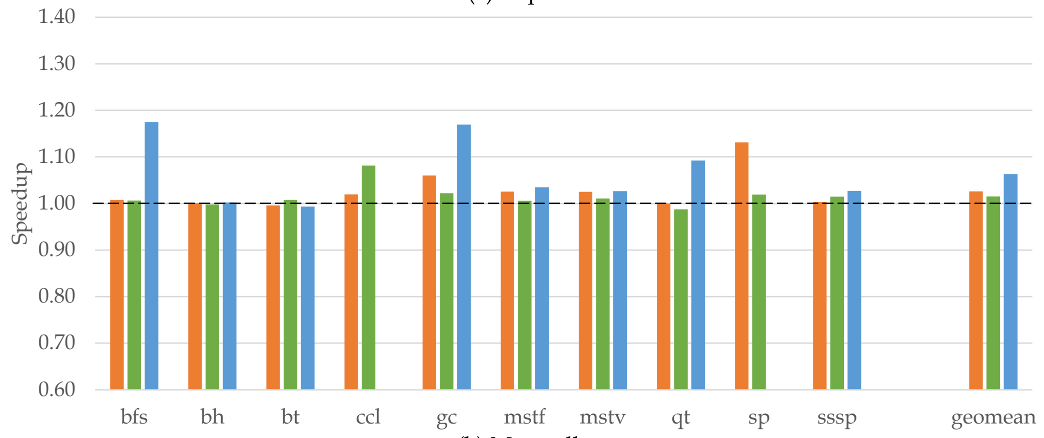
Figure 6.3 shows the incremental speedup of scalarizing uniform arguments over unoptimized horizontal aggregation (which was evaluated in Section 6.1). It is evident that this optimization results in significant speedups for some benchmarks without adversely impacting others.

Figure 6.4 shows the total size of the arguments that are scalarized for each benchmark. An important observation is that the benchmarks showing significant speedups are also those having the largest scalarized arguments. For example, *sp*, the most affected, has 3 scalarized arguments with all 3 being structures with multiple fields (total 160 bytes). *mstf* and *mstv* have 5 and 4 scalarized arguments respectively, both having 2 which are structures with multiple fields (total 160 bytes and 158 bytes respectively). *bh* has all primitive type arguments, but it has 19 of them (total 128 bytes). *sssp* has 3 scalarized arguments with 1 being a structure with multiple fields (total 120 bytes). On the other hand, the benchmarks showing little or no speedup have a small number of scalarized arguments. For example, *bfs*, *gc*, *qt*, *ccl*, and *bt*, have 28, 28, 28, 20, and 8 bytes worth of scalarized arguments respectively.

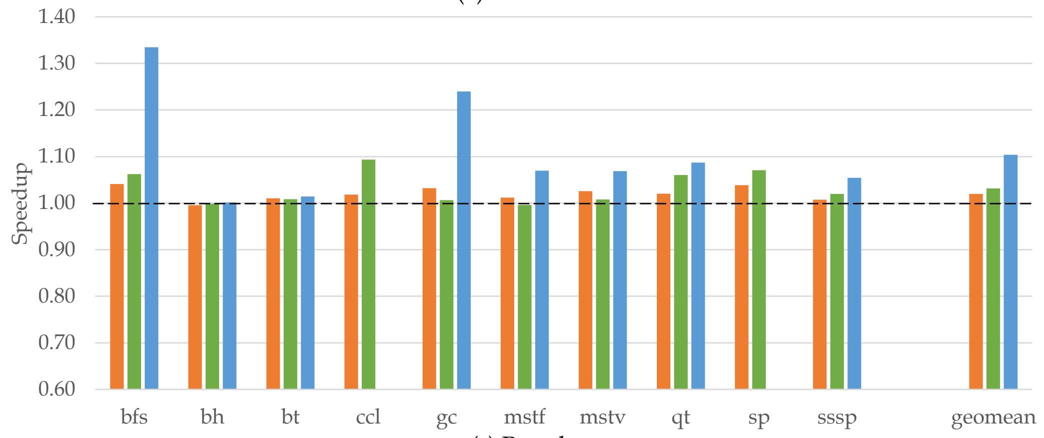
It is interesting to observe that the benchmarks resulting in the most speedups correlate with those having the highest aggregation logic reported in Figure 6.2. This correlation demonstrates that the redundant storage of uniform arguments constitutes a significant portion of the aggregation logic time.



(a) Kepler



(b) Maxwell



(c) Pascal

Figure 6.5: Incremental Speedup from Scalarization of Uniform Configurations

6.3 Scalarization of Uniform Configurations

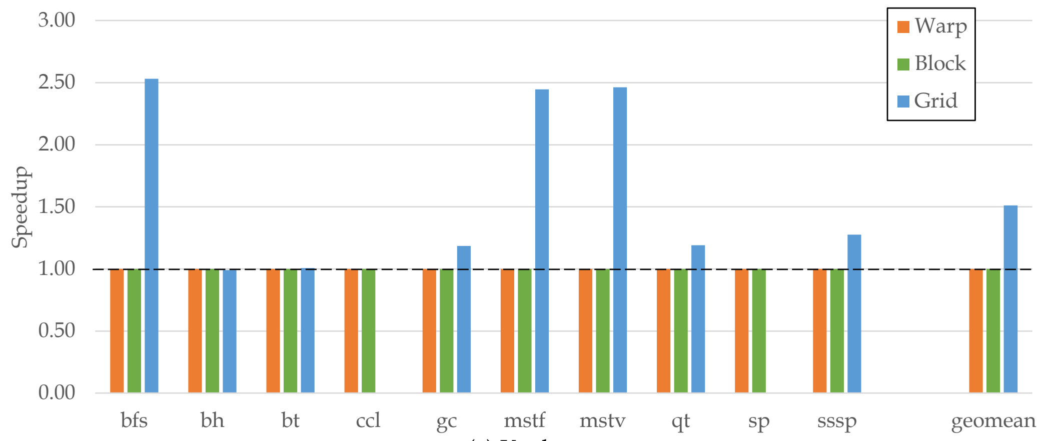
Figure 6.5 shows the incremental speedup of scalarizing uniform configurations over the horizontal aggregation version evaluated in Section 6.2. It is evident that this optimization results in significant speedups for some benchmarks without adversely impacting others.

The most significant speedups for all benchmarks are observed for aggregation at grid granularity. Recall that this optimization eliminates the reduction operations that take place to calculate the maximum block dimension provided by all parent threads. This reduction operation is small at warp and block granularity, but can be large and results in an extra kernel launch at grid granularity which explains why grid granularity witnesses the largest improvement.

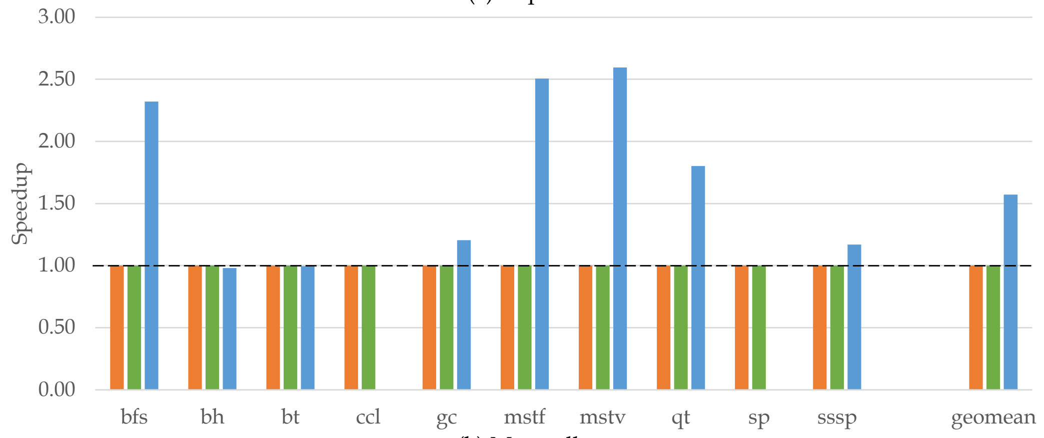
6.4 More Efficient Aggregation Logic at Grid Granularity

Figure 6.6 shows the incremental speedup of employing more efficient aggregation logic at grid granularity over the horizontal aggregation version evaluated in Section 6.3. The speedups are only for aggregation at grid granularity, since this optimization is specific to that granularity. It is evident that this optimization results in significant speedups for some benchmarks without adversely impacting others.

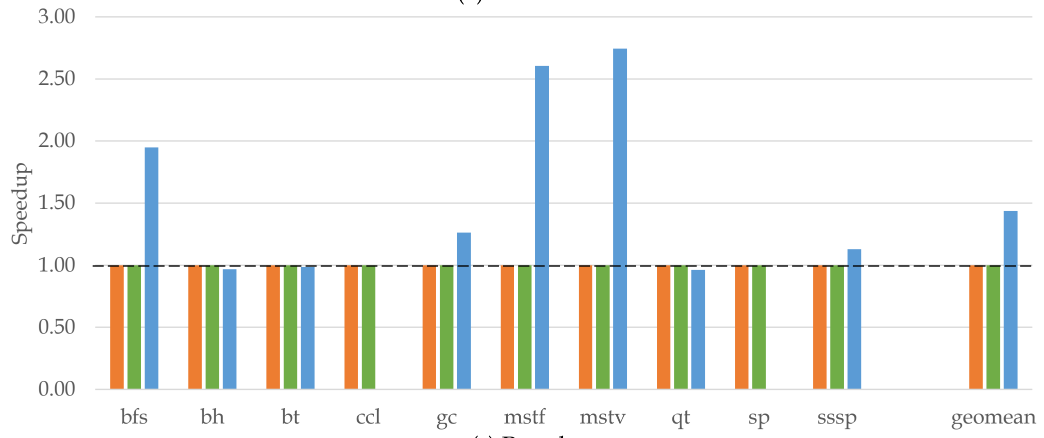
One interesting observation is that this optimization results in the most impressive speedups for *bfs*, *mstf*, and *mstv*. What these benchmarks have in common is that not all parent threads perform a launch, but the launch is guarded by a condition, whereas most of the other benchmarks have all parent threads performing a launch except those at the boundary in the last thread block. Without this optimization, these benchmarks store many zeros in the array containing the number of thread blocks in each child grid. However, with this optimization, these zeros are eliminated, resulting in a shorter search operation in the child grid to identify its parent thread. On the other hand, the other benchmarks do not have this benefit because all threads have non-zero child thread blocks. They only have the benefit of avoiding the extra grid launch to perform the scan operation by replacing it



(a) Kepler



(b) Maxwell



(c) Pascal

Figure 6.6: Incremental Speedup from More Efficient Aggregation Logic at Grid Granularity

with atomics.

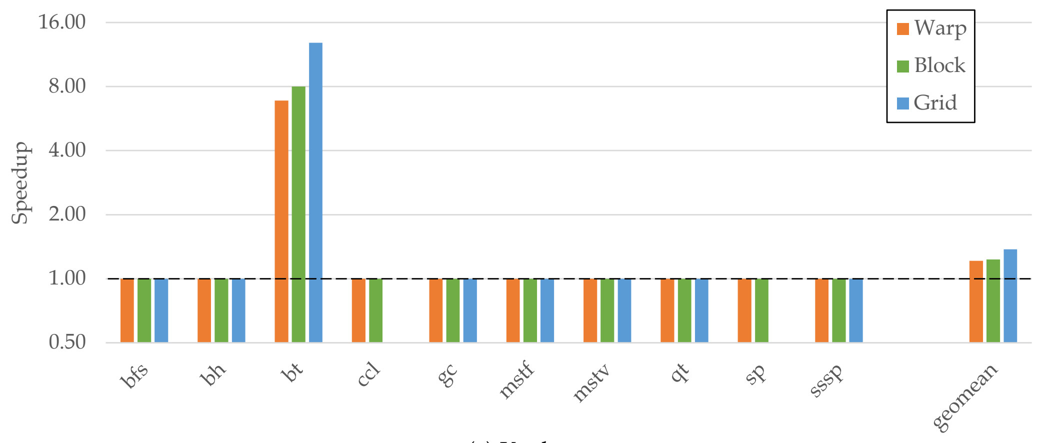
6.5 Aggregation of Dynamic Memory Allocation

Figure 6.7 shows the incremental speedup of aggregating dynamic memory allocation over the horizontal aggregation version evaluated in Section 6.4. The optimization only benefits *bt* since that is the only benchmark that performs dynamic memory allocation. Aggregating dynamic memory allocation results in speedups across all architectures, demonstrating the usefulness of the horizontal aggregation optimization not just to grid launches but to heavyweight runtime functions as well.

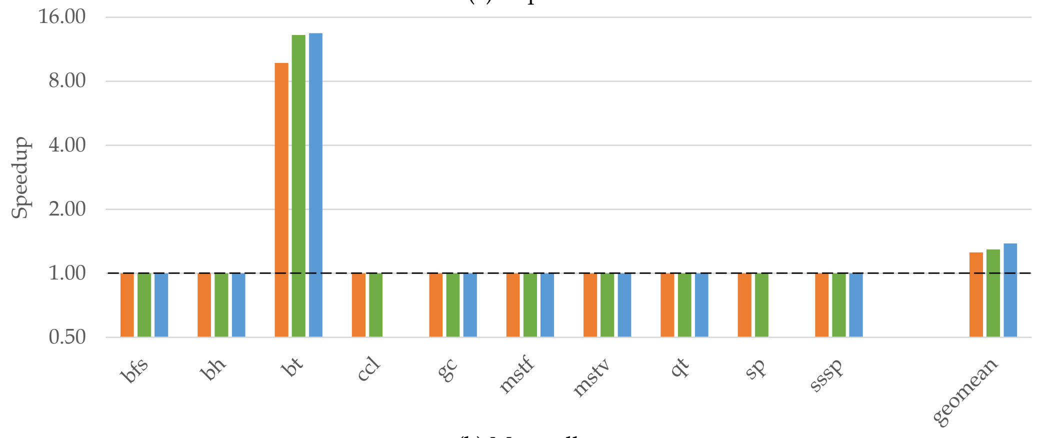
6.6 Performance with Optimizations

Finally, Figure 6.8 shows the combined speedup of all the optimizations put together relative to unoptimized horizontal aggregation, and Figure 6.9 shows the speedup of optimized horizontal aggregation over regular dynamic parallelism. Horizontal aggregation results in impressive speedups at all levels of granularity for all benchmarks across all architectures, with the exception of *ccl* and *qt* at warp and block granularity, which use few launcher threads per block and have little opportunity for aggregation.

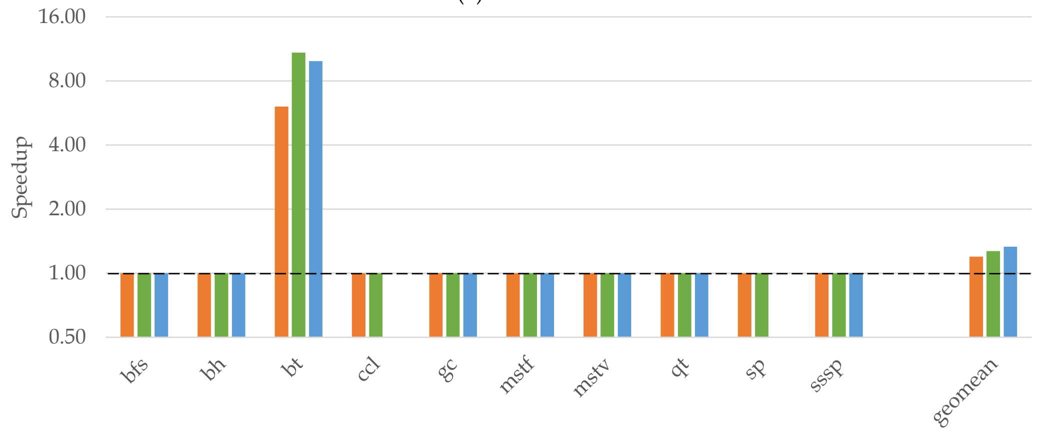
The speedups are sustained with architecture generations, which shows the continued relevance of the horizontal aggregation optimization. In fact, the speedups are higher for later generations. It seems that while GPUs in later generations are becoming more efficient at general purpose computations, there has not been proportional improvement in the performance of dynamic launches. This trend is in the favor of horizontal aggregation which reduces the number of grid launches by trading that off for more work on the GPU to perform the computations required for aggregation.



(a) Kepler

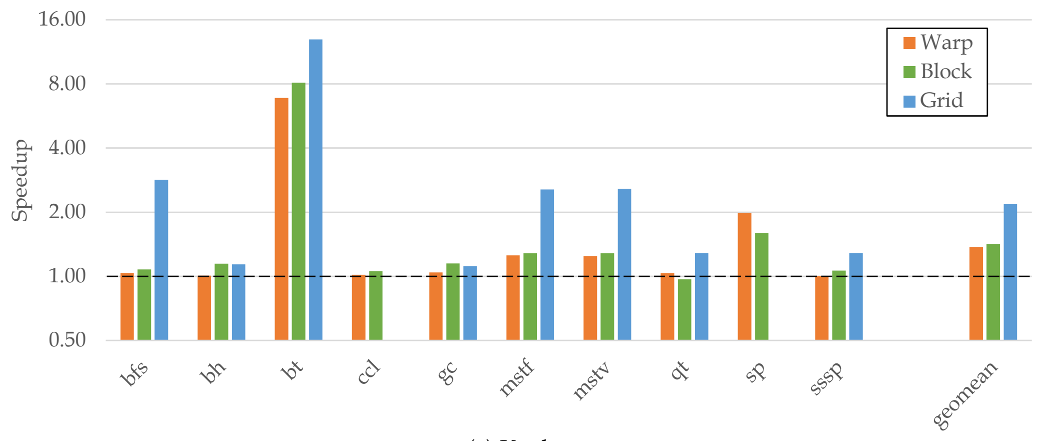


(b) Maxwell

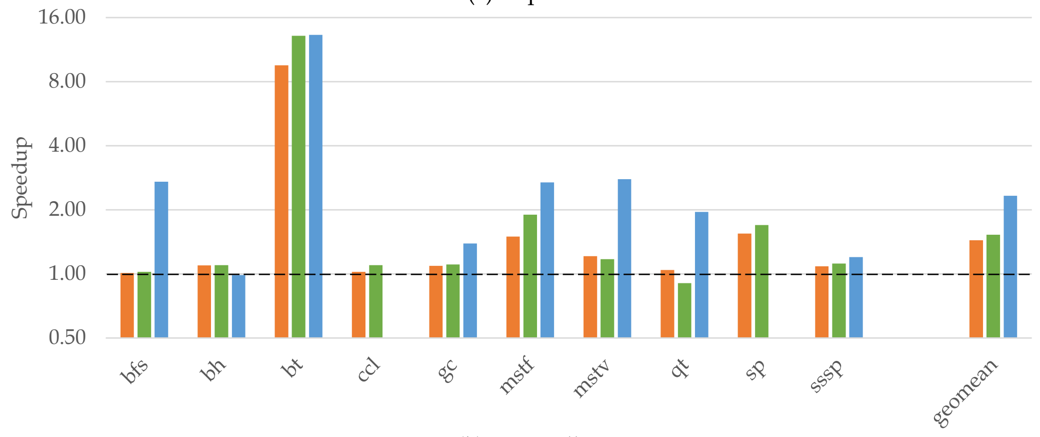


(c) Pascal

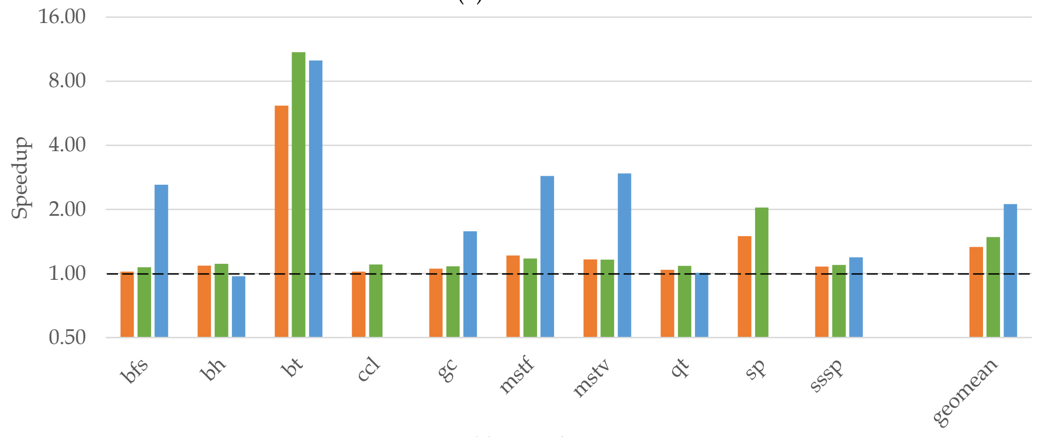
Figure 6.7: Incremental Speedup from Aggregation of Dynamic Memory Allocation



(a) Kepler

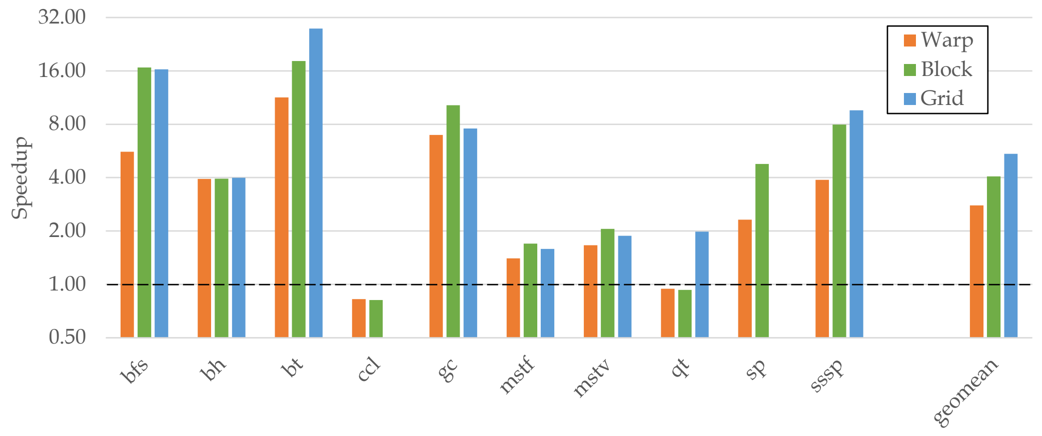


(b) Maxwell

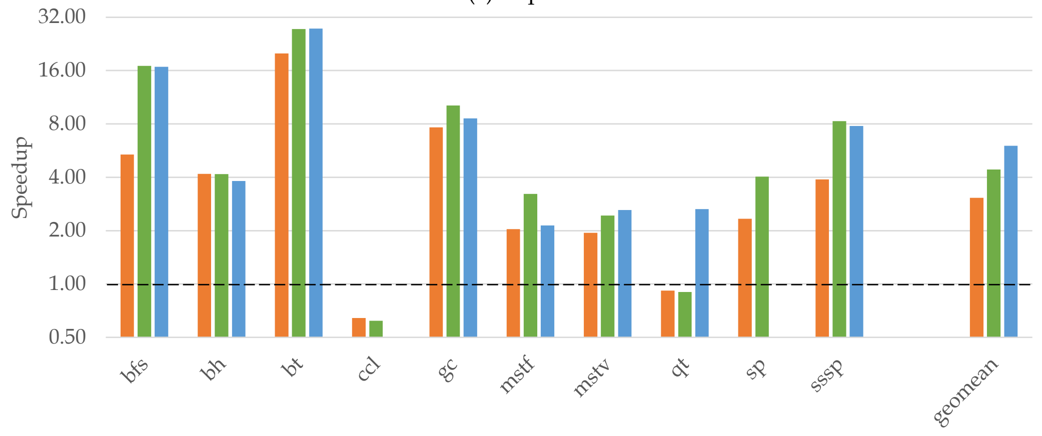


(c) Pascal

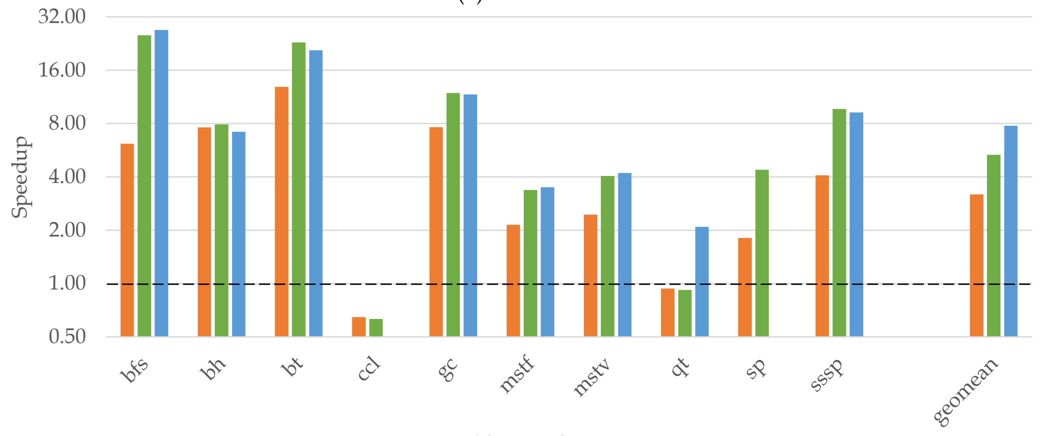
Figure 6.8: Speedup from All Optimizations Combined



(a) Kepler



(b) Maxwell



(c) Pascal

Figure 6.9: Speedup of Kernel Launch Aggregation with Optimizations over Regular Dynamic Parallelism

CHAPTER 7

VERTICAL AGGREGATION EVALUATION

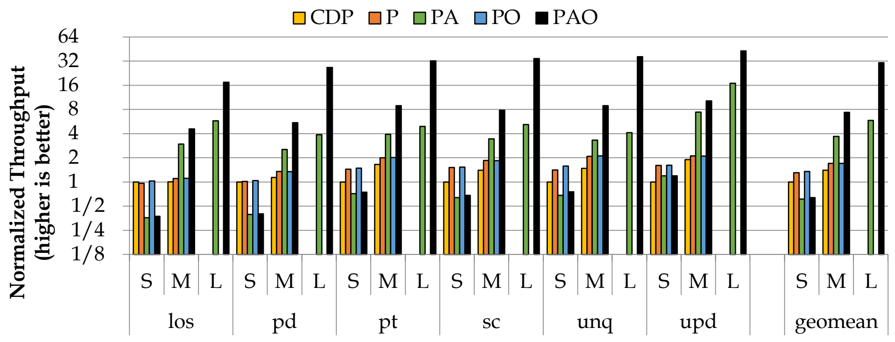
7.1 Performance

Figure 7.1 shows the throughput improvement of vertical aggregation. For each benchmark, results are shown for the small (S), medium (M), and large (L) datasets. The comparison is performed between the original CUDA dynamic parallelism version (CDP), promotion only (P), promotion with aggregation (PA), promotion with overlap (PO), and promotion with aggregation and overlap (PAO). A vertical aggregation granularity of 128 is used in PA and PAO. Throughput is considered to be the effective memory throughput (GB/s) for all benchmarks except *los* which uses ray length per second. The throughput is normalized to that of the CDP version for the small dataset. The reason throughput is used instead of execution time is to make the performance of the small, medium, and large datasets comparable because the large dataset does not have a CDP baseline since CDP fails for that dataset size.

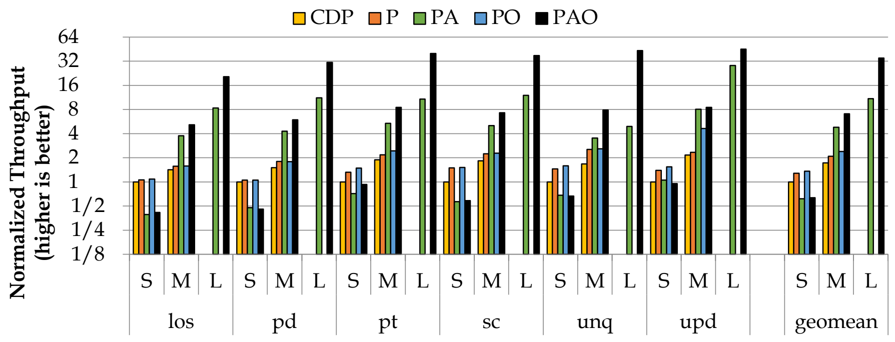
Figure 7.1(a), (b), and (c) show the throughput improvement for the Kepler, Maxwell, and Pascal architectures, respectively. The similarity of the results demonstrates the continued relevance of vertical aggregation across architecture generations.

7.1.1 Small Datasets

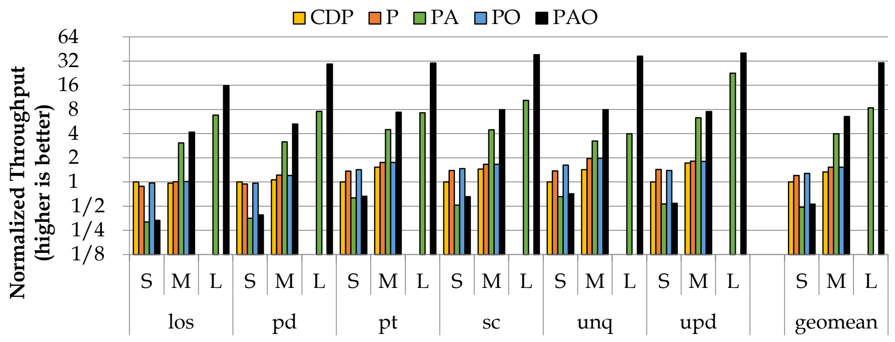
The small datasets are designed to create a chain of only two kernel launches. While not realistic, they are intended to show how the overhead of vertical aggregation can cause performance to degrade if the dataset is not big enough. The aggregation step in particular (PA and PAO) results in the most overhead since most of the thread blocks in the aggregated child grid are not



(a) Kepler



(b) Maxwell



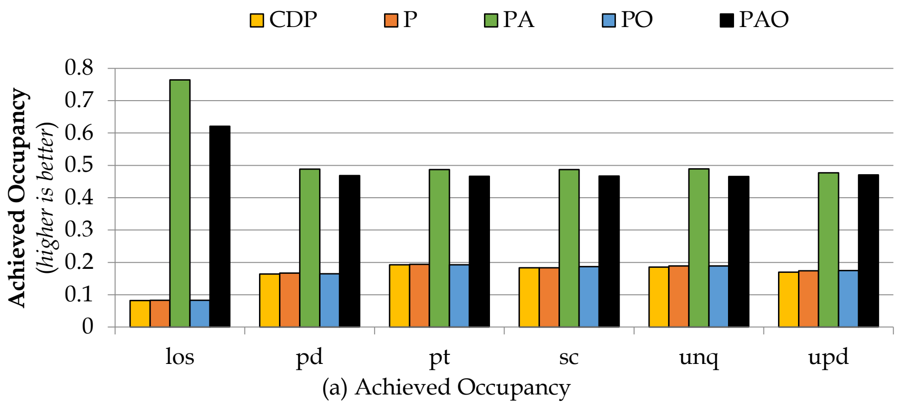
(c) Pascal

Figure 7.1: Throughput of Vertical Aggregation

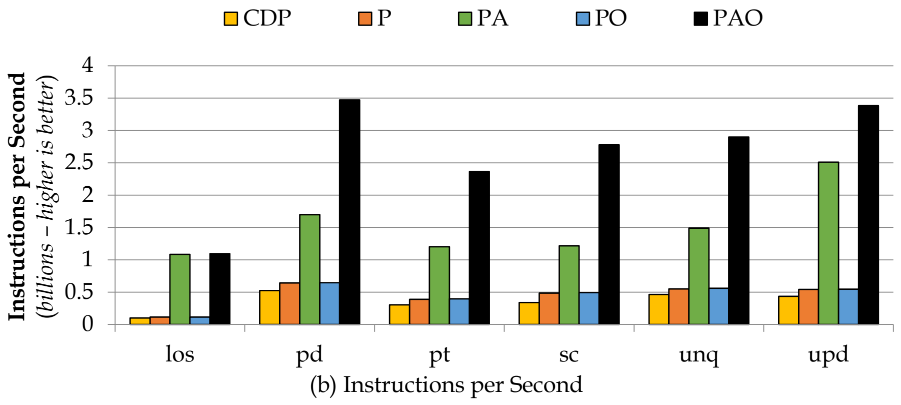
used and have to be aborted. This extreme case demonstrates the tradeoff of vertical aggregation as discussed in Section 4.5.

7.1.2 Medium Datasets

The medium datasets are the maximum size that can be executed in CDP, beyond which the call depth limit is no longer sufficient. All benchmarks show speedup at this scale. PAO on the medium dataset has a geomean speedup of 5.27× over CDP on the medium dataset. An interesting observation is that the benefit of overlap on its own (PO vs. P) is not as pronounced as the



(a) Achieved Occupancy



(b) Instructions per Second

Figure 7.2: Profiling of Vertical Aggregation

benefit of overlap combined with aggregation (PAO vs. PA). The improvement of PAO over P is much greater than the product of the improvement of PA and PO. This result shows that the aggregation and overlap steps of the vertical aggregation transformation are mutually beneficial to each other. The profiling results presented in Section 7.2 shed more light on this issue.

7.1.3 Large Datasets

For the large datasets, only PA and PAO execute to completion while the other versions fail, since they are limited by the maximum call depth. This result demonstrates the power of vertical aggregation in overcoming the call depth limitation. PAO on the large datasets achieves a throughput that is 30.44× and 21.81× higher than that achieved by CDP on the small and medium datasets respectively.

7.2 Profiling

Figure 7.2 shows profiling results for each step of vertical aggregation on the medium datasets. Figure 7.2(a) shows that PA and PAO achieve significantly better occupancy than the other versions. This result demonstrates how the aggregation optimization improves resource utilization by creating coarser grids with more work available to be scheduled. Figure 7.2(b) shows that PAO executes more instructions per second than PA despite both versions having comparable occupancy. This result demonstrates the effectiveness of the overlap optimization in making more work available sooner such that occupant thread blocks perform actual work rather than busy-wait until they are released. Interestingly, the overlap step does not improve instruction throughput when applied alone, but rather only when it is applied together with aggregation. The reason is that without aggregation, PO has low occupancy, so even if work is available sooner, the thread blocks cannot be scheduled to execute that work.

In summary, vertical aggregation improves the performance of dynamic parallelism when used to implement producer-consumer applications. It does so by reducing the total number of grid launches and increasing the granularity of grids via aggregation down the call stack. Moreover, aggregating down the call stack extends the depth of the effective stack which enables the handling of dataset sizes that were not possible otherwise.

CHAPTER 8

CASE STUDY: TRIANGLE COUNTING

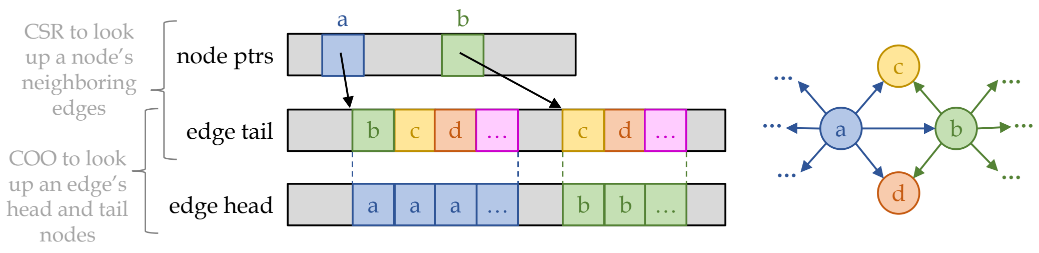
This chapter presents a detailed case study whereby horizontal aggregation is demonstrated on a triangle counting application. First, an overview of triangle counting is provided in Section 8.1. Next, the baseline GPU implementation is described in Section 8.2 followed by the dynamic parallelism implementation in Section 8.3. Finally, the performance of applying horizontal aggregation to the application is evaluated in Section 8.4. The objective of this chapter is to provide a deeper understanding of how dynamic parallelism is used and to show how aggregation interacts with other dynamic parallelism optimizations that programmers manually apply.

8.1 Application Overview

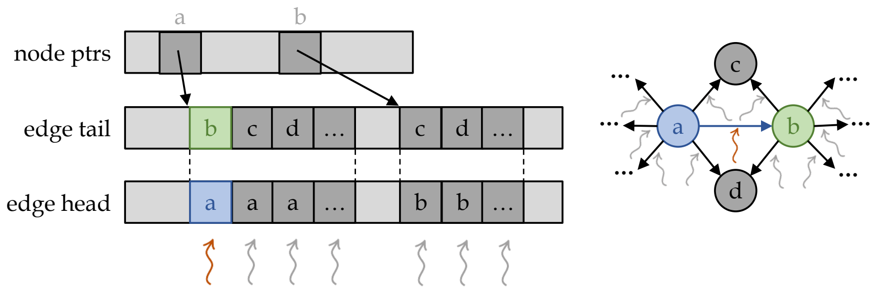
Triangle counting is a graph algorithm that computes the number of triangles in a graph. A triangle consists of three nodes that are mutually connected. Counting triangles is important for assessing a graph's cohesiveness and interconnectivity. For example, triangles are common in social networks whereby people who are connected to each other also have many connections in common, resulting in a triangle with each of those connections. Counting triangles thus helps identify the frequency of common relationships and is useful to other algorithms which identify interconnected communities within these graphs.

8.2 Baseline Implementation

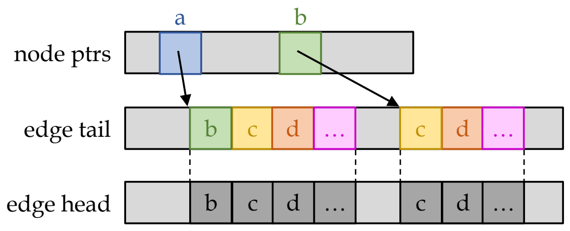
The baseline GPU implementation of triangle counting used in this case study is taken from Mailthody et al. [38]. A simple illustration of the high-level algorithm is shown in Figure 8.1. In Figure 8.1(a), an illustration of the


CSR to look up a node's neighboring edges
COO to look up an edge's head and tail nodes
node ptrs
edge tail
edge head
a
b
c
d
…

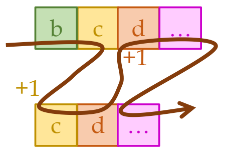
(a) Graph data structure (CSR + COO)



(b) One thread per edge looks up the edge's head and tail nodes (using COO)



(c) Thread looks up adjacency lists of head and tail nodes (using CSR)



(d) Thread loops sequentially through adjacency lists and tallies common elements

Figure 8.1: Baseline GPU Implementation of Triangle Counting

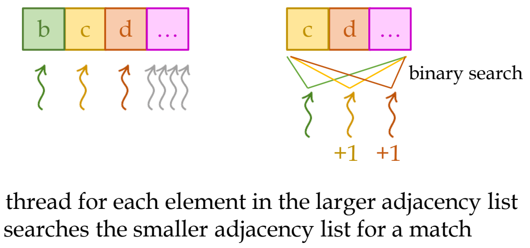

b
c
d
...
c
d
...
binary search
+1
+1
One thread for each element in the larger adjacency list searches the smaller adjacency list for a match

Figure 8.2: Dynamic Parallelism Implementation of Triangle Counting

graph data structure is shown. Both Compressed Sparse Row (CSR) and Coordinate (COO) formats are stored. CSR assists with looking up edges of a node while COO assists with looking up head and tail nodes of an edge. Note that before running the triangle counting algorithm, the graph nodes are canonicalized, and undirected graphs are converted to directed graphs where the head of the edge precedes the tail of the edge in the canonical order. Doing so ensures that triangles are not double counted.

The baseline GPU implementation launches one thread per edge in the graph as illustrated in Figure 8.1(b). The thread indexes the COO arrays to identify the head and tail nodes of the edge it is processing. Next, the thread uses the head and tail nodes to look up their adjacency lists of these nodes via the CSR data structure as shown in Figure 8.1(c). Having obtained the adjacency lists, the thread then loops sequentially through the two lists and counts the number of common elements they contain as shown in Figure 8.1(d).

At the end of this process, the algorithm has computed a triangle count for each edge without double counting edges. In another kernel, these counts are summed up to compute the total triangle count for the entire graph.

8.3 Dynamic Parallelism Implementation

8.3.1 Basic Implementation

The baseline GPU implementation described in Section 8.2 does not fully exploit the parallelism available in the algorithm. In particular, looping through the two adjacency lists sequentially as shown in Figure 8.1(d) can have long latency when the arrays are large. Furthermore, if adjacent threads process lists with significantly different lengths, the program can suffer from control divergence.

To address these issues, a dynamic parallelism version of the application is implemented to extract more parallelism out of the algorithm. Instead of looping through the arrays sequentially, the dynamic parallelism implementation performs a dynamic grid launch with a thread for each element in the larger adjacency list (parallelizing the smaller adjacency list instead was also tried, but did not perform as well). Each thread then performs a binary search on the smaller adjacency list to identify potential matches. This implementation is illustrated in Figure 8.2. Concurrent work [11] uses a similar approach, but does not use dynamic parallelism. Instead, the smaller adjacency list is distributed across threads of a warp and binary search is performed on the larger adjacency list.

Note that while this implementation of the algorithm extracts more parallelism from the application, it is less work efficient. The reduction in work efficiency is because the baseline algorithm visits each element in both adjacency lists once, whereas the dynamic parallelism implementation visits the elements in the larger adjacency list once but may visit the elements in the smaller adjacency list multiple times.

8.3.2 Optimizations

To ensure that the dynamic parallelism implementation is efficient, several optimizations are applied. Per-thread default streams are enabled to avoid serialization of grids launched from the same parent block. The number of threads per block in the child grid is tuned. Shared memory is used to store the smaller adjacency list in the SM if that list fits in the pre-allocated shared memory buffer; otherwise, the list is kept in global memory. In addition to these basic optimizations, other optimizations are applied to keep the number of dynamic grid launches under control. These optimizations are detailed in the rest of this subsection.

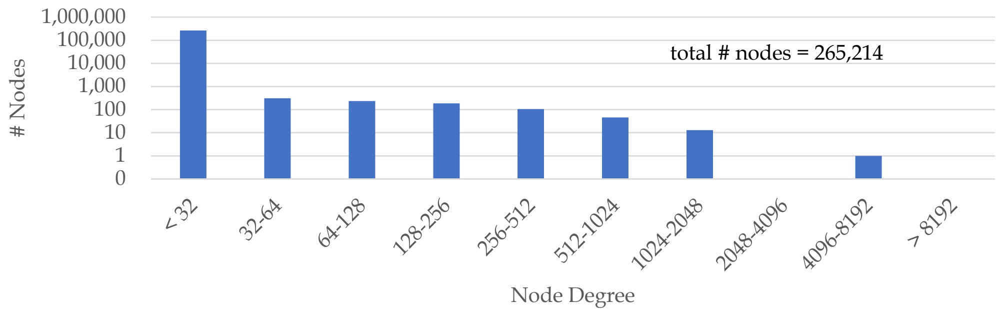
First, to avoid unnecessary dynamic grid launches, the parent thread first checks if the smaller adjacency list is empty. If so, this means that the total number of triangles the edge is involved in is zero, and it is unnecessary to launch a grid.

Second, a threshold is set for performing dynamic grid launches. A child grid is launched only if the number of elements in the larger adjacency list exceeds this threshold. This threshold is important because there can be many adjacency lists with a small number of elements. In this case, it makes more sense to process those sequentially in the parent thread and only perform the dynamic launch for the larger lists. This threshold can be tuned as is shown in Section 8.4.

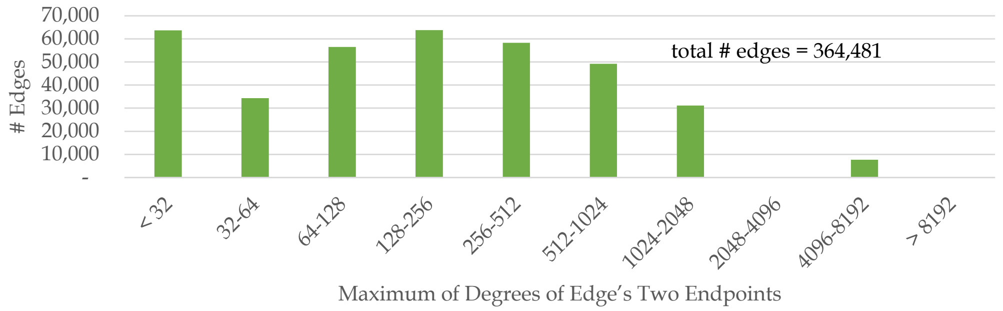
Third, a limit is set on the total number of child threads a parent thread is allowed to create. If the size of the larger adjacency list exceeds this limit, the child threads are reused to process multiple edges in that list. This optimization not only avoids excessive creation of child threads/blocks, but it also allows more data reuse across the reused child threads, particularly data reuse of the smaller adjacency list which has been loaded to shared memory. This limit can also be tuned.

8.4 Performance Evaluation

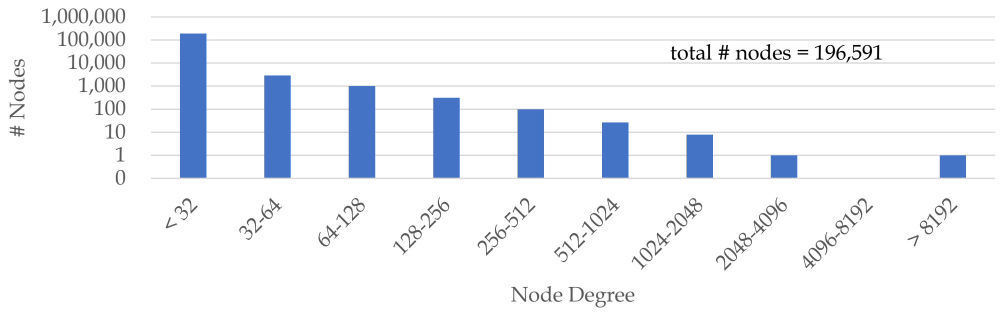
This section compares the performance of the baseline triangle counting implementation with the dynamic parallelism implementation with and without applying horizontal aggregation. Only the system with the Pascal GPU is used for evaluation. Two graphs from the Stanford Large Network Dataset Collection [39] are used: *email-EuAll*, an email network from a European Union research institution, and *loc-Gowalla*, a location-based online social network called Gowalla. The properties of these graphs are summarized in Figure 8.3. In particular, nodes are histogrammed by their degrees and edges are histogrammed by the maximum degree of their endpoints. The latter graph is particularly useful since the maximum degree of an edge's endpoints is essentially the number of child threads that will be launched by the parent thread processing that edge, provided that number is above the threshold.



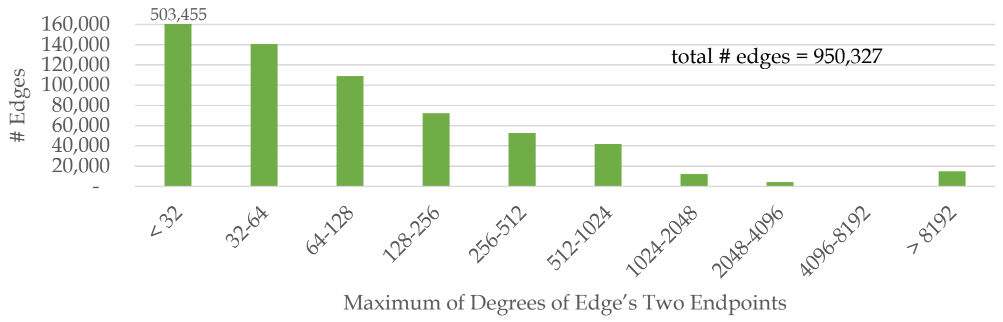
(a) Histogram of nodes by their degrees for *email-EuAll* graph



(b) Histogram of edges by the maximum of the degrees of their two endpoints for *email-EuAll* graph

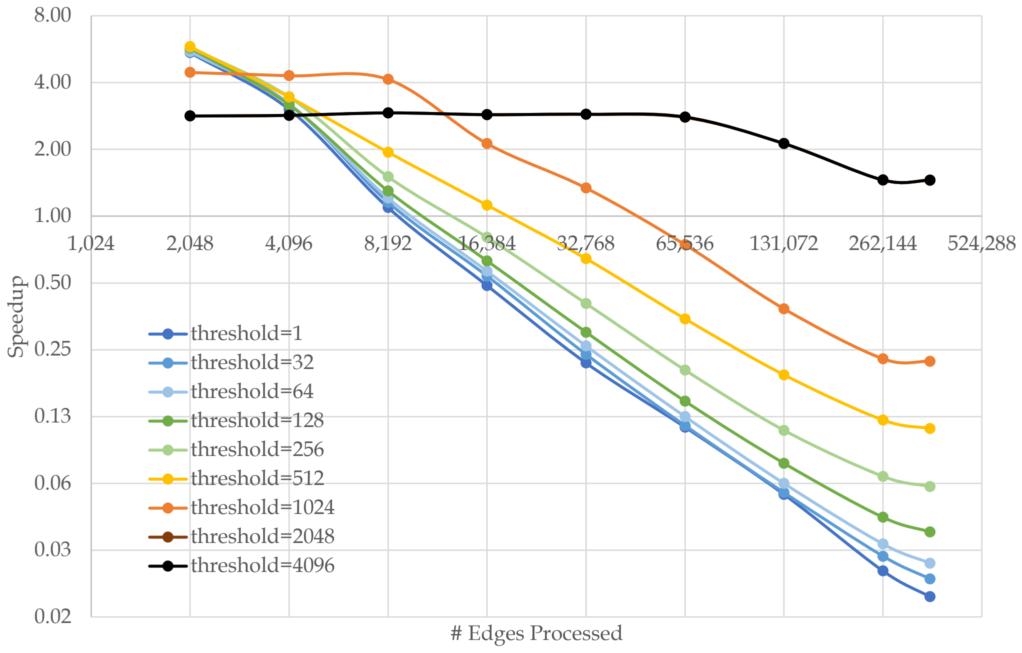


(c) Histogram of nodes by their degrees for *loc-Gowalla* graph

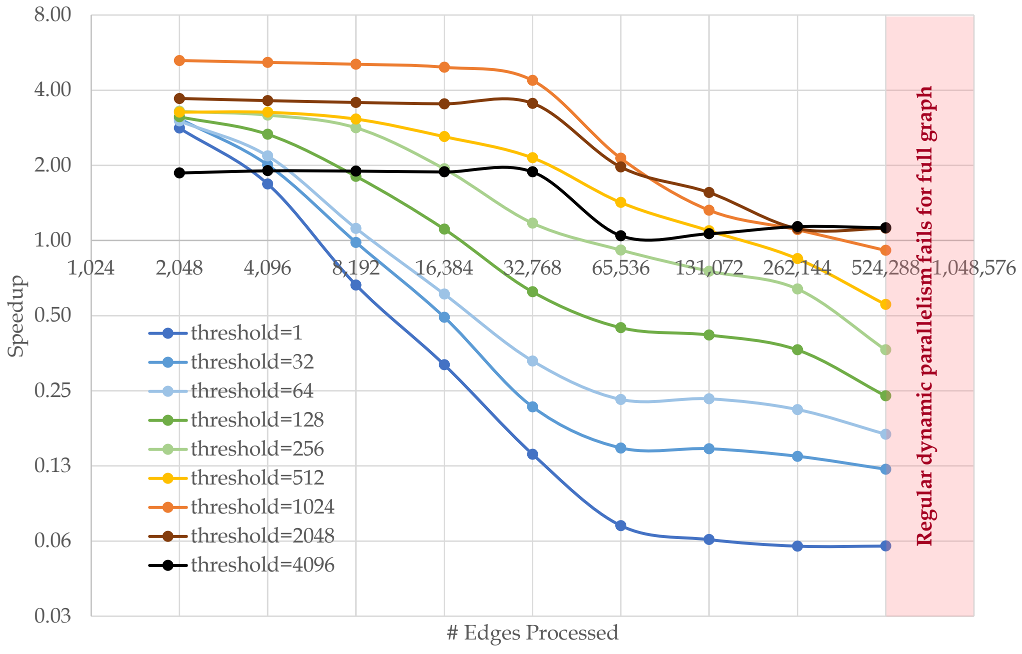


(d) Histogram of edges by the maximum of the degrees of their two endpoints for *loc-Gowalla* graph

Figure 8.3: Properties of the Graphs



(a) Results for *email-EuAll* graph



(b) Results for *loc-Gowalla* graph

Figure 8.4: Speedup of Using Regular Dynamic Parallelism Compared to Baseline Implementation

8.4.1 Regular Dynamic Parallelism vs. Baseline

Figure 8.4 shows the speedup of using regular dynamic parallelism compared to using the baseline implementation. Since the application is written for multiple GPUs, it provides the ability to select the number of edges to process on a single GPU. This number of edges per GPU is varied on the x-axis. Moreover, a different line is shown for each value of the threshold in the dynamic parallelism implementation that is used to select whether to perform the launch or process the work in the parent sequentially.

The first key observation from these results is that the benefit of using dynamic parallelism is higher when a smaller number of edges is processed per GPU. As the number of edges per GPU is increased, the benefit of dynamic parallelism decreases, resulting in a slowdown in some cases. The reason is that when the number of edges is small, the baseline which launches one thread per edge underutilizes the parallelism available on the GPU which favors the use of dynamic parallelism to launch more threads. However, as the number of edges is increased, the GPU is better utilized by the baseline version and there is little parallelism left to be exploited by the use of dynamic parallelism. Therefore dynamic parallelism results in a slowdown when there are many edges because the overhead is not justified. In the case of the larger graph, the dynamic parallelism implementation is unable to complete when processing the entire graph.

The second key observation is that as the number of edges processed per GPU increases, the optimal threshold also increases. For a small number of edges, there are a few parent threads so a lower threshold results in more child launches to fill the GPU. However, as the number of edges (hence parent threads) increases, there is less parallelism to exploit and therefore less need for launching child threads. In this case, increasing the threshold for performing the dynamic launch reduces the total number of launches, which reduces the pressure on the device and only incurs the overhead of performing launches where they are truly beneficial.

Since the optimal threshold varies depending on how many GPUs are used and is also different for different graphs, the programmer needs to tune this parameter appropriately. It is also possible to train a model on many different graphs to learn what threshold value should be used based on certain properties of the graph such as the node and edge counts and histograms.

8.4.2 Regular Dynamic Parallelism vs. Dynamic Parallelism with Horizontal Aggregation

Figure 8.5 shows the speedup of using dynamic parallelism with horizontal aggregation compared to using regular dynamic parallelism. The best horizontal aggregation granularity is shown. It is evident that the benefit of performing horizontal aggregation increases as the number of edges processed on the GPU is increased and as the threshold is decreased. That is because increasing the number of edges processed increases the total number of launches, which increases the need for performing aggregation to reduce this pressure. Similarly, decreasing the threshold also increases the total number of launches, which again increases the need for performing aggregation to reduce the pressure.

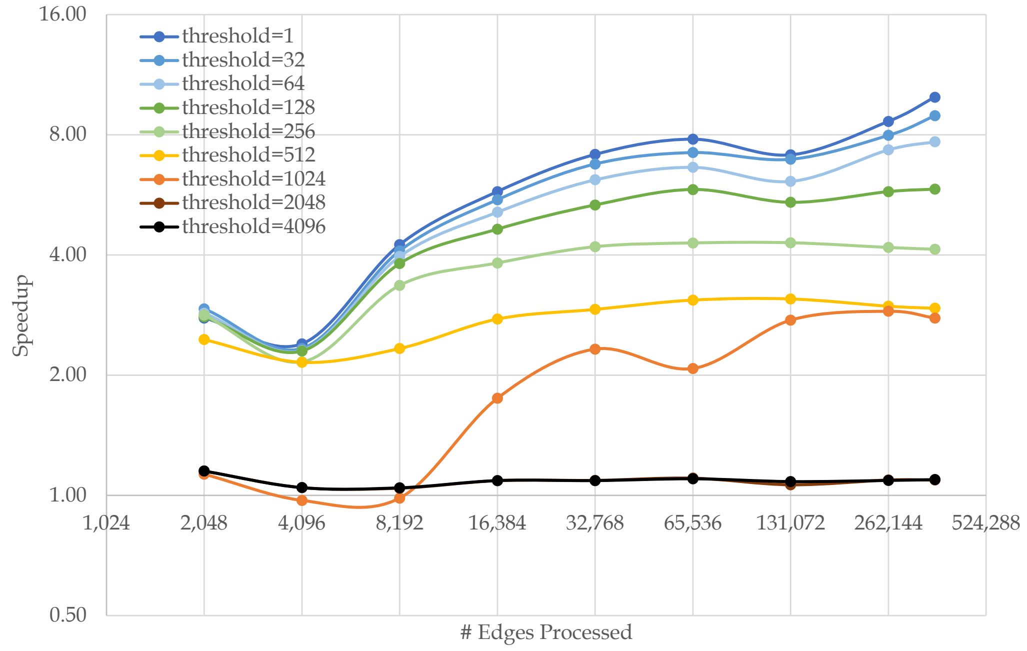
Note that while the regular dynamic parallelism implementation fails for the entire graph in the case of *loc-Gowalla*, dynamic parallelism with horizontal aggregation succeeds. This result showcases the effectiveness of horizontal aggregation in enabling dynamic parallelism at scale by reducing the pressure created by excessive grid launches.

8.4.3 Dynamic Parallelism with Horizontal Aggregation vs. Baseline

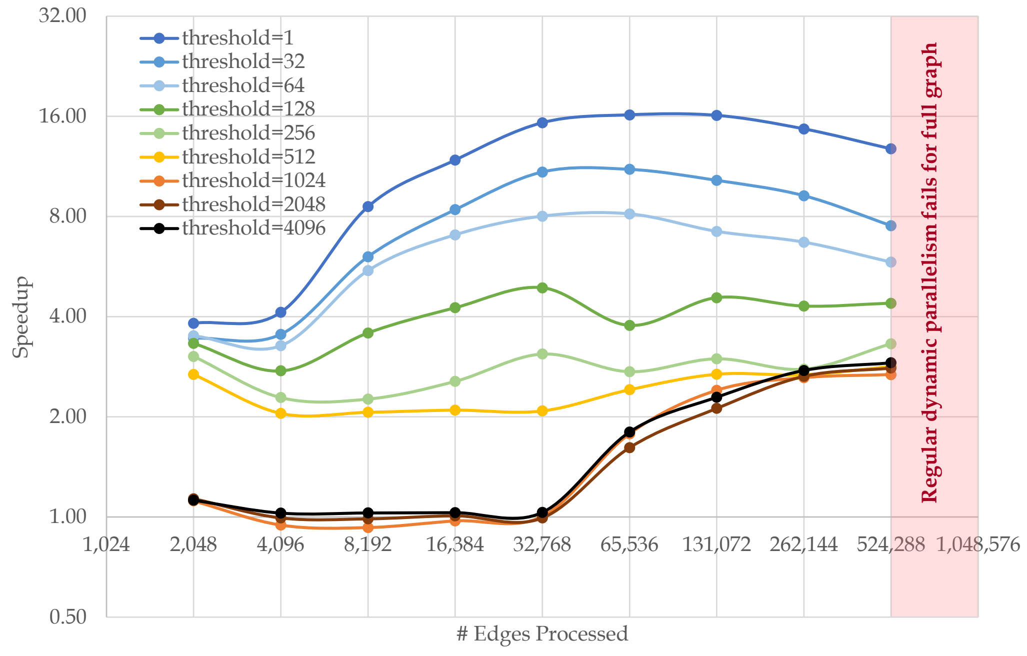
Figure 8.6 shows the speedup of using dynamic parallelism with horizontal aggregation compared to using the baseline implementation. Compared to Figure 8.4, the trend is similar but the relative speedups are much higher. Even for the entire graphs, dynamic parallelism with horizontal aggregation is able to outperform the baseline when the right threshold is picked. This result shows how important the compiler techniques proposed herein are to making dynamic parallelism more useful in practice by improving its performance.

8.4.4 Comparing Horizontal Aggregation Levels of Granularity

Figure 8.7 shows the speedup of using dynamic parallelism with horizontal aggregation at different levels of granularity compared to using the baseline implementation. Only three select threshold values are used (1, 512, 4096)

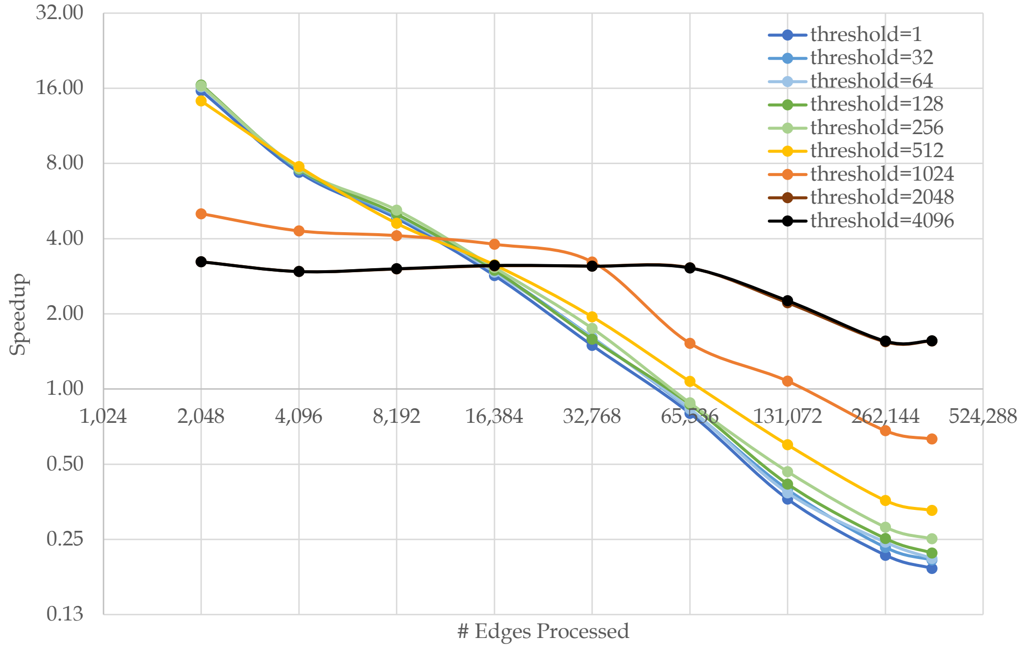


(a) Results for *email-EuAll* graph

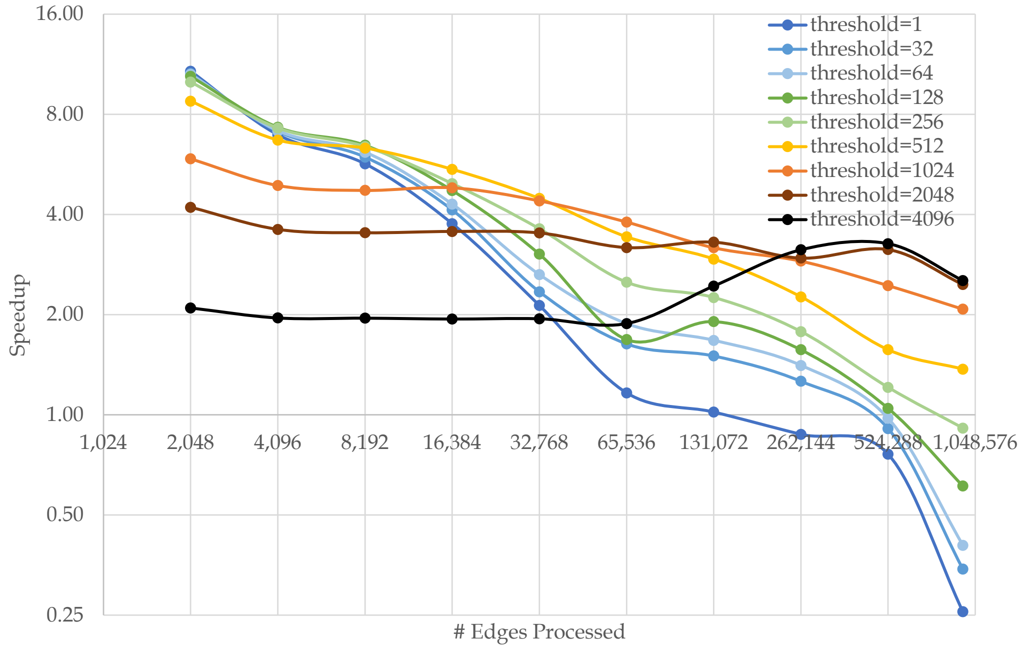


(b) Results for *loc-Gowalla* graph

Figure 8.5: Speedup of Best Horizontal Aggregation Granularity Compared to Regular Dynamic Parallelism

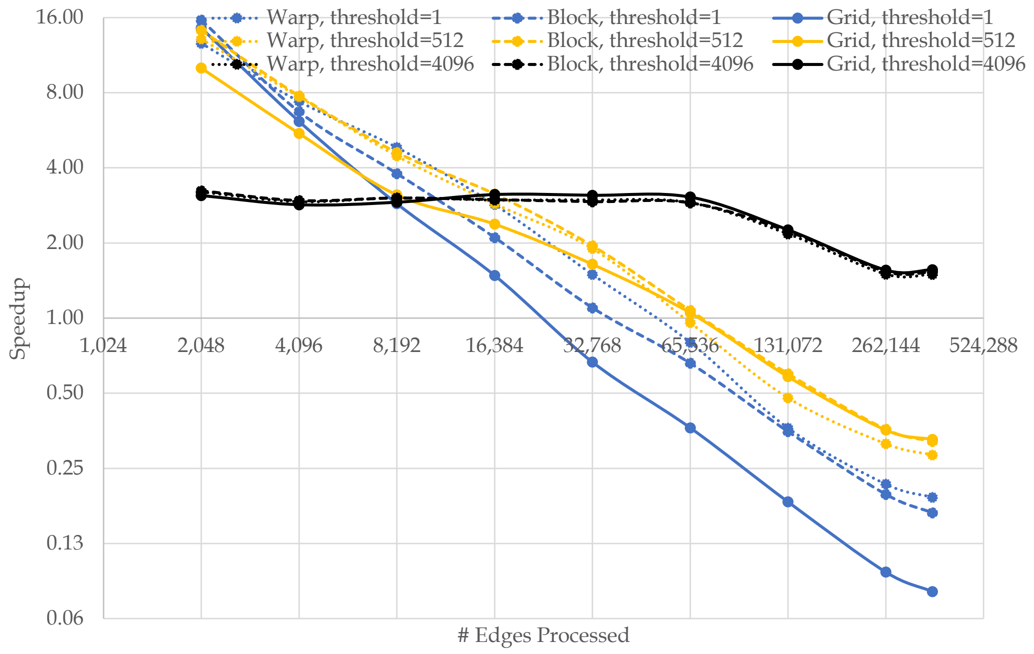


(a) Results for *email-EuAll* graph

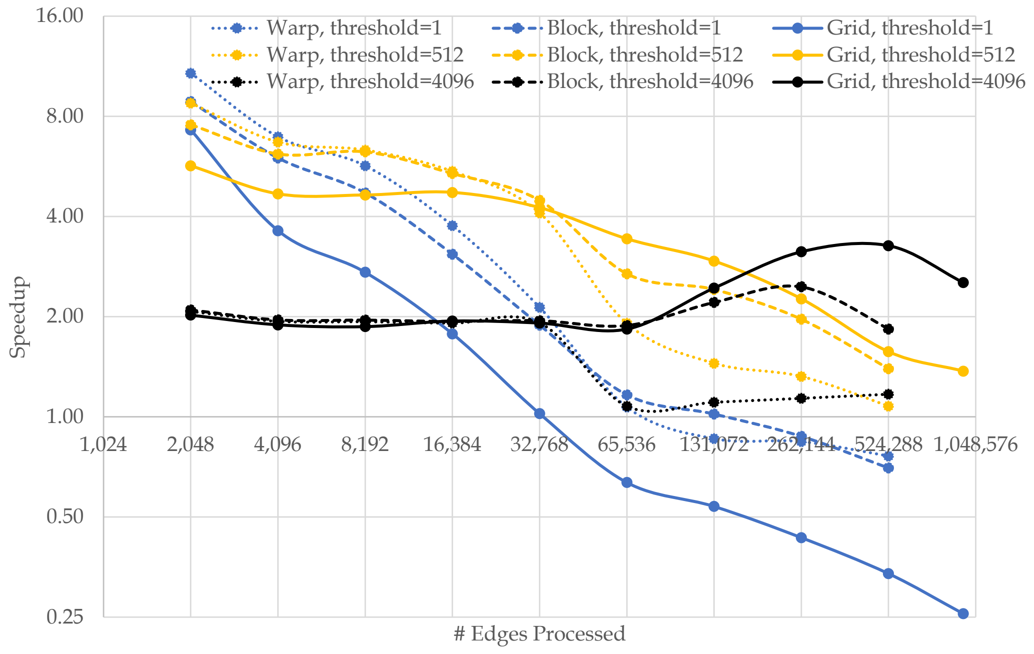


(b) Results for *loc-Gowalla* graph

Figure 8.6: Speedup of Best Horizontal Aggregation Granularity Compared to Baseline Implementation



(a) Results for *email-EuAll* graph



(b) Results for *loc-Gowalla* graph

Figure 8.7: Comparing Horizontal Aggregation Levels of Granularity Speedups over Baseline Implementation

and all three levels of granularity (warp, block, grid) are shown for each threshold value.

The results show that in the most interesting cases where a large graph is being processed and a large threshold is being used to keep the number of launches under control, the aggregation at grid granularity performs the best although other levels of granularity are comparable. This observation is consistent with the results shown in Chapter 6. On the other hand, when the threshold is kept low, aggregation at grid granularity performs the poorest. The reason is that a low threshold implies more parent threads will perform grid launches which implies that the child thread blocks will have to search a larger array at grid granularity to identify their parents. However, at warp or block granularity, the array to be searched does not grow as it is limited by the size of the warp or block, respectively.

CHAPTER 9

RELATED WORK

9.1 Applications using Dynamic Parallelism

Since the introduction of dynamic parallelism, many scientists have attempted to use it to accelerate their applications and software frameworks. It has been used differently for different applications depending on their structure. This section discusses these applications according to their usage models. The usage models can be classified into the following categories: transferring control from CPU (Section 9.1.1), regular nested parallelism (Section 9.1.2), and irregular nested parallelism (Section 9.1.3). Moreover, there have been works that evaluate dynamic parallelism on a wide range of benchmarks, not just specific applications (Section 9.1.4). Other works use dynamic parallelism as a means to implement OpenMP for GPUs (Section 9.1.5).

9.1.1 Transferring Control from CPU

Many applications that use dynamic parallelism use it simply for transferring control from the CPU to the GPU. This class of applications typically have multiple grids being launched one after the other. These grids can be different iterations of an iterative algorithm, different stages of a multi-stage workflow, or even different inputs to an application that is processing a batch of inputs. Before dynamic parallelism, these applications would have the CPU successively launch each of the grids and wait for it to finish before launching the next. With dynamic parallelism, these applications instead launch one or a few threads from the CPU which in turn launch the various grids of the application from the device.

There have been many applications in the literature from various domains that have used dynamic parallelism for transferring control from CPU. These applications include: connected component labelling [40], numerical integration [41], strongly connected component search for LTL model checking [42], stencil [43], lattice quantum chromodynamics [44], sparse iterative solvers [45], X-ray rendering [46], dense linear algebra [47], holography [48], community detection in social networks [49, 50], self-organizing maps [51], and video processing [52].

One advantage of using dynamic parallelism for transferring control from CPU is that it frees up the CPU to do other kinds of work or to sleep and save energy. Another advantage is that it avoids copying data from the GPU to the CPU in the cases where the launch of the next grid is data-dependent (e.g., checking for convergence in an iterative solver).

Note that since the number of parent threads in this usage model is typically one or a few, there is no excessive launching of child grids. Therefore, this usage model is not of interest to the aggregation optimizations described herein. The optimizations herein focus on applications with nested parallelism, which are described in the rest of this section.

9.1.2 Regular Nested Parallelism

We make the distinction between two types of nested parallelism: regular and irregular. In regular nested parallelism, the number of nesting levels and the amount of parallel work available at all levels of the nesting is known at the beginning of execution. Without dynamic parallelism, applications with regular nested parallelism can launch a grid for each level of nesting. Since the number of levels and the amount of work at each level is known, doing so is slightly tedious but not too difficult. With dynamic parallelism, applications with regular nested parallelism can simply have each worker at one level of nesting launch a grid of workers for the next level of nesting.

There are many applications in the literature where dynamic parallelism has been used for capturing regular nested parallelism. One application is aperiodic reflectarray antenna analysis [53, 54] where an interpolation step involves launching a parent thread for each input index that performs a computation then launches a grid of child threads to perform an accumulation to multiple output indices. Another application is inverse distance weighting [55, 56], which also involves interpolation whereby a parent thread is launched for each point, the thread performs a prediction for that point, then the thread launches a grid of child threads to compute the distance to other points. Yet another example is video streaming [57], in which a parent thread is launched for each macro-block that launches child grids to process sub-blocks of that macro-block.

One advantage of using dynamic parallelism for applications with regular nested parallelism is that it provides a more intuitive way of programming. Another advantage is that it avoids having to wait for all workers in the previous level to complete before moving on to the next level. Finally, it transfers control from the CPU to the GPU which captures the same advantages described in Section 9.1.1 of freeing up the CPU and avoiding communication between the CPU and GPU.

9.1.3 Irregular Nested Parallelism

Recall that nested parallelism is a pattern wherein an application has multiple levels of parallelism. Also recall that in regular nested parallelism, the number of nesting levels and amount of work at each level is known prior to execution. In contrast, in irregular nested parallelism, the number of nesting levels and/or amount of work at each level is not known prior to execution. This pattern is the most tedious to program without dynamic parallelism because the number of child threads (if any) at the next level of nesting is not known until the parent threads at the previous level of nesting have actually executed. Without dynamic parallelism, applications with irregular nested parallelism may either serialize the work of the next level of nesting in the thread processing at the previous level of nesting, or they may use queues to collect the work from all the threads in the previous level of nesting and launch a new grid for the next level of nesting. With dynamic parallelism, threads at the previous level of nesting can simply launch child grids for the next level of nesting and configure them with the desired number of threads.

There are two sources of irregularity in irregular nested parallelism. The first source is where the number of child threads varies across parent threads. This kind of irregular nested parallelism is common in graph applications where a thread processing a node launches a grid to visit all of that node's neighbors. Examples from the literature where dynamic parallelism has been used to accelerate such applications include breadth first search [2, 58], single-source shortest path [2, 58], PageRank [58], and depth first search [59]. Some applications [58] perform the optimization of setting a threshold on the number of child threads which was applied in the triangle counting case study in Chapter 8. This category of applications is the most suitable for this optimization because of the variable child thread count.

The second source of irregularity in irregular nested parallelism is where the launch condition may not be known. This kind of irregular nested parallelism is common in tree applications such as mesh refinement trees, search trees, and data mining trees where one does not know whether to recurse deeper at a node in the tree until that node is evaluated. Examples where dynamic parallelism has been used to accelerate mesh refinement trees include quadtrees [29] where one does not know whether to process a bounding box at a finer resolution until visiting that bounding box and evaluating its properties. Examples where dynamic parallelism has been used to accelerate search trees include the N-Queens problem [60, 61, 62, 63], NegMax trees [64], and the travelling salesman problem [62, 63], where one must evaluate whether they have actually arrived at a solution before recursing deeper into the tree and continuing the search. Examples where dynamic parallelism has been used to accelerate data mining trees include cluster feature trees [65], generic hierarchical clustering [66], and frequent pattern trees [67]. Irregular nested parallelism due to unknown launch conditions can also be found in other applications besides trees. For example, dynamic parallelism has been used to accelerate genomic read mapping [68, 69] where parent threads perform seed generation then launch child grids to perform seed extension only when necessary. It has also been used to accelerate spiking neural networks [70] where parent threads update neuron states and evaluate the spiking threshold, then only launch child grids for spiked cases to update all the synapses.

One advantage of using dynamic parallelism for applications with irregular nested parallelism is that it avoids the need to serialize the work at the next nesting level, which wastes parallelism. Another advantage is that it avoids the need to use complicated queue data structures to collect the work to be processed in parallel. It also has the same advantages of regular nested parallelism described in Section 9.1.2, namely being easier to program and avoiding waiting for the entire nesting level to finish before proceeding, and the same advantages as transferring control from the CPU described in

Section 9.1.1, namely freeing up the CPU and avoiding communication with it.

9.1.4 Performance Evaluation

Beyond specific applications, there has been work on evaluating dynamic parallelism on a wide range of benchmarks. Some works [30, 71] mainly evaluate benchmarks that use dynamic parallelism for transferring control to CPU. Others [7] evaluate benchmarks with actual nested parallelism and have made similar observations about the inefficiency of dynamic parallelism when launches are excessive.

9.1.5 Dynamic Parallelism for Implementing OpenMP

Dynamic parallelism has also been used by implementations of OpenMP. Liao et al. [72] and Bertolli et al. [73] use dynamic parallelism for transferring control from the CPU when implementing OpenMP parallel regions for GPUs. Ozen et al. [74] use dynamic parallelism to capture nested parallelism in OpenMP.

9.2 Dynamic Parallelism Optimizations

9.2.1 Compiler and Software Optimizations

Several compiler/software optimizations have been proposed for optimizing dynamic parallelism. CUDA-NP [8] is a compiler approach whereby rather than using the dynamic parallelism API, parallel loops are nested inside parent threads and are annotated with directives. The compiler transforms the parent kernel to launch an excess number of threads such that each parent thread is accompanied with multiple slave threads in the same thread block. Using control flow, these slave threads are activated whenever a parallel loop is encountered and are used to execute the iterations of that parallel loop concurrently. Collaborative Task Engagement [75] further improves this technique by having multiple parent threads share slave threads for better load balance and less control divergence. This approach specifically targets the case where the amount of nested parallelism in the parent thread is low such that a single thread block is sufficient to handle it.

Free Launch [16] eliminates launches of child grids entirely and instead reuses parent threads/blocks to process the child grids. This transformation requires the code for all the kernels involved in the call hierarchy to be inlined into the same kernel, a technique often referred to as a megakernel [76]. It also requires the blocks of the megakernel to be resident on the GPU throughout the entire execution, even if those blocks are idle, in order to pick up tasks that may be created in the future, a technique often referred to as persistent threads [77]. One drawback of this approach is that constructing the megakernel requires the code for all participating kernels to be available to the compiler in the same translation unit which prevents separate compilation of kernels. Another drawback is that the different kernels participating in the megakernel may have different resource requirements for resources such as registers, shared memory, threads per block, etc. The megakernel therefore requires the maximum of each resource across all participating kernels, so if there is a large disparity across participating kernels, this may lead to significant underutilization of resources by the megakernel. Yet another drawback is that the use of persistent threads hogs GPU resources even when they are not in use which leads to underutilization of resources when there are co-running grids on the GPU. In contrast, the compiler techniques presented herein do not eliminate child kernel calls, but aggregate them to mitigate their overhead. Megakernels or persistent threads are not used, and therefore separate compilation is possible and co-runner kernels are not starved.

Li et al. [78, 79], Wu et al. [80], and KLAP [81] propose aggregation techniques similar to the techniques presented herein. Li et al. [78, 79] use parallelization templates targeting optimize irregular nested loops and parallel recursive computations. Wu et al. [80] use generic compiler transformations with the assistance of directive-based annotations. KLAP [81] uses generic compiler optimizations without additional annotations. This work provides more details about how these techniques can be implemented and optimized as well as an evaluation across multiple architecture generations and a case study on a real application.

Zhang et al. [82, 83] further enhance the technique of aggregating child grids by grouping together child grids with similar optimal configurations rather than aggregating all child grids into the same grid. This optimization is based on the observation that in applications with irregular nested parallelism, a parent thread may launch anywhere between one and thousands of child threads. Provisioning a large child block size for parents with few child threads results in wasted threads, whereas provisioning a small child block size for parents with many child threads underutilizes the GPU. Instead, this optimization groups child grids favoring small block sizes together and child grids favoring large block sizes together in separate aggregated child grids with different configurations resulting in better utilization of resources. However, this technique is hard to implement in a compiler for CUDA without hints from the programmer because it is difficult for the compiler to prove that it is legal to change the thread block size and that the programmer has not made any assumptions about it elsewhere in the code.

9.2.2 Hardware Optimizations

Many hardware techniques have also been proposed for mitigating the overhead of dynamic parallelism. Dynamic Thread Block Launch (DTBL) [10, 84] proposes hardware support for lightweight dynamic launching of thread blocks rather than heavyweight dynamic launching of entire grids. The dynamically launched thread blocks are essentially coalesced to existing kernels on the fly by the hardware. DTBL is further enhanced with LaPerm [14], a locality-aware scheduler for dynamically launched thread blocks which prioritizes the execution of child thread blocks and attempts to execute them on the same SMs as their parents for improved locality while remaining aware of load balance across SMs.

SPAWN [15] propose a hardware controller that assists the programmer in identifying whether or not a dynamic launch would be favorable based on historic behavior of child grids as well as the current state of the device and how busy it is. If the controller finds that performing the launch is favorable, it schedules the child grid, otherwise it informs the programmer who may then execute the child work serially in the parent.

While these techniques are promising for improving dynamic parallelism performance in future GPU generations, the compiler techniques proposed herein improve performance on current GPUs.

9.3 Parallelization and Optimization of Recursion and Producer-Consumer Chains

Kernel Weaver [85] fuses producer-consumer kernels performing relational algebra operations, but this work was before dynamic parallelism was introduced. Vertical aggregation specifically targets producer-consumer patterns expressed with dynamic parallelism. The benchmarks used include relational operations. Gómez-Luna et al. [4] introduce a set of GPU algorithms (called data sliding algorithms) that perform promotion with aggregation and overlap through libraries. However, this work does not connect these techniques with recursion using dynamic parallelism for better programmability.

There has been a lot of work on optimizing recursive function calls or sequential loops with dependent operations by extracting parallelism from them to execute on multi-core CPUs or CPU vector units [86, 87, 88, 89, 90] or by improving their locality [91, 92, 93]. The promotion and overlap steps of vertical aggregation have a similar objective of extracting more parallelism from long dependence chains on GPUs by removing as much work as possible from the critical path.

9.4 Other Trends in GPU Computing

9.4.1 Parallelization of Task Graphs

Not all computation patterns can be described in terms of either flat or nested parallelism. Some computation patterns have a more complex parallelism structure. In some applications, a task may be dependent on multiple predecessor tasks and cannot execute until all its predecessors have completed. For example, in wavefront applications that diagonally sweep through a grid, a tile of that grid cannot be processed until its top and left neighboring tiles have been processed. This kind of parallelism structure cannot be expressed with nested parallelism since the tile cannot be nested in either of its predecessors because it must wait for both. Therefore, dynamic parallelism is not an intuitive way of programming such applications.

An intuitive way of programming such applications on GPUs is to allow the programmer to describe a task graph and have a runtime system schedule that task graph on the GPU and enforce dependences. CUDA [94] and HSA [95] both provide support for describing and executing such task graphs, but this support is at the granularity of entire grids. There have been multiple efforts to provide support for task graph scheduling and finer granularity such as at the granularity of thread blocks. Wireframe [96] and Liu [97] propose hardware support to implement such a system while Juggler [98] provides a software-only solution. Versapipe [99] is a software framework for implementing and optimizing task pipelines, which are a special case of task graphs.

9.4.2 Kernel Synthesis

As GPUs continue to evolve, optimizing code for GPUs is like chasing a moving target. Different architecture generations tend to favor different optimizations and tuning parameters. To address this issue, tools like Tangram [100, 101, 102, 103, 104] enable programmers to write simple codelets representing different algorithms for the same computation and compose these codelets together based on the architectural hierarchy of the device to automatically synthesize optimized kernels. These kernels are further autotuned according to the parameters of the specific architecture generation.

Tangram's approach for kernel synthesis is appropriate for supporting dynamic parallelism. Without dynamic parallelism, the GPU hierarchy in Tangram is such that grids distribute work to blocks which distribute to warps which distribute to threads. Since dynamic parallelism allows threads to launch other grids, it creates recursion in the architecture hierarchy such that threads which are at the lowest level can distribute work to grids which are at the highest level. Moreover, Tangram's facilities for selecting between serial and parallel codelets is appropriate for automatically synthesizing code that either serializes nested parallel work, parallelizes it via a dynamic launch, or does either depending on a tunable threshold. This kind of optimization was applied manually in the triangle counting case study in Chapter 8.

9.4.3 Memory Systems

Dynamic parallelism increases the autonomy of GPUs in systems because it allows the GPU to launch new grids without CPU intervention. As GPUs become more mainstream, other features have also been added to increase their autonomy. One notable feature is shared virtual memory or unified memory whereby a GPU can directly access memory allocated by the CPU without having to rely on the CPU for performing explicit memory copies. Various works have showcased the collaborative execution patterns that shared virtual memory enables. These works include benchmark suites such as Hetero-mark [105, 106, 107], Chai [108, 109], and HeteroSync [110, 111], optimization of applications such as Bézier Surfaces [112] and Betweenness Centrality [113], simulation frameworks such as gem5-gpu [114, 115] and MGSim [116], and other evaluation efforts [117, 118, 119, 120, 121, 122]. Various software infrastructures have also been built to assist with efficient scheduling of collaborative workloads. These infrastructures include EMRF [123] for balancing fairness and efficiency, Luminar [124] for profile guided resource scheduling, FinePar [125] and Cho et al. [126] for automated workload partitioning, and Airavat [127, 128] for efficient power management. Veselý et al. [129, 130] use shared virtual memory to implement system calls from GPUs. Architecture enhancements in the context of shared virtual memory include topics such as coherence [131, 132], address translation [133], and cache design [134, 135, 136, 137, 138]. Vijayaraghavany et al. [139] use shared virtual memory in the design of accelerator building blocks for exascale supercomputers. While integrating the CPU and GPU memory hierarchies benefits programmability and performance, it poses security concerns since the GPU may now access CPU memory. Olson et al. [140] propose a sandboxing mechanism for accelerators while Erb et al. [141] address the problem of buffer overflow from GPU code.

Beyond volatile memory, non-volatile memory technologies are gaining attention for their promise to provide high density and low-power byte-addressable memories. They have stimulated in research on various topics such as programming models for guaranteeing crash consistency of persistent data [142, 143, 144], design of durable data structures [145, 146, 147], representation of pointers in persistent objects [148, 149, 150], garbage collection for non-volatile memory [151], and the use of non-volatile memory devices for compute [152, 153, 154]. Erudite [155] aims at redesigning the memory hierarchy to bring GPUs and NVM closer together.

CHAPTER 10

CONCLUSION

Dynamic parallelism provides an intuitive interface for programming applications with nested parallelism on GPUs, but it has seen limited adoption due its high overhead. This work shows that to make dynamic parallelism practical for accelerating applications with nested parallelism, compiler transformations can be used to aggregate dynamically launched grids, thereby amortizing their launch overhead and improving their occupancy, without the need for additional hardware support.

A set of compiler techniques is presented to mitigate the overhead of dynamic parallelism. The first technique, horizontal aggregation, fuses grids launched by multiple parent threads into a single aggregated grid thereby reducing the number of launches and increasing the amount of work per grid to improve occupancy. This technique can be applied at warp, block, or grid granularity with larger levels of granularity trading off higher aggregation overhead and delaying blocks for fewer launches and larger grids. The second technique, vertical aggregation, optimizes recursive kernels with producer-consumer chains by aggregating grids down the call stack and launching them earlier than their original call site, enforcing dependences via release-acquire chains. This optimization not only reduces the number of grids launched, but also extends the depth of the call stack and shortens the critical path of the application. Horizontal and vertical aggregation at varying levels of granularity result in significant speedups for a variety of benchmarks representing common nested parallelism patterns. The speedups are obtained across multiple generations of GPU architectures showing the continued relevance of these techniques.

The lessons learned from the compiler techniques, the case study, and the literature review presented herein suggest several best practices for writing dynamic parallelism code. The fundamental objectives behind these best practices are to control the number of grid launches and to amortize the launch overhead across more work in the child grid. The first practice is to set a threshold on the number of child threads and to serialize the nested work in the parent thread if the threshold is not met. This practice is demonstrated in the triangle counting case study and is also applied by some applications and targeted by some hardware optimizations in the literature. It is possible to automatically apply this technique in the compiler which is an interesting future direction. The second practice is to aggregate grids across parent threads or down the call stack. This work shows that this optimization can be performed effectively by the compiler. However, cognizance of the compiler's ability to aggregate grids informs other best practices that follow. The third practice is to use uniform arguments and child block sizes across parent threads whenever possible. This practice enables the compiler to generate more efficient aggregation code. For applications where parent threads have significant variation in the number of child threads, hence variation in optimal child block sizes, child grids can be grouped according to their optimal child block sizes and groups can be aggregated separately where each group has a uniform child block size. This technique has been shown to be effective when applied manually in the literature. Automating it in the compiler would require hints from the programmer on how to safely modify the block size which is an interesting future direction. Finally, the fourth practice is to reuse child threads, which increases the amount of work per thread block, amortizing the overhead introduced from aggregation. This practice has been demonstrated manually in the triangle counting case study. It is also possible to automatically apply this technique in the compiler which is an interesting future direction.

REFERENCES

[1] D. B. Kirk and W.-m. Hwu, *Programming Massively Parallel Processors: A Hands-On Approach.* Morgan Kaufmann, 2016.

[2] P. Zhang, E. Holk, J. Matty, S. Misurda, M. Zalewski, J. Chu, S. McMillan, and A. Lumsdaine, "Dynamic parallelism for simple and efficient GPU graph algorithms," in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA3 '15. ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2833179.2833189 pp. 11:1–11:4.

[3] M. L. Sætra, A. R. Brodtkorb, and K.-A. Lie, "Efficient GPU-implementation of adaptive mesh refinement for the shallow-water equations," *Journal of Scientific Computing*, vol. 63, no. 1, pp. 23–48, 2015.

[4] J. Gómez-Luna, L.-W. Chang, I.-J. Sung, W.-M. Hwu, and N. Guil, "In-place data sliding algorithms for many-core architectures," in *Parallel Processing (ICPP), 2015 44th International Conference on.* IEEE, 2015, pp. 210–219.

[5] S. Jones, "Introduction to dynamic parallelism," in *GPU Technology Conference Presentation*, 2012.

[6] W.-m. Hwu, *Heterogeneous System Architecture: A New Compute Platform Infrastructure.* Morgan Kaufmann, 2015.

[7] J. Wang and S. Yalamanchili, "Characterization and analysis of dynamic parallelism in unstructured GPU applications," in *Workload Characterization (IISWC), 2014 IEEE International Symposium on.* IEEE, 2014, pp. 51–60.

[8] Y. Yang and H. Zhou, "CUDA-NP: Realizing nested thread-level parallelism in GPGPU applications," in *ACM SIGPLAN Notices*, vol. 49, no. 8. ACM, 2014, pp. 93–106.

[9] "A CUDA dynamic parallelism case study: PANDA," https://devblogs.nvidia.com/parallelforall/a-cuda-dynamic-parallelism-case-study-panda, accessed: 2016-04-01.

[10] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili, "Dynamic thread block launch: A lightweight execution mechanism to support irregular applications on GPUs," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture.* ACM, 2015, pp. 528–540.

[11] Y. Hu, H. Liu, and H. H. Huang, "TriCore: Parallel triangle counting on GPUs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. Piscataway, NJ, USA: IEEE Press, 2018. [Online]. Available: http://dl.acm.org/citation.cfm?id=3291656.3291675 pp. 14:1–14:12.

[12] K. Hou, W. Liu, H. Wang, and W.-c. Feng, "Fast segmented sort on GPUs," in *Proceedings of the International Conference on Supercomputing.* ACM, 2017, p. 12.

[13] K. Hou, "Exploring performance portability for accelerators via high-level parallel patterns," Ph.D. dissertation, Virginia Tech, 2018.

[14] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili, "Laperm: Locality aware scheduler for dynamic parallelism on GPUs," in *The 43rd International Symposium on Computer Architecture (ISCA)*, June 2016.

[15] X. Tang, A. Pattnaik, H. Jiang, O. Kayiran, A. Jog, S. Pai, M. Ibrahim, M. T. Kandemir, and C. R. Das, "Controlled kernel launch for dynamic parallelism in GPUs," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on.* IEEE, 2017, pp. 649–660.

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

[17] S. Xiao and W.-c. Feng, "Inter-block GPU communication via fast barrier synchronization," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on.* IEEE, 2010, pp. 1–12.

[18] T. Sorensen, A. F. Donaldson, M. Batty, G. Gopalakrishnan, and Z. Rakamarić, "Portable inter-workgroup barrier synchronisation for GPUs," in *ACM SIGPLAN Notices*, vol. 51, no. 10. ACM, 2016, pp. 39–58.

[19] M. Harris and K. Perelygin, "Cooperative groups: Flexible CUDA thread programming," https://devblogs.nvidia.com/cooperative-groups, 2017.

[20] T. Kaldewey, J. Hagen, A. Di Blas, and E. Sedlar, "Parallel search on video cards," in *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, ser. HotPar'09. USENIX Association, 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855591.1855600 p. 9.

[21] J. A. Stratton, S. S. Stone, and W.-m. Hwu, "MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs," in *International Workshop on Languages and Compilers for Parallel Computing.* Springer, 2008, pp. 16–30.

[22] H.-S. Kim, I. El Hajj, J. Stratton, S. Lumetta, and W.-M. Hwu, "Locality-centric thread scheduling for bulk-synchronous programming models on CPU architectures," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization.* IEEE Computer Society, 2015, pp. 257–268.

[23] H.-S. Kim, I. El Hajj, J. A. Stratton, and W.-M. Hwu, "Multi-tier dynamic vectorization for translating GPU optimizations into CPU performance," Center for Reliable and High-Performance Computing, Tech. Rep., 2014.

[24] I. El Hajj, "Dynamic loop vectorization for executing opencl kernels on CPUs," M.S. thesis, University of Illinois, 2014.

[25] L. Howes and A. Munshi, *The OpenCL Specification, Version 2.0.* Khronos Group, 2015.

[26] S. Yan, G. Long, and Y. Zhang, "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*, ser. PPoPP '13. ACM, 2013, pp. 229–238.

[27] 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*, ser. GPGPU-3. ACM, 2010, pp. 63–74.

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

[29] NVIDIA, "CUDA samples v. 7.5," 2015.

[30] Y. Ukidave, F. N. Paravecino, L. Yu, C. Kalra, A. Momeni, Z. Chen, N. Materise, B. Daley, P. Mistry, and D. Kaeli, "NUPAR: A benchmark suite for modern GPU architectures," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '15. ACM, 2015, pp. 253–264.

[31] A. V. P. Grosset, P. Zhu, S. Liu, S. Venkatasubramanian, and M. Hall, "Evaluating graph coloring on GPUs," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '11. ACM, 2011, pp. 297–298.

[32] "Matrix market," http://math.nist.gov/MatrixMarket, accessed: 2016-04-01.

[33] "GPU pro tip: CUDA 7 streams simplify concurrency," http://devblogs.nvidia.com/parallelforall/gpu-pro-tip-cuda-7-streams-simplify-concurrency, accessed: 2016-04-01.

[34] "CUDA dynamic parallelism API and principles," https://devblogs.nvidia.com/parallelforall/cuda-dynamic-parallelism-api-principles, accessed: 2016-04-10.

[35] N. Bell and J. Hoberock, "Thrust: a productivity-oriented library for CUDA," *GPU Computing Gems: Jade Edition*, 2012.

[36] M. Billeter, O. Olsson, and U. Assarsson, "Efficient stream compaction on wide SIMD many-core architectures," in *Proceedings of the Conference on High Performance Graphics 2009*, ser. HPG '09. ACM, 2009, pp. 159–166.

[37] NVIDIA, "Profiler user's guide v. 7.5," 2015.

[38] V. S. Mailthody, K. Date, Z. Qureshi, C. Pearson, R. Nagi, J. Xiong, and W.-m. Hwu, "Collaborative (CPU + GPU) algorithms for triangle counting and truss decomposition," in *2018 IEEE High Performance Extreme Computing Conference (HPEC)*, 2018.

[39] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford Large Network Dataset Collection," 2015.

[40] F. N. Paravecino and D. Kaeli, "Accelerated connected component labeling using CUDA framework," in *International Conference on Computer Vision and Graphics*. Springer, 2014, pp. 502–509.

[41] E. De Doncker, J. Kapenga, and R. Assaf, "Monte Carlo automatic integration with dynamic parallelism in CUDA," in *Numerical Computations with GPUs*. Springer, 2014, pp. 273–298.

[42] Z. Wu, Y. Liu, Y. Liang, and J. Sun, "GPU accelerated counterexample generation in LTL model checking," in *International Conference on Formal Engineering Methods*. Springer, 2014, pp. 413–429.

[43] L. Oden, B. Klenk, and H. Fröning, "Energy-efficient stencil computations on distributed GPUs using dynamic parallelism and GPU-controlled communication," in *Energy Efficient Supercomputing Workshop (E2SC), 2014*. IEEE, 2014, pp. 31–40.

[44] V. Mehta, "Exploiting CUDA dynamic parallelism for low power arm based prototypes," in *GPU Technology Conference, San Jose*, 2015.

[45] J. Aliaga, D. Davidović, J. Pérez, and E. S. Quintana-Ortí, "Harnessing CUDA dynamic parallelism for the solution of sparse linear systems," *Parallel Computing: On the Road to Exascale*, vol. 27, p. 217, 2016.

[46] M. Abdellah, A. Eldeib, and M. I. Owis, "GPU acceleration for digitally reconstructed radiographs using bindless texture objects and CUDA/OpenGL interoperability," in *Engineering in Medicine and Biology Society (EMBC), 2015 37th Annual International Conference of the IEEE*. IEEE, 2015, pp. 4242–4245.

[47] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra, "Performance, design, and autotuning of batched GEMM for GPUs," in *International Conference on High Performance Computing*. Springer, 2016, pp. 21–38.

[48] Y. Zhang, J. Liu, X. Li, and Y. Wang, "Fast processing method to generate gigabyte computer generated holography for three-dimensional dynamic holographic display," *Chinese Optics Letters*, vol. 14, no. 3, p. 030901, 2016.

[49] M. Alandoli, M. Al-Ayyoub, M. Al-Smadi, Y. Jararweh, and E. Benkhelifa, "Using dynamic parallelism to speed up clustering-based community detection in social networks," in *Future Internet of Things and Cloud Workshops (FiCloudW), IEEE International Conference on*. IEEE, 2016, pp. 240–245.

[50] M. Al-Ayyoub, M. Al-andoli, Y. Jararweh, M. Smadi, and B. Gupta, "Improving fuzzy c-mean-based community detection in social networks using dynamic parallelism," *Computers & Electrical Engineering*, 2018.

[51] A. F. Sibero, O. S. Sitompul, and M. K. Nasution, "Enhancing performance of parallel self-organizing map on large dataset with dynamic parallel and hyper-q," *Data Science: Journal of Computing and Applied Informatics*, vol. 2, no. 02, pp. 62–73, 2018.

[52] Y. Tian, C. Taylor, and Y. Ji, "Improving the performance of the CamShift algorithm using dynamic parallelism on GPU," in *Information Technology-New Generations*. Springer, 2018, pp. 667–675.

[53] A. Capozzoli, C. Curcio, A. Liseno, and G. Toso, "Speeding up aperiodic reflectarray antenna analysis by CUDA dynamic parallelism," in *Numerical Electromagnetic Modeling and Optimization for RF, Microwave, and Terahertz Applications (NEMO), 2014 International Conference on*. IEEE, 2014, pp. 1–4.

[54] A. Capozzoli, C. Curcio, A. Liseno, and G. Toso, "Fast, phase-only synthesis of aperiodic reflectarrays using NUFFTs and CUDA," *Progress In Electromagnetics Research*, vol. 156, pp. 83–103, 2016.

[55] G. Mei, "Evaluating the power of GPU acceleration for IDW interpolation algorithm," *The Scientific World Journal*, vol. 2014, 2014.

[56] G. Mei and H. Tian, "Impact of data layouts on the efficiency of GPU-accelerated IDW interpolation," *SpringerPlus*, vol. 5, no. 1, p. 104, 2016.

[57] A. O. Adeyemi-Ejeye and S. Walker, "4kUHD H264 wireless live video streaming using CUDA," *Journal of Electrical and Computer Engineering*, vol. 2014, p. 2, 2014.

[58] S. Lai, G. Lai, F. Lu, G. Shen, J. Jin, and X. Lin, "A BSP model graph processing system on many cores," *Cluster Computing*, vol. 20, no. 2, pp. 1359–1377, 2017.

[59] F. Wang, J. Dong, and B. Yuan, "Graph-based substructure pattern mining using CUDA dynamic parallelism," in *International Conference on Intelligent Data Engineering and Automated Learning*. Springer, 2013, pp. 342–349.

[60] M. Plauth, F. Feinbube, F. Schlegel, and A. Polze, "Using dynamic parallelism for fine-grained, irregular workloads: A case study of the N-queens problem," in *2015 Third International Symposium on Computing and Networking (CANDAR)*. IEEE, 2015, pp. 404–407.

[61] M. Plauth, F. Feinbube, F. Schlegel, and A. Polze, "A performance evaluation of dynamic parallelism for fine-grained, irregular workloads," *International Journal of Networking and Computing*, vol. 6, no. 2, pp. 212–229, 2016.

[62] T. Carneiro Pessoa, J. Gmys, F. H. de Carvalho Júnior, N. Melab, and D. Tuyttens, "GPU-accelerated backtracking using CUDA dynamic parallelism," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 9, p. e4374, 2018.

[63] T. Carneiro, J. Gmys, N. Melab et al., "Dynamic configuration of CUDA runtime variables for CDP-based divide-and-conquer algorithms," in *13th International Meeting on High Performance Computing for Computational Science*, 2018.

[64] A. A. Elnaggar, M. Gadallah, M. A. Aziem, and H. El-Deeb, "Enhanced parallel negamax tree search algorithm on GPU," in *Progress in Informatics and Computing (PIC), 2014 International Conference on.* IEEE, 2014, pp. 546–550.

[65] J. Dong, F. Wang, and B. Yuan, "Accelerating BIRCH for clustering large scale streaming data using CUDA dynamic parallelism," in *International Conference on Intelligent Data Engineering and Automated Learning.* Springer, 2013, pp. 409–416.

[66] J. DiMarco and M. Taufer, "Performance impact of dynamic parallelism on different clustering algorithms," in *Modeling and Simulation for Defense Systems and Applications VIII*, vol. 8752. International Society for Optics and Photonics, 2013, p. 87520E.

[67] F. Wang and B. Yuan, "Parallel frequent pattern mining without candidate generation on GPUs," in *Data Mining Workshop (ICDMW), 2014 IEEE International Conference on.* IEEE, 2014, pp. 1046–1052.

[68] E. J. Houtgast, V.-M. Sima, K. Bertels, and Z. Al-Ars, "GPU-accelerated BWA-MEM genomic mapping algorithm using adaptive load balancing," in *International Conference on Architecture of Computing Systems.* Springer, 2016, pp. 130–142.

[69] E. J. Houtgast, V. Sima, K. Bertels, and Z. Al-Ars, "An efficient GPU-accelerated implementation of genomic short read mapping with BWA-MEM," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 4, pp. 38–43, 2017.

[70] B. Kasap and A. J. van Opstal, "Dynamic parallelism for synaptic updating in GPU-accelerated spiking neural network simulations," *Neurocomputing*, vol. 302, pp. 55–65, 2018.

[71] I. Harb and W.-C. Feng, "Characterizing performance and power towards efficient synchronization of GPU kernels," in *Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2016 IEEE 24th International Symposium on.* IEEE, 2016, pp. 451–456.

[72] C. Liao, Y. Yan, B. R. De Supinski, D. J. Quinlan, and B. Chapman, "Early experiences with the OpenMP accelerator model," in *International Workshop on OpenMP.* Springer, 2013, pp. 84–98.

[73] C. Bertolli, S. F. Antao, A. E. Eichenberger, K. O'Brien, Z. Sura, A. C. Jacob, T. Chen, and O. Sallenave, "Coordinating GPU threads for OpenMP 4.0 in LLVM," in *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC*. IEEE Press, 2014, pp. 12–21.

[74] G. Ozen, E. Ayguade, and J. Labarta, "Exploring dynamic parallelism in OpenMP," in *Proceedings of the Second Workshop on Accelerator Programming using Directives*. ACM, 2015, p. 5.

[75] F. Khorasani, B. Rowe, R. Gupta, and L. N. Bhuyan, "Eliminating intra-warp load imbalance in irregular nested patterns via collaborative task engagement," in *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE, 2016, pp. 524–533.

[76] S. Laine, T. Karras, and T. Aila, "Megakernels considered harmful: Wavefront path tracing on GPUs," in *Proceedings of the 5th High-Performance Graphics Conference*. ACM, 2013, pp. 137–143.

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

[78] D. Li, H. Wu, and M. Becchi, "Exploiting dynamic parallelism to efficiently support irregular nested loops on GPUs," in *Proceedings of the 2015 International Workshop on Code Optimisation for Multi and Many Cores*. ACM, 2015, p. 5.

[79] D. Li, H. Wu, and M. Becchi, "Nested parallelism on GPU: Exploring parallelization templates for irregular loops and recursive computations," in *Parallel Processing (ICPP), 2015 44th International Conference on*. IEEE, 2015, pp. 979–988.

[80] H. Wu, D. Li, and M. Becchi, "Compiler-assisted workload consolidation for efficient dynamic parallelism on GPU," *arXiv preprint arXiv:1606.08150*, 2016.

[81] 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 *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–12.

[82] J. Zhang, A. M. Aji, M. L. Chu, H. Wang, and W.-c. Feng, "Taming irregular applications via advanced dynamic parallelism on GPUs," in *Proceedings of the ACM International Conference on Computing Frontiers*, 2018.

[83] J. Zhang, “Transforming and optimizing irregular applications for parallel architectures,” Ph.D. dissertation, Virginia Tech, 2018.

[84] J. Wang, “Acceleration and optimization of dynamic parallelism for irregular applications on GPUs,” Ph.D. dissertation, Georgia Institute of Technology, 2016.

[85] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili, “Kernel weaver: Automatically fusing database primitives for efficient GPU computation,” in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture.* IEEE Computer Society, 2012, pp. 107–118.

[86] M. I. Frank, “SUDS: Automatic parallelization for raw processors,” Ph.D. dissertation, Massachusetts Institute of Technology, 2003.

[87] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August, “Decoupled software pipelining with the synchronization array,” in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques.* IEEE Computer Society, 2004, pp. 177–188.

[88] B. Ren, Y. Jo, S. Krishnamoorthy, K. Agrawal, and M. Kulkarni, “Efficient execution of recursive programs on commodity vector hardware,” in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, 2015, pp. 509–520.

[89] S. Gupta, R. Shrivastava, and V. K. Nandivada, “Optimizing recursive task parallel programs,” in *Proceedings of the International Conference on Supercomputing.* ACM, 2017, p. 11.

[90] B. Ren, S. Krishnamoorthy, K. Agrawal, and M. Kulkarni, “Exploiting vector and multicore parallelism for recursive, data- and task-parallel programs,” in *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP ’17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3018743.3018763 pp. 117–130.

[91] K. Sundararajah, L. Sakka, and M. Kulkarni, “Locality transformations for nested recursive iteration spaces,” in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3037697.3037720 pp. 281–295.

[92] J. Lifflander and S. Krishnamoorthy, "Cache locality optimization for recursive programs," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3062341.3062385 pp. 1–16.

[93] L. Sakka, K. Sundararajah, and M. Kulkarni, "Treefuser: a framework for analyzing and fusing general recursive tree traversals," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 76, 2017.

[94] P. Ramarao, "CUDA 10 features revealed: Turing, CUDA graphs, and more," https://devblogs.nvidia.com/cuda-10-features-revealed, 2018.

[95] 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*, ser. GPGPU '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2884045.2884052 pp. 53–62.

[96] A. 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 *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture.* ACM, 2017, pp. 600–611.

[97] J. Liu, "Efficient synchronization for GPGPU," Ph.D. dissertation, University of Pittsburgh, 2018.

[98] 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.

[99] Z. Zheng, C. Oh, J. Zhai, X. Shen, Y. Yi, and W. Chen, "Versapipe: A versatile programming framework for pipelined computing on GPU," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture.* ACM, 2017, pp. 587–599.

[100] W.-m. Hwu, L.-W. Chang, H.-S. Kim, A. Dakkak, and I. El Hajj, "Transitioning HPC software to exascale heterogeneous computing," in *Computational Electromagnetics International Workshop (CEM), 2015.* IEEE, 2015, pp. 1–2.

[101] L.-W. Chang, I. El Hajj, H.-S. Kim, J. Gómez-Luna, A. Dakkak, and W.-m. Hwu, "A programming system for future proofing performance critical libraries," *ACM SIGPLAN Notices*, vol. 51, no. 8, p. 32, 2016.

[102] L.-W. Chang, I. E. Hajj, C. Rodrigues, J. Gómez-Luna, and W.-m. Hwu, "Efficient kernel synthesis for performance portable programming," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 2016, p. 12.

[103] L.-W. Chang, "Toward performance portability for CPUs and GPUs through algorithmic compositions," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2017.

[104] S. Garcia De Gonzalo, S. Huang, J. Gómez-Luna, S. Hammond, O. Mutlu, and W.-m. Hwu, "Automatic generation of warp-level primitives and atomic operations for fast-portable GPU reductions," in *Proceedings of the 17th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2019.

[105] S. Mukherjee, X. Gong, L. Yu, C. McCardwell, Y. Ukidave, T. Dao, F. N. Paravecino, and D. Kaeli, "Exploring the features of OpenCL 2.0," in *Proceedings of the 3rd International Workshop on OpenCL*. ACM, 2015, p. 5.

[106] Y. Sun, X. Gong, A. K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. McCardwell, A. Villegas, and D. Kaeli, "Hetero-mark, a benchmark suite for CPU-GPU collaborative computing," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2016, pp. 1–10.

[107] S. Mukherjee, Y. Sun, P. Blinzer, A. K. Ziabari, and D. Kaeli, "A comprehensive performance analysis of HSA and OpenCL 2.0," in *Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 183–193.

[108] J. Gómez-Luna, I. El Hajj, L.-W. Chang, V. García-Floreszx, S. G. de Gonzalo, T. B. Jablin, A. J. Pena, and W.-m. Hwu, "Chai: collaborative heterogeneous applications for integrated-architectures," in *Performance Analysis of Systems and Software (ISPASS), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 43–54.

[109] L.-W. Chang, J. Gómez-Luna, I. El Hajj, S. Huang, D. Chen, and W.-m. Hwu, "Collaborative computing for heterogeneous integrated systems," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. ACM, 2017, pp. 385–388.

[110] M. D. Sinclair, J. Alsop, and S. V. Adve, "Heterosync: A benchmark suite for fine-grained synchronization on tightly coupled GPUs," in *Workload Characterization (IISWC), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 239–249.

[111] M. D. Sinclair, "Efficient coherence and consistency for specialized memory hierarchies," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2017.

[112] R. Palomar, J. Gómez-Luna, F. A. Cheikh, J. Olivares-Bueno, and O. J. Elle, "High-performance computation of Bézier surfaces on parallel and heterogeneous platforms," *International Journal of Parallel Programming*, pp. 1–28, 2017.

[113] S. Che, M. Orr, and J. Gallmeier, "Work stealing in a shared virtual-memory heterogeneous environment: A case study with betweenness centrality," in *Proceedings of the Computing Frontiers Conference*. ACM, 2017, pp. 164–173.

[114] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, "gem5-gpu: A heterogeneous CPU-GPU simulator," *IEEE Computer Architecture Letters*, vol. 14, no. 1, pp. 34–36, 2015.

[115] J. Lowe-Power, *On Heterogeneous Compute and Memory Systems*. The University of Wisconsin-Madison, 2017.

[116] Y. Sun, T. Baruah, S. A. Mojumder, S. Dong, R. Ubal, X. Gong, S. Treadway, Y. Bao, V. Zhao, J. L. Abellán et al., "MGSim + MGMark: A framework for multi-GPU system research," *arXiv preprint arXiv:1811.02884*, 2018.

[117] K. L. Spafford, J. S. Meredith, S. Lee, D. Li, P. C. Roth, and J. S. Vetter, "The tradeoffs of fused memory hierarchies in heterogeneous computing architectures," in *Proceedings of the 9th conference on Computing Frontiers*. ACM, 2012, pp. 103–112.

[118] K. Lee, H. Lin, and W.-c. Feng, "Performance characterization of data-intensive kernels on amd fusion architectures," *Computer Science-Research and Development*, vol. 28, no. 2-3, pp. 175–184, 2013.

[119] Q. Zhu, B. Wu, X. Shen, K. Shen, L. Shen, and Z. Wang, "Understanding co-run performance on CPU-GPU integrated processors: Observations, insights, directions," *Frontiers of Computer Science*, vol. 11, no. 1, pp. 130–146, 2017.

[120] M. Dashti and A. Fedorova, "Analyzing memory management methods on integrated CPU-GPU systems," in *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*, ser. ISMM 2017. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3092255.3092256 pp. 59–69.

[121] J. Fang, H. Chen, and J. Mao, "Understanding data partition for applications on CPU-GPU integrated processors," in *International Conference on Mobile Ad-Hoc and Sensor Networks.* Springer, 2017, pp. 426–434.

[122] Y. Sun, S. Mukherjee, T. Baruah, S. Dong, J. Gutierrez, P. Mohan, and D. Kaeli, "Evaluating performance tradeoffs on the Radeon Open Compute platform," in *Performance Analysis of Systems and Software (ISPASS), 2018 IEEE International Symposium on.* IEEE, 2018, pp. 209–218.

[123] S. Tang, B. He, S. Zhang, and Z. Niu, "Elastic multi-resource fairness: balancing fairness and efficiency in coupled CPU-GPU architectures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* IEEE Press, 2016, p. 75.

[124] N. Farooqui, I. Roy, Y. Chen, V. Talwar, and K. Schwan, "Accelerating graph applications on integrated GPU platforms via instrumentation-driven optimizations," in *Proceedings of the ACM International Conference on Computing Frontiers.* ACM, 2016, pp. 19–28.

[125] F. Zhang, B. Wu, J. Zhai, B. He, and W. Chen, "Finepar: Irregularity-aware fine-grained workload partitioning on integrated architectures," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization.* IEEE Press, 2017, pp. 27–38.

[126] Y. Cho, F. Negele, S. Park, B. Egger, and T. R. Gross, "On-the-fly workload partitioning for integrated CPU/GPU architectures," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques.* ACM, 2018, p. 21.

[127] T. Baruah, Y. Sun, S. Dong, D. Kaeli, and N. Rubin, "Airavat: Improving energy efficiency of heterogeneous applications," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE).* IEEE, 2018, pp. 731–736.

[128] T. Baruah, "Energy efficient execution of heterogeneous applications," Ph.D. dissertation, Northeastern University, 2017.

[129] J. Veselỳ, A. Basu, A. Bhattacharjee, G. H. Loh, M. Oskin, and S. K. Reinhardt, "Generic system calls for GPUs," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA).* IEEE, 2018, pp. 843–856.

[130] A. Basu, J. L. Greathouse, G. Venkataramani, and J. Veselỳ, "Interference from GPU system service requests," in *Workload Characterization (IISWC), 2018 IEEE International Symposium on.* IEEE, 2018.

[131] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous system coherence for integrated CPU-GPU systems," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2013, pp. 457–467.

[132] J. Alsop, M. D. Sinclair, and S. V. Adve, "Spandex: a flexible interface for efficient heterogeneous coherence," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 2018, pp. 261–274.

[133] Y. Hao, Z. Fang, G. Reinman, and J. Cong, "Supporting address translation for accelerator-centric architectures," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 37–48.

[134] V. García, J. Gomez-Luna, T. Grass, A. Rico, E. Ayguade, and A. J. Pena, "Evaluating the effect of last-level cache sharing on integrated GPU-CPU systems with heterogeneous applications," in *Workload Characterization (IISWC), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 1–10.

[135] V. García Flores, "Memory hierarchies for future HPC architectures," Ph.D. dissertation, Universitat Politècnica de Catalunya, 2017.

[136] A. Patil and R. Govindarajan, "HAShCache: Heterogeneity-aware shared DRAMCache for integrated heterogeneous systems," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 4, p. 51, 2017.

[137] V. García-Flores, E. Ayguade, and A. J. Peña, "Efficient data sharing on heterogeneous systems," in *Parallel Processing (ICPP), 2017 46th International Conference on*. IEEE, 2017, pp. 121–130.

[138] J. Zhan, O. Kayıran, G. H. Loh, C. R. Das, and Y. Xie, "OSCAR: Orchestrating STT-RAM cache traffic for heterogeneous CPU-GPU architectures," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–13.

[139] T. Vijayaraghavany, Y. Eckert, G. H. Loh, M. J. Schulte, M. Ignatowski, B. M. Beckmann, W. C. Brantley, J. L. Greathouse, W. Huang, A. Karunanithi et al., "Design and analysis of an apu for exascale computing," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 85–96.

[140] L. E. Olson, J. Power, M. D. Hill, and D. A. Wood, "Border control: Sandboxing accelerators," in *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 2015, pp. 470–481.

[141] C. Erb, M. Collins, and J. L. Greathouse, "Dynamic buffer overflow detection for GPGPUs," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization.* IEEE Press, 2017, pp. 61–73.

[142] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950379 pp. 91–104.

[143] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950380 pp. 105–118.

[144] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2660193.2660224 pp. 433–452.

[145] J. Izraelevitz, H. Mendes, and M. L. Scott, "Linearizability of persistent memory objects under a full-system-crash failure model," in *International Symposium on Distributed Computing.* Springer, 2016, pp. 313–327.

[146] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "NV-Tree: Reducing consistency cost for NVM-based single level systems." in *FAST*, vol. 15, 2015, pp. 167–181.

[147] B. Debnath, A. Haghdoost, A. Kadav, M. G. Khatib, and C. Ungureanu, "Revisiting hash table design for phase change memory," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 2, pp. 18–26, 2016.

[148] I. El Hajj, A. Merritt, G. Zellweger, D. Milojicic, R. Achermann, P. Faraboschi, W.-m. Hwu, T. Roscoe, and K. Schwan, "Spacejmp: programming with multiple virtual address spaces," *ACM SIGOPS Operating Systems Review*, vol. 50, no. 2, pp. 353–368, 2016.

[149] I. El Hajj, T. B. Jablin, D. Milojicic, and W.-m. Hwu, "SAVI objects: sharing and virtuality incorporated," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 45, 2017.

[150] N. Cohen, D. T. Aksun, and J. R. Larus, "Object-oriented recovery for non-volatile memory," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 153:1–153:22, Oct. 2018. [Online]. Available: http://doi.acm.org/10.1145/3276523

[151] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm, "Makalu: Fast recoverable allocation of non-volatile memory," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2016. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2983990.2984019 pp. 677–694.

[152] P. Bruel, S. R. Chalamalasetti, C. Dalton, I. El Hajj, A. Goldman, C. Graves, W.-m. Hwu, P. Laplante, D. Milojicic, G. Ndu et al., "Generalize or die: Operating systems support for memristor-based accelerators," in *Rebooting Computing (ICRC), 2017 IEEE International Conference on.* IEEE, 2017, pp. 1–8.

[153] J. Ambrosi, A. Ankit, R. Antunes, S. R. Chalamalasetti, S. Chatterjee, I. E. Hajj, G. Fachini, P. Faraboschi, M. Foltin, S. Huang, W.-m. Hwu, G. Knuppe, S. V. Lakshminarasimha, D. Milojicic, M. Parthasarathy, F. Ribeiro, L. Rosa, K. Roy, P. Silveira, and J. P. Strachan, "Hardware-software co-design for an analog-digital accelerator for machine learning," in *Rebooting Computing (ICRC), 2018 IEEE International Conference on.* IEEE, 2018.

[154] A. Ankit, I. El Hajj, S. R. Chalamalasetti, G. Ndu, M. Foltin, R. S. Williams, P. Faraboschi, W.-m. Hwu, J. P. Strachan, K. Roy, and D. Milojicic, "PUMA: A programmable ultra-efficient memristor-based accelerator for machine learning inference," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[155] W.-m. Hwu, I. El Hajj, S. Garcia de Gonzalo, C. Pearson, N. S. Kim, D. Chen, J. Xiong, and Z. Sura, "Rebooting the data access hierarchy of computing systems," in *IEEE International Conference on Rebooting Computing (ICRC).* IEEE, 2017.