AMERICAN UNIVERSITY OF BEIRUT

A COMPILER FRAMEWORK FOR OPTIMIZING DYNAMIC PARALLELISM ON GPUS

by
MHD GHAITH OLABI

A thesis
submitted in partial fulfillment of the requirements
for the degree of Master of Science
to the Department of Computer Science
of the Faculty of Arts and Sciences
at the American University of Beirut

Beirut, Lebanon
January 2021

AMERICAN UNIVERSITY OF BEIRUT

A COMPILER FRAMEWORK FOR OPTIMIZING DYNAMIC PARALLELISM ON GPUS

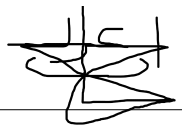
by

MHD GHAITH OLABI

Approved by:

Dr. Izzat El Hajj, Assistant Professor — Advisor

Computer Science

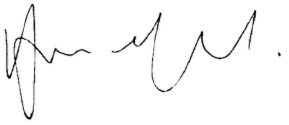


Dr. George Turkiyyah, Professor — Member of Committee

Computer Science

Dr. Amer E. Mouawad, Assistant Professor — Member of Committee

Computer Science



Date of thesis defense: January 26, 2021

AMERICAN UNIVERSITY OF BEIRUT

THESIS, DISSERTATION, PROJECT RELEASE FORM

Olabi	Mhd Ghaith	Omar
Last	First	Middle

Student Name: _____

○ Master's Thesis ○ Master's Project ○ Doctoral Dissertation

☒ I authorize the American University of Beirut to: (a) reproduce hard or electronic copies of my thesis, dissertation, or project; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes.

☐ I authorize the American University of Beirut, to: (a) reproduce hard or electronic copies of it; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes after: **One ___ year from the date of submission of my thesis, dissertation or project.**
Two ___ years from the date of submission of my thesis , dissertation or project.
Three ___ years from the date of submission of my thesis , dissertation or project.

Mhd Ghaith Olabi — 8, February, 2021

Signature Date

This form is signed when submitting the thesis, dissertation, or project to the University Libraries

Acknowledgements

With the utmost of appreciation and gratitude, I would like to thank my advisor, Professor Izzat El Hajj for his constant support and understanding. This thesis would have not been possible without his mentorship. Thanks also to the committee, Professor George Turkiyyah and Professor Amer E. Mouawad for their thorough feedback and constructive discussions.

My deepest appreciation goes to my family for their love and support.

An Abstract of the Thesis of

Mhd Ghaith Olabi for Master of Science
Major: Computer Science

Title: A Compiler Framework for Optimizing Dynamic Parallelism on GPUs

Dynamic Parallelism on GPUs provides the means for the GPU to generate work for itself instead of relying on the CPU where a thread running on the GPU can also launch grids of threads that also run on the GPU. This mechanism is particularly useful with applications where the required parallelism is dynamic and unknown on execution. However, multiple performance issues arise when using dynamic parallelism. First, the massive number of small launches incurs massive overhead. Second, the high number of launches is bottlenecked by the limited numbers of simultaneously executable kernels. Third, the small grids occupying the GPU causes the device to be underutilized. In this thesis, we aim to propose a framework that optimizes dynamic parallelism performance by applying three key compiler optimization techniques: threshold, coarsening, and aggregation. Thresholding serializes the kernel work when the dynamic parallelism benefit is potentially cancelled by the launch overhead. Coarsening allows a single child thread block to sequentially execute the work of multiple other child thread blocks. Aggregation consolidates multiple child grids into a single aggregated grid. We automate these optimizations as separate compiler passes then analyze and evaluate the interactions between them. We also combine them in a single compiler flow, our evaluation on data sets with high parallelism irregularity shows that when our compiler framework is applied on applications with nested parallelism, on average, it achieves 43.0x speedup over applications that uses dynamic parallelism, 8.7x speedup over applications that do not use dynamic parallelism, and 3.6x speedup over applications that use dynamic parallelism with aggregation only. Our evaluation also shows that even with all optimizations applied, on datasets that have low irregularity and low parallelism requirements, dynamic parallelism still performs significantly worse.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Nowadays, mainstream modern computer systems includes both Central Processing Units (CPUs) and Graphics Processing Units (GPUs). CPUs have always been the central general purpose processors in computers whereas GPUs started off as accelerators that can only perform graphical tasks. By design, GPUs are composed from many cores where the architecture is optimized for high throughput achieved by the massive number of operations that can be executed concurrently. Over the past decade, modern GPUs had also proven to be effective as general purpose processors, this drove the development of many tools that simplifies the process of expressing programs that can be executed on GPUs such as CUDA [1]. Distinguished by their ability to execute many tasks at once, GPUs are often used to accelerate applications with parallel computing patterns. The most straightforward parallel patterns are the ones where the work is flat, with minimal dependencies between workers such as drawing pixels on the screen. The parallel portions of these applications called kernels and are written

in CUDA, gets executed on the GPU (device) while the sequential portions are often executed on the CPU (host). In CUDA, Execution is organized in the form of an array of thread blocks called a grid. A thread block is a batch of threads that can cooperate with each other. Fig. 1.1 shows how a typical program written in CUDA runs on the GPU. The host (CPU) issues work to the device by calling a kernel with a set of configuration and parameters. The configuration specifies the parallelism requirements, and the parameters are the data needed to execute the work. GPUs are also leveraged to accelerate many types of computing patterns, one of these patterns is *Nested Parallelism* in which the program can exploit hierarchical levels of parallelism.

Dynamic parallelism is a technique that provides the ability for the GPU to launch work for itself without relying on the CPU, dynamically, simultaneously, and independently [2]. The usage of this interface makes it simpler for the developers to write simpler programs that implement various algorithms with nested parallelism, especially in the cases where the amount of nested parallelism is irregular and can only be discovered dynamically through execution. An example of such algorithms is graph algorithms where a parent thread visiting a vertex in the graph might want to perform some work for each of its neighbours, in such case, dynamic parallelism is useful such as the parent thread would launch a grid of thread, whereby a thread is assigned for each neighbour vertex, each thread would concurrently execute the work.

However, in practice, as shown by prior work [3], dynamic parallelism exhibits inefficiency due to limitations in hardware. First, the device can only execute a limited number of grids and/or thread blocks simultaneously in parallel, when this limit is exceeded the device serializes the remaining grids and/or blocks. Second, whenever the GPU API is called to launch a grid there is an incurred overhead.

Both of these hardware attributes combined are key to understanding the exhibited inefficiency. The launch overhead is exceedingly high and significant when a massive number of small grids are launched. Moreover, these small grids will occupy all the device resources without fully utilizing them because of the small parallelism requirements. As a result, instead of executing all the thread blocks in parallel, the device would have to serialize many of them in hardware incurring additional overhead due to the limited number of concurrently executable grids.

To mitigate the overhead of dynamic parallelism, many hardware and software approaches were proposed. On the software side, CUDA-NP [4], Free Launch [5], KLAP [6] and more. CUDA-NP transforms the source code where a potentially large number of threads is launched upfront, these threads are segmented into two categories, master and slave threads, where essentially, master threads will be doing the work of parent threads discovering the work and slave threads perform the actual work. Free launch, on the other hand, takes a scheduling approach where it includes multiple techniques to reuse parent threads having them execute child work in a load-balanced way, either sequentially or in parallel. Both of these approaches mitigate dynamic parallelism by avoiding it entirely. On the other hand, KLAP consolidates kernel launches being launched by multiple parent threads into a single grid. This approach mitigates the overhead by reducing the number of grids to be launched. Hence, reducing the launch overhead and allowing higher hardware utilization.

As for the hardware approaches, SPAWN [7] mitigates the overhead by assessing the profitability of the dynamic launches based on the GPU state, then advises the programmer to either launch or take an alternative method such as serializing the work in the parent thread. Dynamic Thread Block Launch (DTBL) [8, 9] allows lightweight launches through extending the grids on the

fly. LASER [10] enhances dynamic parallelism with (locality aware) scheduling policies that enhances performance by being aware of the data references locality, and LaPerm [11] adds locality aware scheduling on top of DTBL.

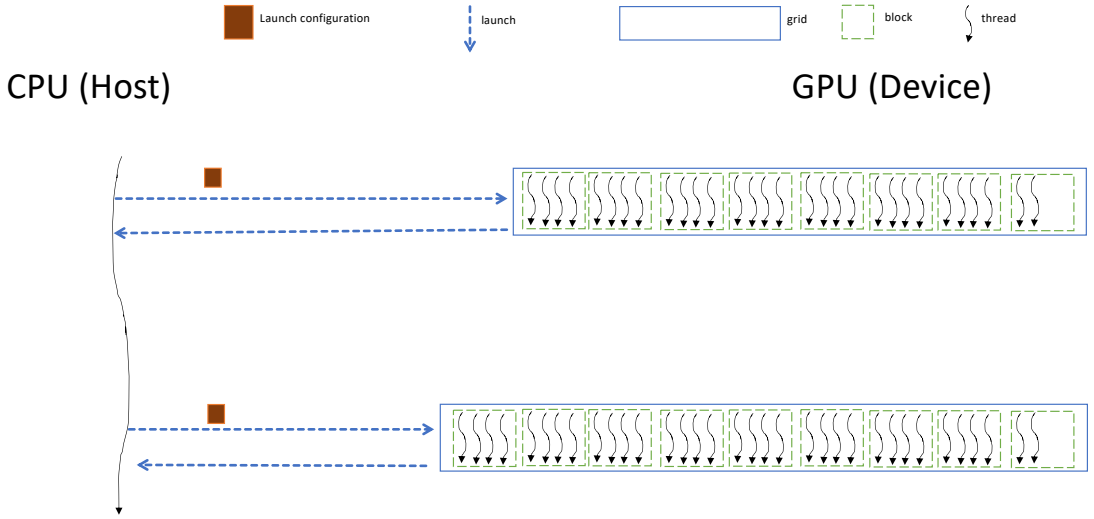


Figure 1.1: A Typical CUDA Application

1.2 Goals and Contributions

We propose a compiler framework for optimizing dynamic parallelism in applications with *irregular nested parallelism*. The framework features three key optimizations: *thresholding, coarsening, and aggregation*. Thresholding allows a dynamic launch in the parent thread only if the number of child threads exceeds a certain threshold, the work would be serialized in the parent thread otherwise. This optimization aims to minimize the overhead incurred by the small launched grids and underutilized thread blocks. Moreover, when combined with aggregation, this optimization ensures that only large grids are aggregated into the dynamic launches.

Second, coarsening optimization involves combining the work of multiple

thread blocks into a single block. The first benefit of this optimization is that it reduces the number of thread blocks that need to be scheduled, this is potentially useful when the device is oversubscribed with the high number of launches. Moreover, coarsening integrates well with the aggregation optimization where it is used to amortize the cost of aggregation across multiple blocks.

Third, aggregation is an optimization where multiple grids being launched by various parent threads are consolidated into a single one. Instead of having each thread configure and launch a grid, threads collaborate with each other and launch a single aggregated grid. The benefit of this optimization is that it reduces the total number of grids which allows higher device utilization. In this optimization, we use prior work [6] in the compiler flow.

In this thesis we aim to deliver the following contributions:

- Present a compiler transformation that automates *thresholding* for dynamic parallelism
- Present a compiler transformation that automates *coarsening* in the context of dynamic parallelism
- Introduce a framework that combines *thresholding, coarsening, and aggregation*

Chapter 2

Dynamic Parallelism Optimizations

2.1 Background on Dynamic Parallelism

Dynamic parallelism allows the programmer to configure and launch grids from GPU threads instead of switching the context to the CPU then having the CPU thread configure and launch the grid. A practical example of how dynamic parallelism could be used is shown in Fig. 2.1. In this particular example, four threads are executing on the GPU and each one of these threads discovers nested work that needs to be executed in parallel, however, the amount of parallelism required per nested work is different. Hence, each GPU thread configures a different nested launch in a separate grid that is scheduled to be run on the GPU. The above example simulates a graph application algorithm where each parent is set to process a vertex and for each of its neighbours, the vertex may want to perform some operation. Instead of the parent thread serially processing these operations it decides to perform the nested work in parallel, and each of these parents would

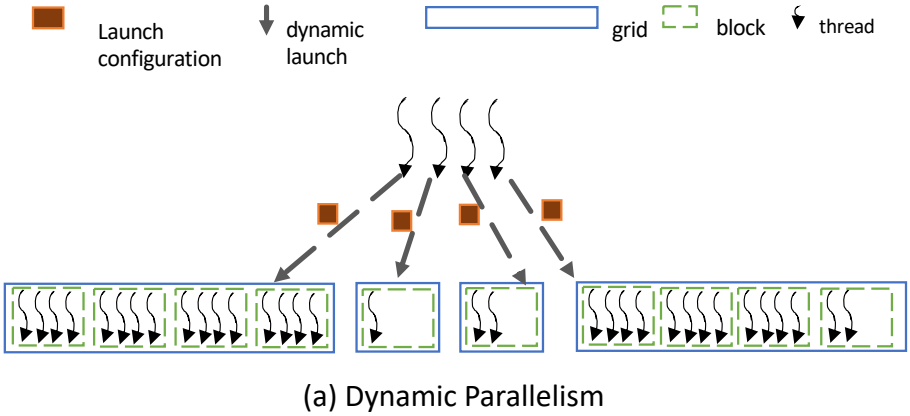


Figure 2.1: Dynamic Parallaleism Example

provide a different set of configurations and parameters for the nested launches. For example, different vertices may have a different set of neighbours and a different number of neighbours to work on. Hence the grid will be provided with a different grid size (launch configurations) and parameters.

As shown in Fig. 2.1 different parent threads may configure launches differently whereby a massive number of child grids may be launched and many of them could be small. An example where such a case would occur would be a graph that has thousands of vertices and many of these vertices only have a small number of neighbours. As a result, a massive number of grids that include very few child threads would be scheduled to launch. In this case, the massive number of launches will cause a massive launch overhead, and these small grids could be serialized by the hardware due to the limited amount of grids that could be run simultaneously causing the device to be underutilized, moreover, the launch overhead of these small threads could cancel the benefit gained from trying to run them in parallel.

2.2 Thresholding

Thresholding is an optimization that tackles the massive launch overhead by cancelling the small grids being launched. Before committing the launch, the code would look first at the number of threads required in the to-be-launched grid as a measurement of potential profitability, then, it compares this amount of nested work to a certain `threshold`. Based on this comparison, if the work is assessed profitable (exceeding the `threshold`), the code then commits the launch. Otherwise, it runs sequentially on the parent thread. Fig. 2.2(b) shows how thresholding can be applied to the example in Fig. 2.2(a). In this example, there are four parent threads trying to do nested work, two of which only have work that exceeds the threshold. As a result, two parent threads proceed to launch two grids where the other two threads work is serialized in their parent. As aforementioned, when launching the two small grids the benefit gained from running the work in parallel would very likely be cancelled by the launch overhead. Also, if the number of small grids exceeds the finite amount of simultaneously executable grids, this would cause the device to be underutilized. Collectively, this overhead will have us end up with a result that is much worse than just serializing the work.

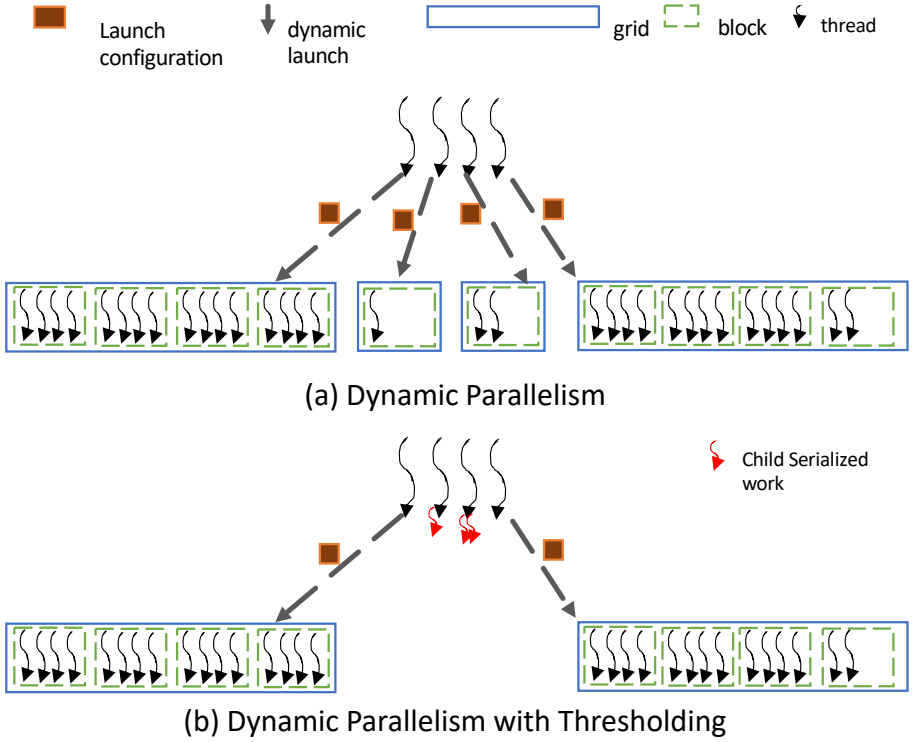


Figure 2.2: Dynamic Parallaleism with Thresholding Example

Thresholding is an optimization that is commonly applied by programmers [12, 13, 14, 7]. However, when applied manually it can cause unwanted effects on code quality such as code duplication. We propose to automate this optimization in a compiler (Section 3.1). However, automating this optimziation does come with its own challenges discussed in sections 3.1.3 and 3.1.2.

2.3 Coarsening

Coarsening is a contextual optimization that can be used to tackle a specific type of overhead or bottleneck depending on the context [15, 16, 17]. Essentially, coarsening assigns the work of multiple thread blocks to one thread block. Effec-

tively, this one thread block would sequentially execute the work of these multiple blocks one after the other. The coarsening optimization is usually done by applying some `coarsening factor`. For example, given a grid that originally consists of 6 thread blocks, after applying a $COARSE\_FACTOR = 2$ the end result would be 3 thread blocks that are assigned the work of 6. Fig. 2.3(b) shows an example of how coarsening can be applied on the example given in Fig. 2.3(a), in this example, we use $COARSE\_FACTOR = 2$ and as a result, each one thread block would execute the work of two thread blocks.

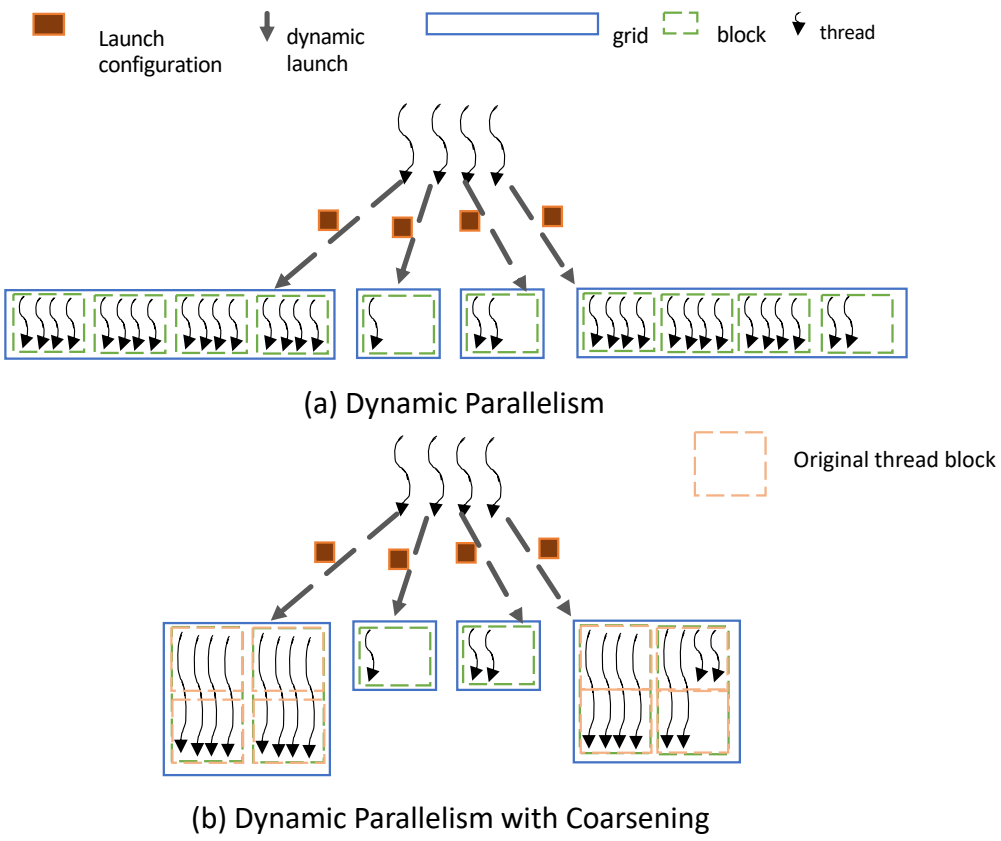


Figure 2.3: Dynamic Parallelism with Coarsening Example

The immediate result of coarsening is that it reduces the total number of

block threads that would be scheduled on the GPU, optimally allowing all thread blocks to run in parallel. For example, when a program oversubscribes the GPU with large thread block requirements that the hardware is not capable of supporting, instead of having these blocks run in parallel, they would be scheduled in-order, sequentially, so that once one finish the other starts. Depending on the context, this end-result can be leveraged to either tackle a specific bottleneck such as hardware limitations or optimize some specific behaviour depending on the program. There are multiple advantages that can be achieved by applying coarsening. First, as aforementioned, when a program oversubscribes the GPU, instead of having the hardware do the serialization and incur an extra overhead we can apply coarsening to remove that extra unwanted overhead. Second, since that work is scheduled in a block level, then executed on a warp level, some warps could finish their work and wait for the other warp in the thread block, by applying coarsening this would allow some warps to proceed to work on the original thread block work before other warps have completed there work. Third, it leaves the option to the developer to factor out common work across the original thread blocks to be done once by the coarsened block having its work cost amortized. For example, in the context of dynamic parallelism, coarsening can potentially further increase the performance benefit of the aggregation optimization by amortizing the cost of the disaggregation logic (detailed in Section 2.4). Another example outside dynamic parallelism would be matrix-matrix multiplication, when assigning a thread block to process multiple output tiles that share input tiles, the cost of loading the input tiles to shared memory is amortized as the tile is loaded once, then reused multiple times [18]. Moreover, given the case of a parallel histogram implementation, programmers often uses privatization which is an optimization that builds a local histogram per thread block to min-

imize the usage of `atomicAdd`, coarsening can further enhance the performance whereby assigning the work of multiple thread blocks to one would allow it to build a larger local histogram in local memory before committing the global histogram which amortizes the cost of `atomicAdds` [19, 20]. on the other hand, the disadvantage of coarsening is that it reduces parallelism, as a result, if the used `coarse factor` is too high this would lead to having the device underutilized with work that could have been running in parallel otherwise, as such, the best coarsening factor in code should be tied to the specific device available resources.

As a result, in the context of dynamic parallelism, when thread block coarsening is applied the number of blocks to be scheduled is reduced. However, the reason we propose to automate it in a compiler and apply this optimization is for the combined potential when applied alongside the `aggregation` optimization discussed in section 2.4.

2.4 Aggregation

While coarsening reduces the total number of thread blocks, aggregation is an optimization that reduces the total number of grids by consolidating multiple grids into a single one. Before committing the launch, instead of having each parent thread discover the nested work and do the launch, they would coordinate between each other to consolidate these launches into a single grid. For example, Fig. 2.4(b) shows an example of how aggregation can be applied on the example given in Fig. 2.4(a). In this example, in the original code, every parent thread discovers some nested work and proceeds to do a dynamic launch, whereas, in the transformed code, the parent threads would coordinate the launch requirements and parameters and consolidate these launches into a single grid.

This coordination usually happens on different scopes, warps, blocks, or a grid, this largely depends on the aggregation implementation. In the case of warps or block, the launch is done dynamically from one GPU thread on behalf of all others whereas if the coordination is done across the whole grid, the launch is delayed until all parent threads discover all dynamic work, aggregated, and then the aggregated grid would be launched from the host. The aggregation optimization is an optimization that has been previously done either manually or using a compiler [12, 13, 14, 6].

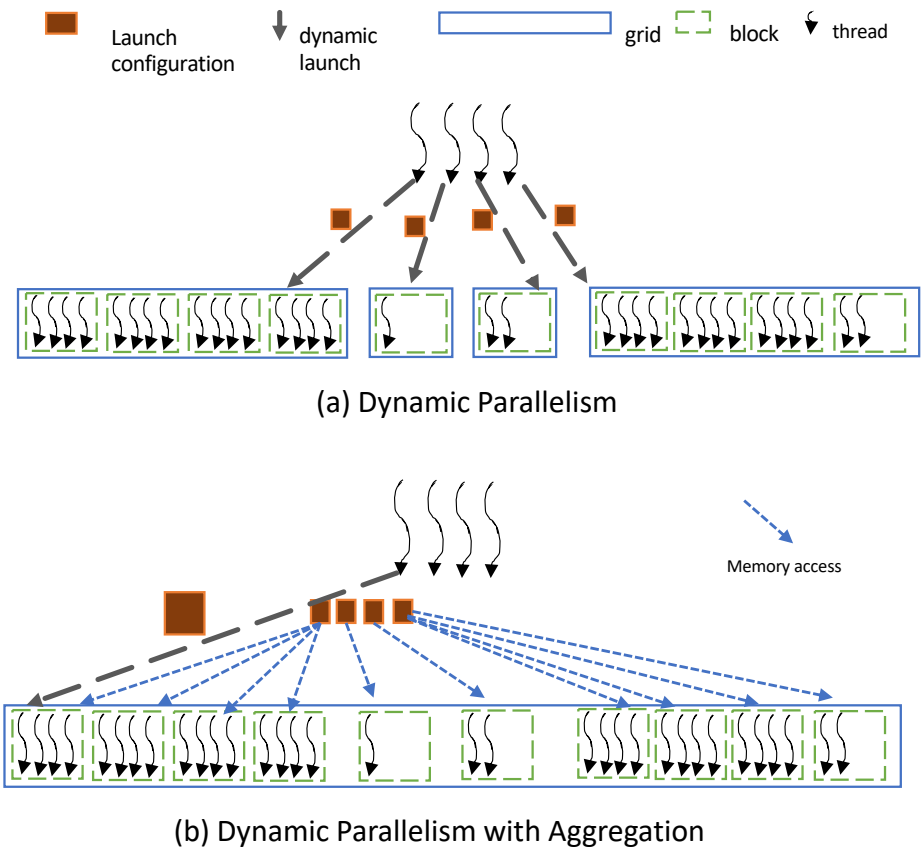


Figure 2.4: Dynamic Parallaleism with Aggregation Example

In the original code, each parent thread directly knows the relevant launch

configuration and parameters and provides them directly to each child grid. However, after the aggregation transformation, the launched grid/s can not only include one set of parameters and launch configuration. Hence, before completing the launch the parent threads must coordinate in order to store their original parameters and configuration in memory then a pointer to this memory location would be passed to the aggregated grid, this work is referred to as `aggregation logic`. Then, after the aggregated grid is launched each child thread must execute a search operation to identify the original parent thread before aggregation in order to load the correct launch configuration and parameters from memory this work is referred to as `disaggregation logic`.

The advantage of the aggregation optimization is that it reduces the number of dynamic launches, which lowers the launch overhead cost alongside maximizing the size of launched grids per scope, allowing better hardware utilization since the probability of hitting the hardware limit of simultaneously executable grids is considerably lower. However, on the other hand, the disadvantages are that it also incurs the overhead of both the `aggregation` and `disaggregation` logic which extra instructions aside from the original code logic.

Chapter 3

The Compiler Framework

Our compiler framework consists of three optimizations: thresholding, coarsening, and aggregation. Each optimization is implemented individually, and separately as a source-to-source transformation pass that takes a `.cu` file and generates a `.cu` file. This ensures that no optimization relies on the other to generate correct code and achieves separation of concerns. Also, they can be extended and configured independently, or can be applied in different order, or can be integrated in different flows.

3.1 Thresholding Transformation

The thresholding optimization transforms the code so that only grids larger than a certain size are launched, otherwise, the work is serialized. This transformation includes three key parts: constructing a serial version of the parallel code to be executed by the parent thread, detecting the amount of work parallel work to be launched by the parent thread, and applying the threshold guard to either perform the launch or call the serial kernel.

Fig. 3.1 shows an example of how the thresholding transformation is applied across both parent and child kernel. The original code before the transformation in Fig. 3.1(a) consisting of two key elements: a parent kernel(lines 04-08) and a child kernel (lines 01-03). the parent kernel configures a child kernel with some grid dimension `gDim` and block dimension `bDim` then launches the child kernel using dynamic parallelism (line 06). The transformed code in Fig. 3.1(b) shows the code after the transformation consisting of three key elements: a parent kernel (lines 19-28), a device serial function (lines 09-15) and a parallel child kernel (lines 16-18). The parent kernel decides to either launch the parallel kernel or the serial function based on some value `__threads` that represents the required number of threads in the dynamic call.

3.1.1 Serial Kernel Construction

To construct the serial version, the child kernel is replicated, a `__serial` suffix is added to its name then transformed from a kernel to a device function by changing its attribute to `__device__`. Then, two parameters are appended to the replicated function: `gDim` which represents the original grid dimension in the parallel kernel, and `bDim` which represents the original block dimension. Then, we insert loops that iterate over the original grid and block dimension, the first loop (line 10) iterates over the grid blocks, whereas the second loop (line 11) iterates over the threads in a block. Finally we replace all the uses of reserved indices and dimension variables such as `blockIdx.x`, `threadIdx.x`, `blockDim.x`, `gridDim.x` with relevant local variables based on the inserted loop/s. For simplicity, the above example shows a 1-dimensional child kernel. Noting that this serialization approach does not work for all cases. The `non-serializeable` cases are discussed in section 3.1.2.

```
01  __global__ child(...) {
02      child body
03  }

04  __global__ parent(...) {
05      ...
06      child <<< gDim, bDim >>> (...);
07      ...
08  }
```

(a) Original Code

```
09  __device__ child_serial(..., dim3 _gDim, dim3 _bDim) {
10      for(_bx = 0; _bx < _gDim.x; ++_bx) {
11          for(_tx = 0; _tx < _bDim.x; ++_tx) {
12              child body // Replace uses of blockIdx.x with _bx,
13          }              // threadIdx.x with _tx, gridDim with
14      }                  // _gDim, and blockDim with _bDim
15  }

16  __global__ child(...) {
17      child body
18  }

19  __global__ parent(...) {
20      ...
21      _threads = ...; // Extracted from gDim expression
22      if(_threads >= _THRESHOLD) {
23          child <<< gDim, bDim >>> (...);
24      } else {
25          child_serial (..., gDim, bDim);
26      }
27      ...
28  }
```

(b) Code after Thresholding Transformation

Figure 3.1: Thresholding Transformation Example

3.1.2 Non-Serializeable Cases

There are multiple cases where constructing a serial kernel automatically is either infeasible or would yield poor results. The first case is that when the parallel kernel requires collaboration between multiple threads, usually, this collaboration is achieved either through performing barrier synchronization across threads via `__syncthreads()` or warp-level primitives. First, code that requires thread collaboration assumes the presence of multiple threads executing the work in parallel which is true in the original code, but not in the transformed code where only one thread, being the parent, will be executing the work. This collaborative behaviour requires threads to wait for each other at every barrier operation which cannot be trivially maintained when serialized, the result of the code transformation described in section 3.1.1 would execute the work of every child thread to completion before moving to the next child's thread work which effectively would not be able to port the collaborative nature of the parallel kernel. A similar type of serialization that supports serializing collaborative kernels has been previously done in literature [21, 22], however, the target was to serialize multiple GPU threads to a single CPU thread. The key strategy in these approaches is to divide the code to multiple sections, separated by barriers, with loops around each section, where the state of all threads is preserved across all barriers through performing scalar expansion on all local variables. On the GPU, performing such scalar expansion would convert all register accesses to local memory accesses which would become extremely inefficient. Moreover, code that uses barrier synchronization often implements a parallel algorithm that is not efficient when performed serially. For example, when choosing an algorithm that does reduction or scan operations on a GPU, a parallel reduction tree is often used where barrier synchronization operations are needed between levels of the tree. However, these algorithms are only

efficient when done in parallel due to the nature of the algorithm. Thus, when running serially on one thread it is much more efficient to just run a sequential version of the algorithm like a linear reduction. As a result, in such a case, it is better to allow the programmer to choose the more efficient and better performing algorithm and apply thresholding manually since the best sequential and parallel algorithms differ. Moreover, we do not construct serial versions if the child kernel uses shared memory. The reason is that the aggregated shared memory requirements per parent thread would be too high as each parent thread would require what's equal to an entire child thread grid. In addition to that, code that relies on shared memory also often uses barrier synchronization across threads (`__syncthreads()`) operation to coordinate shared memory read and write across the block threads.

3.1.3 Identifying the Number of Child Threads

One of the key steps and dependencies in applying the transformation described in 3.1 is the value of `__threads` which represents the units of parallelism required. The transformation relies on this value to be compared with the threshold, which is a key part in the decision of whether to serialize or launch a parallel kernel. However, being able to detect the count of child threads is not a trivial operation and is often challenging since the number of threads is not explicitly provided by the programmer in code. When configuring a kernel launch, the programmer provides the parallel requirements to the kernel call encapsulated in the grid dimension (total number of blocks) and block dimension (number of thread blocks), more specifically, the number of threads is included in the calculation of the grid dimension. One possible approach to identifying the desired number of threads would be to multiply the grid dimension by the block dimension. However, this

approach would calculate the total number of threads inside the launched grid instead of the usable threads. This value can be misleading and would lead to overestimating the required amount of parallelism. For example, consider the case with a parent kernel that also includes a child kernel launch where the block size dimension is configured to use 1024 threads per block. One of the parent threads discovering dynamic work identifies two units of nested parallel work to be processed by a dynamic launch. This parent will configure the child kernel with exactly one block that includes 1024 threads. In this case, multiplying the grid dimension by the block size would yield the value of 1024 which is much larger than the actual required amount of parallelism (two threads). Ideally, the threshold should be compared to the exact amount of required units of parallelism. As a result, multiplying the grid and block dimension sizes is not a good approach.

The required number of threads is available in the expression that calculates the grid dimension, we rely on an observation that this grid dimension is often calculated by applying a ceiling division operation of the desired number of threads (which represents the exact value of the required amount of parallelism) over the block size dimension. Moreover, our observation is complemented by a set of commonly applied expressions that are commonly used by programmers to calculate grid dimensions. Fig. 3.2 shows the aforementioned set of expressions. Options (a)-(c) uses integer arithmetic to calculate the result. Options (d)-(e) uses floating-point casting operations then calls the `ceil` function which returns the same result. Finally, option (f) that represents multi-dimensional blocks including multiple arguments through the `dim3` constructor that takes three arguments, each one of these arguments is an expression that often resembles one of the options (a) - (e). For all the expression options, the expression may be

(a) `(N - 1)/b + 1`

(b) `(N + b - 1)/b`

(c) `N/b + (N%b == 0)?0:1`

(d) `ceil((float)N/b)`

(e) `ceil(N/(float)b)`

(f) `dim3(..., ..., ...)`
// dim3 args could be one of the above expressions

`N`: desired number of threads

`b`: block dimension

Figure 3.2: Common Expressions for Calculating the Grid Dimension

expressed as whole or it might be calculated through several steps in parts that are stored in intermediate variables. noting that the expressions `N` and `b` can be any arbitrary expressions.

To identify the number of child threads, based on the mentioned discussion, we implement an analysis pass to detect and extract the number of threads from the grid dimension expression. We build the analysis upon an observation made in Fig. 3.2 that the sub-expression containing `N` is usually in the sub-expression on the left-hand side of the division operator. Moreover, the sub-expression `N` might also include other sub-expressions that could be constants such as 1 or `b` which is also usually a constant. Based on this observation, our analysis unwraps the expressions and looks for a division operation, takes the sub-expression for the left-hand-side and removes additions, subtractions of constants only considering the remaining sub-expression as the desired number of threads. This analysis is heuristic by nature and does not guarantee to find the exact number of threads. However, it is acceptable in our context since the result will only be used to decide to either serialize or launch the parallel kernel and as a result, there is

absolutely no impact on correctness in any way. After the analysis detects the number of threads, as shown in Fig. 3.1 we introduce an intermediary variable `__threads` that holds the value of the found sub-expression. Then, the value of `gDim` is replaced with `__threads` to ensure that the expression is not duplicating this code which would impact correctness if the original expression has any side-effects.

3.1.4 Applying the Launch Threshold

The final step in the transformation described in 3.1 requires inserting a guard around the dynamic kernel launch to decide either to launch a dynamic kernel, or serialize the work. We insert an `if` statement around the dynamic launch to ensure that the launch is only performed if the value of `__threads` variable introduced in 3.1.3 is greater than or equal to `__THRESHOLD`. `__THRESHOLD` is a macro that can be overridden at compile time for tuning and usability purposes. If the value `__threads` is less than `__THRESHOLD` then the serial version that has been constructed as described in 3.1.1 is called, thereby the child work would be serialized in the parent thread otherwise a nested grid is launched.

3.2 Coarsening Transformation

The coarsening optimization transforms the code whereby the work of multiple thread blocks in the original code is assigned to a single thread block that executes the thread blocks serially one after the other. This transformation includes two key parts: constructing the coarsened child kernel that iterates over multiple thread blocks, and modifying the launch configuration to launch the coarsened child kernel with the updated grid configuration.

Fig. 3.3 shows an example of how the coarsening transformation is applied across both parent and child kernel. The original code before the transformation in Fig. 3.3(a) showing two key elements: the parent kernel (lines 04-08) and child kernel (lines 01-03). Fig. 3.3(b) shows the result of the coarsening transformation that also consists of two key parts: a modified parent kernel (lines 14-20), and a modified child kernel (lines 09-13). In the parent kernel, we modify the grid dimension (lines 16-18) `gDim` from the original dynamic launch configuration and update it to the coarsened grid dimension using `__CFACTOR` which is a macro configured at compile time. Also, we update the child kernel creating a new version using a loop that iterates over the original grid dimension. In this example, we show coarsening in one dimension only for simplicity.

3.2.1 Constructing the Coarsened Kernel

The transformation described in 3.2 requires a modified kernel such that every thread block executes the work of multiple thread blocks. To construct such kernel, we start by appending a parameter `_gDim` to the parameters list as shown in Fig (b) (line 09). This parameter represents the original grid dimension without coarsening. Then, we append a loop that iterates over child thread blocks assigned to the new coarse block. We also replace the uses of all reserved indices such as `blockIdx.x` and `gridDim.x` with the corresponding loop indices and bounds. For simplicity, the above example only shows transformation with a single grid dimension, if the child grid is multidimensional, loops would be inserted for each dimension and reserved indices would also be replaced as such. The way loops are inserted and the calculation of thread block indices can be done in different strategies, we discuss three different strategies in section 3.2.2

```
01  __global__ child(...) {
02      child body
03  }

04  __global__ parent(...) {
05      ...
06      child <<< gDim, bDim >>> (...);
07      ...
08  }
```

(a) Original Code

```
09  __global__ child(..., _gDim) {
10    for(_bx = blockIdx.x; _bx < _gDim.x; _bx += gridDim.x) {
11      child body  // Replace uses of blockIdx.x with _bx
12    }             // and gridDim with _gDim
13  }

14  __global__ parent(...) {
15      ...
16      _cgDim = _gDim = gDim ;
17      _cgDim.x = (_gDim.x + _CFACTOR – 1)/_CFACTOR;
18      child <<< _cgDim, bDim >>>(..., _gDim);
19      ...
20  }
```

(b) Code after Child Coarsening

Figure 3.3: Coarsening

3.2.2 Coarsening Strategies

A coarsened kernel can be expressed in multiple different ways. Fig. 3.4 shows three different ways that we use to express coarsening in the child kernel.

Fig. 3.4(a) shows the coarsening strategy (1) that uses a loop iterating over the child thread blocks assigned to the coarse block, the iterator is bounded by the original grid dimension and incremented using the coarsened grid dimension. For example, given a grid that originally consists of 6 zero-indexed blocks (blocks 0-5), after applying coarsening using `__CFACTOR` = 2 we end up with exactly 3 zero-indexed coarsened blocks (blocks 0-2), this strategy would assign the extra work of block 3 to 0, the work of block 4 to 1, and block 5 to block 2 as shown in Fig. 3.5(a).

Fig. 3.4(b) shows the coarsening strategy (2) that uses a loop iterating over the child thread blocks assigned to the coarse block, the iterator is bounded by the original grid dimension and incremented using an offset of 1. For example, given a grid that originally consists of 6 zero-indexed blocks (blocks 0-5), after applying coarsening using `__CFACTOR` = 2 we end up with exactly 3 zero-indexed coarsened blocks (blocks 0-2), this strategy would manipulate the indices whereas every thread block would execute the work based on its index plus a certain offset. In this particular example, block 0 is assigned the work of (0-1), block 1 is assigned the work of (2-3), and block 2 is assigned the work of (4-5).

Fig. 3.4(c) shows the coarsening strategy (3) that uses a loop iterating over `_CI`, the iterator is bounded by the coarsening factor represented by `_CI` and incremented using an offset of 1, then the new block index is calculated as a factor of the coarsened block index, the coarse factor, and the iterator. For example, given a grid that originally consists of 6 zero-indexed blocks (blocks 0-5), after coarsening using `__CFACTOR` = 2 we end up with exactly 3 zero-indexed

coarsened blocks (blocks 0-2), this strategy works the same way with reindexing as a strategy (2) as shown in Fig. 3.5(c). The difference between strategy (2) and (3) is that the latter might allow the compiler to apply loop unrolling optimization since the iterator factors are always constants.

As such the main difference between strategy (1) and strategies (2-3) is how the work is assigned across the launched thread blocks which could potentially result in a difference in performance due to the possible locality influence. Applying the desired coarsening strategy in our compiler is configurable with a flag on compile time.

```
09  __global__ child(..., _gDim) {
10    for(_bx = blockIdx.x; _bx < _gDim.x; _bx += gridDim.x) {
11      child body  // Replace uses of blockIdx.x with _bx
12    }             // and gridDim with _gDim
13  }
```

(a) Strategy 1

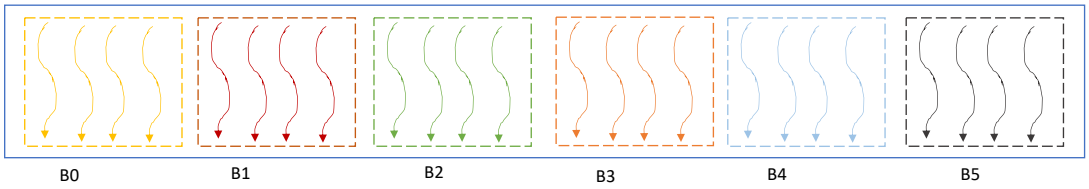
```
09  __global__ child(..., _gDim) {
10    for(_bx= blockIdx.x * _CF; _bx< min((blockIdx.x + 1) * _CF, _gDim.x);++_bx) {
11      child body  // Replace uses of blockIdx.x with _bx
12    }             // and gridDim with _gDim
13  }
```

(b) Strategy 2

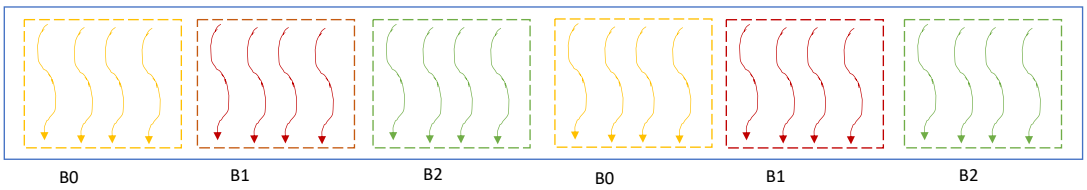
```
09  __global__ child(..., _gDim) {
10  for (_CI = 0; _CI < _CF; ++_CI) {
11      _bx = blockIdx.x * _CF + _CI;
12      if(_bx < gDim.x) {
11          child body  // Replace uses of blockIdx.x with _bx
12    }             // and gridDim with _gDim
13  }
```

(c) Strategy 3

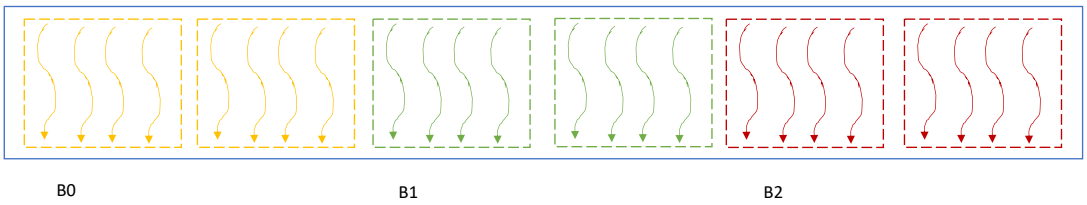
Figure 3.4: Coarsening Strategies



(a) Original Grid



(b) Strategy 1: assigns work using the new gridDim as factor to process original block threads



(c) Strategy 2 & 3: assigns work using coarsening factor offset to process neighbouring blocks

Figure 3.5: Coarsening Strategies Processing

3.2.3 Modifying the Grid Dimension

The transformation described in 3.2 requires the original parent kernel to launch a modified coarsened child kernel as shown in Fig. 3.2.1(line 18). To modify the launch configuration of the dynamic child launch, first, we store the original grid dimension `gDim` in a variable `_gDim` line(16). The value is then also copied to `_cgDim` which would represent the new coarsen grid dimension. The x-dimension of the coarsened grid dimension `_cgDim` is then calculated as the ceiling division by `_CFACTOR`(line 17) which represents the coarsening factor. `_CFACTOR` is a macro that can be configured at compile time for tuning and usability purposes.

3.3 Overall Compiler Flow

As discussed in section. 3 the optimizations are implemented independently and can be applied in different orders. We apply the optimizations in the following order: thresholding, coarsening, then aggregation as shown in 3.6. We start by applying thresholding because coarsening updates the original grid dimension as discussed in section 3.2.3 which makes it harder to identify the number of threads that need to be compared with the threshold as detailed in section 3.1.4. Thresholding is also applied before aggregation to isolate small grids and serialize them before they are aggregated and set to be launched dynamically since that the logic of aggregation makes it much more difficult to detect those small grids. Coarsening is also applied before aggregation because the aggregation logic should be outside the coarsening loop so that the cost of disaggregation is amortized across the coarsen child block threads.

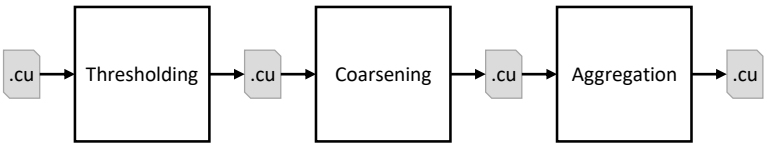


Figure 3.6: Overall Compiler Flow

For thresholding and coarsening we transform the code using our implementation as detailed in 3.1 and 3.2. As for aggregation, we leverage a prior compiler-based implementation [6] in our compiler flow.

Chapter 4

Methodology

4.1 Implementation and Setup

The thresholding and coarsening compiler passes are implemented as source-to-source transformation pass in Clang version 7.1.0 [23]. For the aggregation transformation component, we leverage prior works [6]. We evaluate our work on a system with the following specifications:

- CPU: AMD EPYC 7551P
- Main Memory: 15GB
- GPU: Nvidia's Volta V100 with 32GB of device memory

4.2 Benchmarks

The benchmarks used are shown in table 4.1. For each benchmark, we use two base implementations `CDP` and `NoCDP`. The `NoCDP` version is a manual implementation that does not use `CUDA` dynamic parallelism feature. The `CDP` version is

a manual implementation that uses dynamic parallelism. All other versions are generated through the compiler using the `CDP` version as a base. `T` is for thresholding, `C` for coarsening, and `A` for aggregation. To evaluate, we apply all the optimizations separately or combined on each benchmark. For example, `CDP + T + C + A` would mean that thresholding, coarsening, and aggregation are applied through the compiler to this benchmark in the same respective order as discussed in 3.3. The only exception for this is with Triangle Counting (`TC`) benchmark because the thresholding is manually applied in the original code implementation. Each of these benchmarks also includes a testsuite that asserts the correctness of the benchmark result, we leverage that to verify the correctness of our generated code, we also verify the results manually. For our combined optimizations evaluation, we always use coarsening strategy (1) described in 3.4, when we tune the threshold, we always report speedups where there is at least one nested launch unless explicitly stated otherwise. To avoid overflowing the launch buffer pool, all the benchmarks are configured with appropriate launch count. Also, when compiled, we use per-thread default streams to ensure that the launches from the same block are not bottlenecked on the same default stream.

Table 4.1: Benchmarks

Name	**Description**	**Dataset**
BFS	Breadth First Search [24]	KRON, CNR
BT	Bezier Tessellation [25]	T0032-C16, T2048-C64
MSTF	Minimum Spanning Tree (find kernel) [26]	KRON, CNR
MSTV	Minimum Spanning Tree (verify kernel) [26]	KRON, CNR
SP	Survey Propagation [26]	RAND-3, 5-SAT
SSSP	Single Source Shortest Path [26]	KRON, CNR
TC	Triangle Counting [27]	KRON, CNR

4.3 Datasets

The datasets used are shown in table 4.2. We use larger datasets than prior works because we evaluate on a larger GPU. For all the graph application benchmarks, namely, `BFS`, `MSTF`, `MSTV`, and `SSSP` [24, 26, 27] we use the same set of graphs KRON, and CNR [28, 29] in full. The only exception is TC where we only use part of the two mentioned graphs (570K edges) for memory constraints. As for `BT` we test on two datasets, one with the default value from the source code implementation and the other where we synthetically increase the curvature and max tessellation points [25]. Finally, for SP we use `5-SAT` and `RAND-3` datasets. [30, 26].

Table 4.2: Datasets

Name	**Description**
KRON	kron_g500-simple-logn16, 65,536 vertices, 2,456,071 edges [29]
CNR	cnr-2000, 325,557 vertices, 2,738,969 edges [28]
T0032-C16	Max Tessellation 32, Curvature: 16, Lines: 20,000 [25]
T2048-C64	Max Tessellation: 2048, Curvature: 64, Lines: 20,000 [25]
RAND-3	random-42000-10000-3, 10,000 literals [26]
5-SAT	5-SATISFIABLE, 117,296 literals [30]

Chapter 5

Evaluation

5.1 Performance

We apply all the optimizations and their combinations on the benchmarks and present the results in form of performance speedup over the manual CDP and NoCDP implementations. When directly comparing the CDP and NoCDP versions it is clear that CDP almost always performs significantly worse. Fig. 5.1 shows the overall speedup when each optimization is applied by its own. First, we start by only applying thresholding showing that it alone provides a substantial speedup of 13.4x (geomean) over CDP. Second, as for coarsening, by itself, it only provides a very modest speedup of 1.01x over CDP. Finally, we apply aggregation by itself providing a substantial speedup of 12.1x than CDP. In our compiler flow, when applied separately, aggregation and thresholding are the main techniques that directly targets the launch overhead, whereas coarsening is designed to work in synergy with aggregation as shown later in section 5.4. In accordance with previous works, the major slowdown in CDP vs NoCDP is caused by the sheer number of launches that overwhelms the device by causing

a massive overhead [12, 13, 6, 14], hence when targeted directly, both aggregation and thresholding recovers from the performance degradation and achieves substantial performance increase over both CDP and NoCDP.

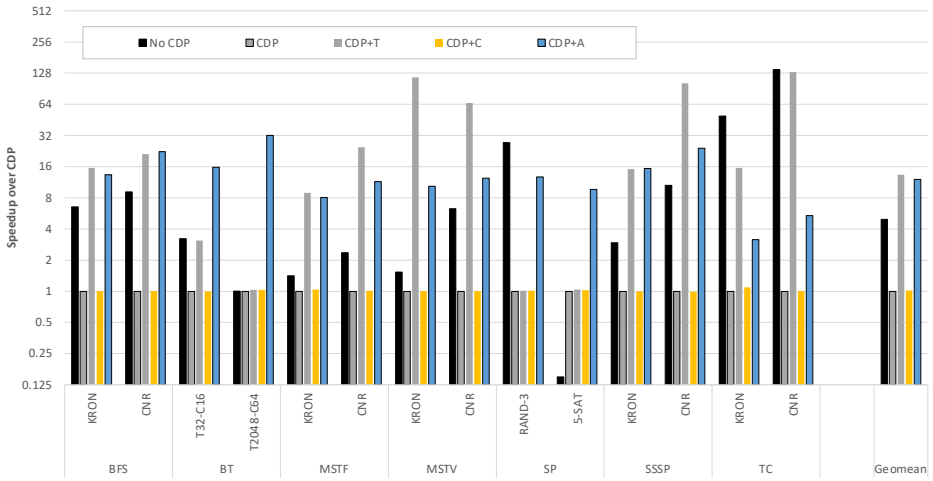


Figure 5.1: Individual Optimizations Speedup

5.1.1 Thresholding

Applying thresholding alone provides a substantial speedup of 13.4x (geomean) over CDP. Being an optimization that directly targets the launch overhead by reducing the number of launches and only allowing launches where the benefit is not cancelled by the launch overhead.

We apply thresholding alone and tune the threshold where there is at least one dynamic launch, except for the last data point where we experimentally

tune the threshold beyond the largest launch, however, this data point is not considered as part of the eventual speedup and only considered in section 5.5. Fig 5.6 shows the design space achieved when thresholding is applied. The first observation is that for most benchmarks, incrementally increasing the threshold improves performance which makes sense since that it lowers the launch overhead. The second observation is that for most benchmarks, setting too high of a threshold decreases performance, the reason behind that is when the threshold is too large the dynamic launches are serialized in their parent threads reducing parallelism. There are few inconsistencies in the behaviour of benchmarks after applying thresholding, notably `BT` (T2048-C64 dataset) and `TC`, as for `BT` the different increments in thresholding is yielding different values, the reason behind this is that the implementation highly depends on dynamic memory allocation in the child kernel as (broken down further in section 5.2), thus the huge memory calls largely affects the result. As for `TC`, incremental threshold steps does not make a substantial difference until the threshold reaches the highest tune-able values, the reason behind this lies in the difference between `TC` and the other algorithms. BFS, SSSP, MSTF, and MSTV are iterative algorithms that operate on a subset of the edges at a time, per iteration, whereas for `TC` the used implementation uses all the processable edges of the graph at the same time, for that reason, the device is extremely overwhelmed with launches even at relatively very high threshold values.

Fig. 5.2 shows that when aggregation is applied after thresholding in CDP + T + A version, thresholding still gives a speedup where it performs significantly better at 2.9x than when aggregation is applied alone (geomean). Although aggregation and thresholding tackle the same launch overhead issue, aggregation could include underutilized thread blocks whereas thresholding serializes them.

This behaviour is further discussed in 5.3.

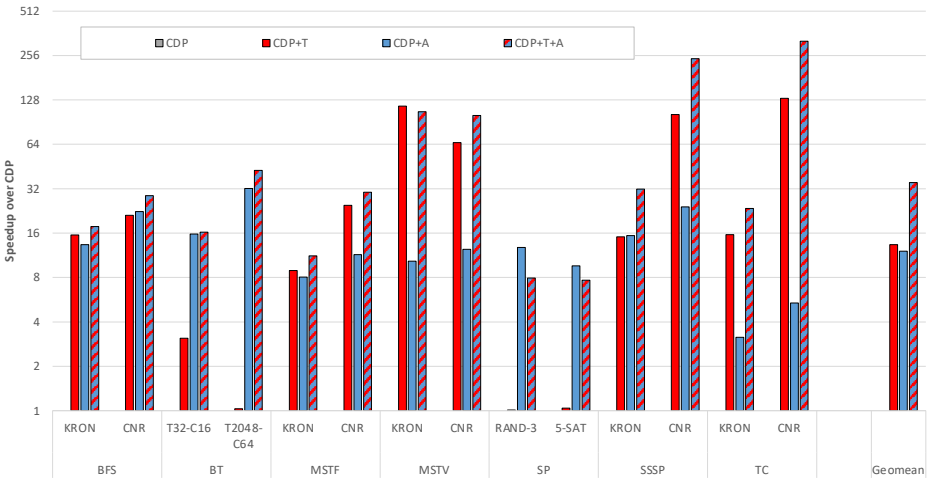


Figure 5.2: Thresholding Optimization Speedup

5.1.2 Coarsening

Fig. 5.1 shows that applying coarsening alone provides a very modest speedup of 1.01x (geomean) over CDP. This makes sense for the reason is that in the context of dynamic parallelism, and our benchmarks, by itself, coarsening does not tackle any specific bottleneck or tries to minimize any specific overhead.

However, Fig. 5.3 shows that coarsening is synergistic with both aggregation and thresholding. When coarsening is applied on top on thresholding, in the CDP + T + C version it provides 1.09x speedup over just CDP + T. When also combined with aggregation, CDP + C + A provides 1.16x speedup over just applying CDP + A. Although these speedups are less apparent than CDP + A

and CDP + T over the base CDP version, they are still significant. as discussed in 5.4

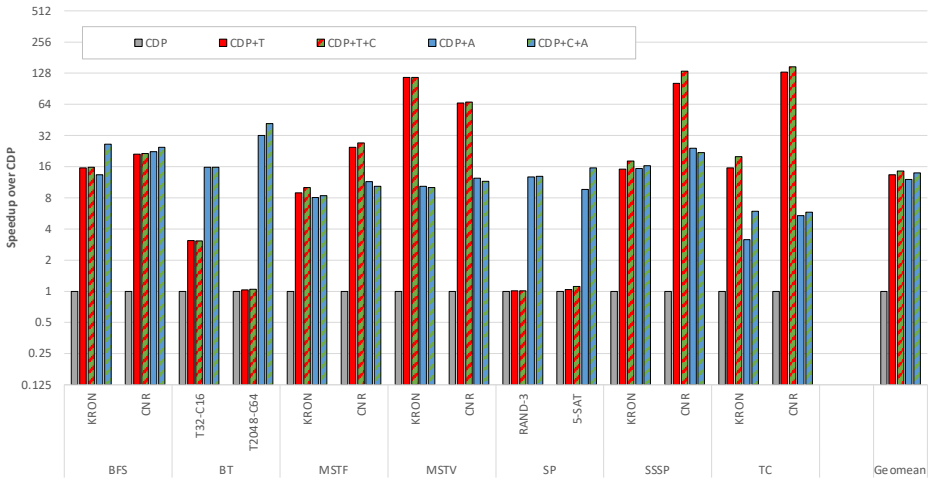


Figure 5.3: Coarsening Synergistic Speedup

5.1.3 Aggregation

As shown in Fig. 5.1 Applying aggregation alone provides a significant speedup of 12.1x faster than CDP (geomean). This speedup makes sense since that aggregation directly tackles the massive overhead caused by the massive number of launches, this observation is consistent with evaluations made in prior works [12, 13, 6, 14]. Moreover, the speedup achieved by aggregation alone overtakes the performance degradation from the launch overhead being 2.4X faster than the NoCDP implementation.

5.1.4 Compiler Framework

As discussed in section 3.3 we choose our implementations based not only on their separate performance enhancements but also on their synergistic potential. We apply our 3 optimizations in the following order: thresholding, coarsening, then aggregation. With the three optimizations applied in CDP + T + C + A this version provides a speedup of 1.22x (geomean) over CDP + T + A, and 3.1x (geomean) over CDP + C + A. In total, when compared to the base CDP and NoCDP versions, Fig. 5.4 shows that a major speedup of 43.0x (geomean) is achieved over CDP and a significant speedup of 8.7x (geomean) over NoCDP version.

As a result, being able to understand the bottlenecks of dynamic parallelism then applying the right optimizations that tackles them showcases that dynamic parallelism is a powerful programming feature that can provide a significant speedup.

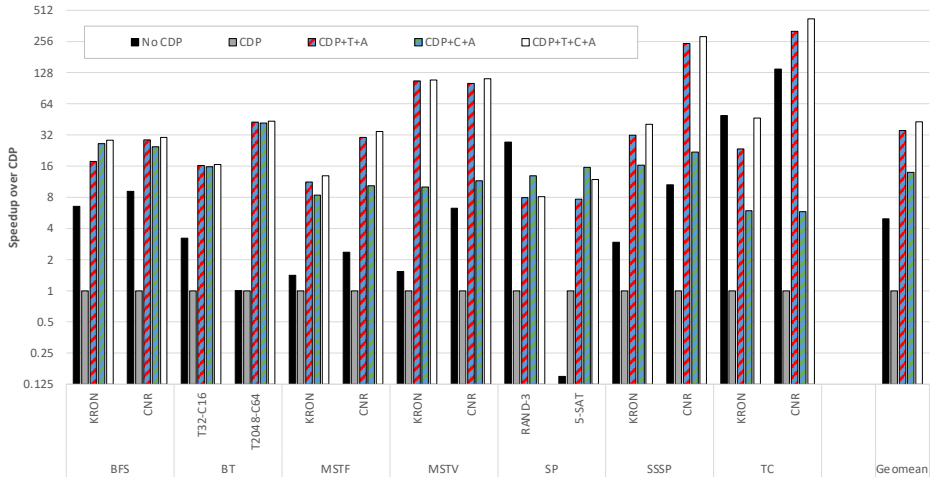


Figure 5.4: Overall Compiler Framework Speedup

5.2 Breakdown of Execution Time

To better understand the effects of each optimization, and the incremental combinations we profile the run-times to obtain the breakdown of execution time shown in Fig. 5.5. To obtain this data, we start with the base unaltered code then incrementally deactivate parts of the same code, execute and measure the execution time difference. When deactivating the code, we use guards around the code with conditionals that are always false so that the code is never executed, we also eliminate the possibility of dead-code-elimination by using conditionals that cannot be detected to be false at compile time.

In order to understand the interaction between aggregation alone and the

combination of the other optimizations, we do not include the CDP version and use CDP + A as a baseline. However, prior work [6] already shows a thorough comparison between CDP + A and CDP.

We start our observations by looking at Fig. 5.5 from left to right. We start by comparing CDP + A and CDP + T. The first and most notable observation is that when thresholding is applied, the child work greatly decreases and parent work greatly increases, this result is predictable since thresholding would cancel many launches from the parent that would have been child work and this same work is shifted to the parent when the work is serialized. Second, the launch overhead is also significantly reduced, this result is not surprising because thresholding reduces the number of launches overall, and by cancelling the small launches it eliminates a significant part of the launch overhead. Finally, thresholding decreases both the aggregation and disaggregation logic, through reducing the number of the launches, fewer parent threads would engage in proceeding to do the aggregation work and the disaggregation logic would also be reduced since fewer child threads would be searching for their parents.

We proceed by adding coarsening to our analysis and compare CDP + T + C + A to CDP + T + A. Our first observation is that coarsening contributes to the overall speedup by decreasing the launch overhead, this is achieved by having a reduced number of thread blocks that would need to be scheduled on the GPU. Second, coarsening decreases the disaggregation logic overhead. This is because the total number of scheduled aggregated child block threads is less overall and as a result, the cost of the disaggregation logic that's usually done across multiple thread blocks would be amortized and done only on the relevant coarsened blocks.

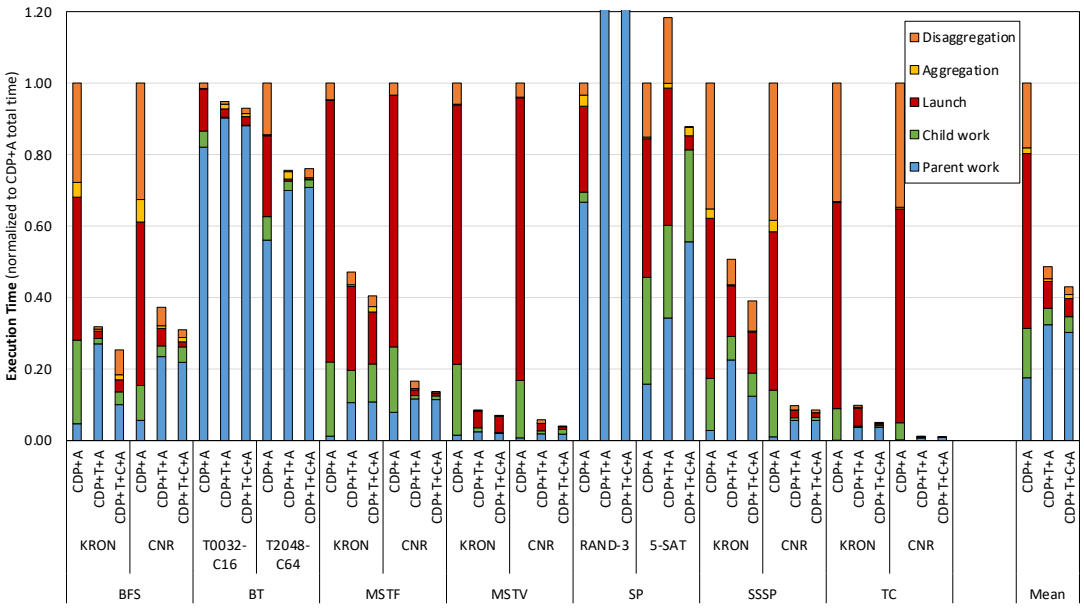


Figure 5.5: Breakdown of Execution time

5.3 Interaction between Thresholding and Aggregation

The overall interaction between thresholding and aggregation in our complete framework is shown in Fig. 5.6. We fix the best coarsening factor for each benchmark, for each data set and observe the differences in behaviour when applying different thresholds across different aggregation granularities. The first observation is that for most datapoints, when combined, higher aggregation granularity always yields the best results even in the presence of thresholding, this is consistent with prior work where the higher granularity performed better [6, 14]. The second observation is that lower granularity level favours higher threshold as the optimal number. The reason behind this is when the granularity level is lower, there would be a higher number of launches overall, as a result, thresholding

would further decrease this number of launches lowering the launch overhead. Third, applying too high of threshold decreases performance as it varies from the optimal value, the reason behind this is that higher threshold would move the work to the parent to be serialized instead of having it aggregated, this effect is also consistent when the threshold was applied on its own 5.1.1.

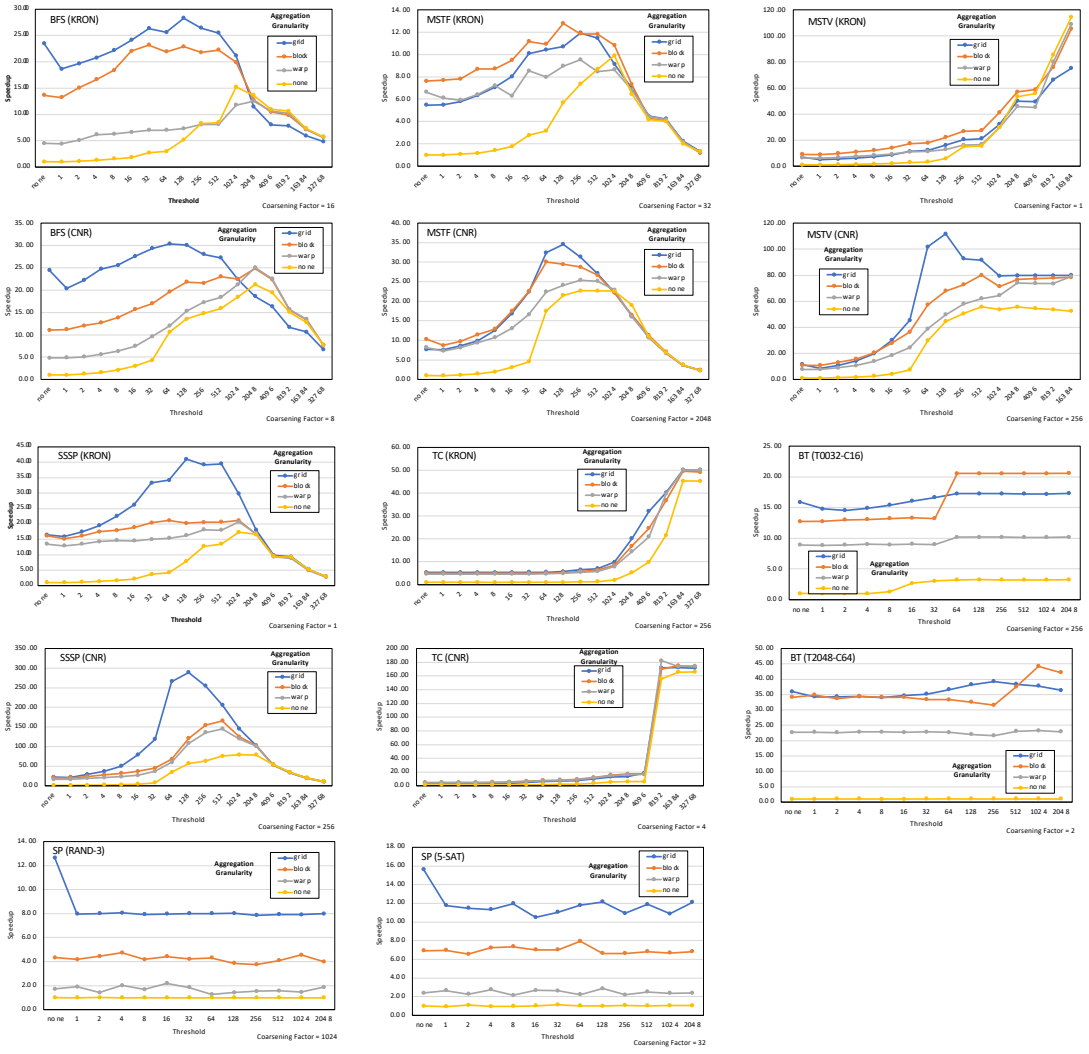


Figure 5.6: Design Space Exploration of Thresholding and Aggregation

5.4 Interaction between Coarsening and Aggregation

Using the same methodology in section 5.3 we fix the best threshold and observe the differences as shown in Fig. 5.7. The first observation, is that coarsening also does not affect the choice of the best granularity which remains to be the highest across almost all benchmarks. The second observation, adding coarsening on top of aggregation increases performance, as we increase the coarsening factor for the first few steps the performance increases. However, adding coarsening by itself does not improve performance notably as discussed in section 5.1.2. The third observation is that if the applied coarsening factor is too high, then it hurts performance. The second and third observation makes sense together, since that there is a sweet spot where the coarsening factor decreases the number of scheduled blocks but also doesn't majorly affect the parallelism as noted in section 5.2. Finally, we note that the highest granularity benefits the most from applying the coarsening optimization, which also makes sense since that the goal from applying coarsening is to amortize the disaggregation logic which is higher when using higher granularities because the search operation for every thread to find its parent would be more expensive, this note is also consistent with prior works where the breakdown between different aggregation granularities is also shown [6].

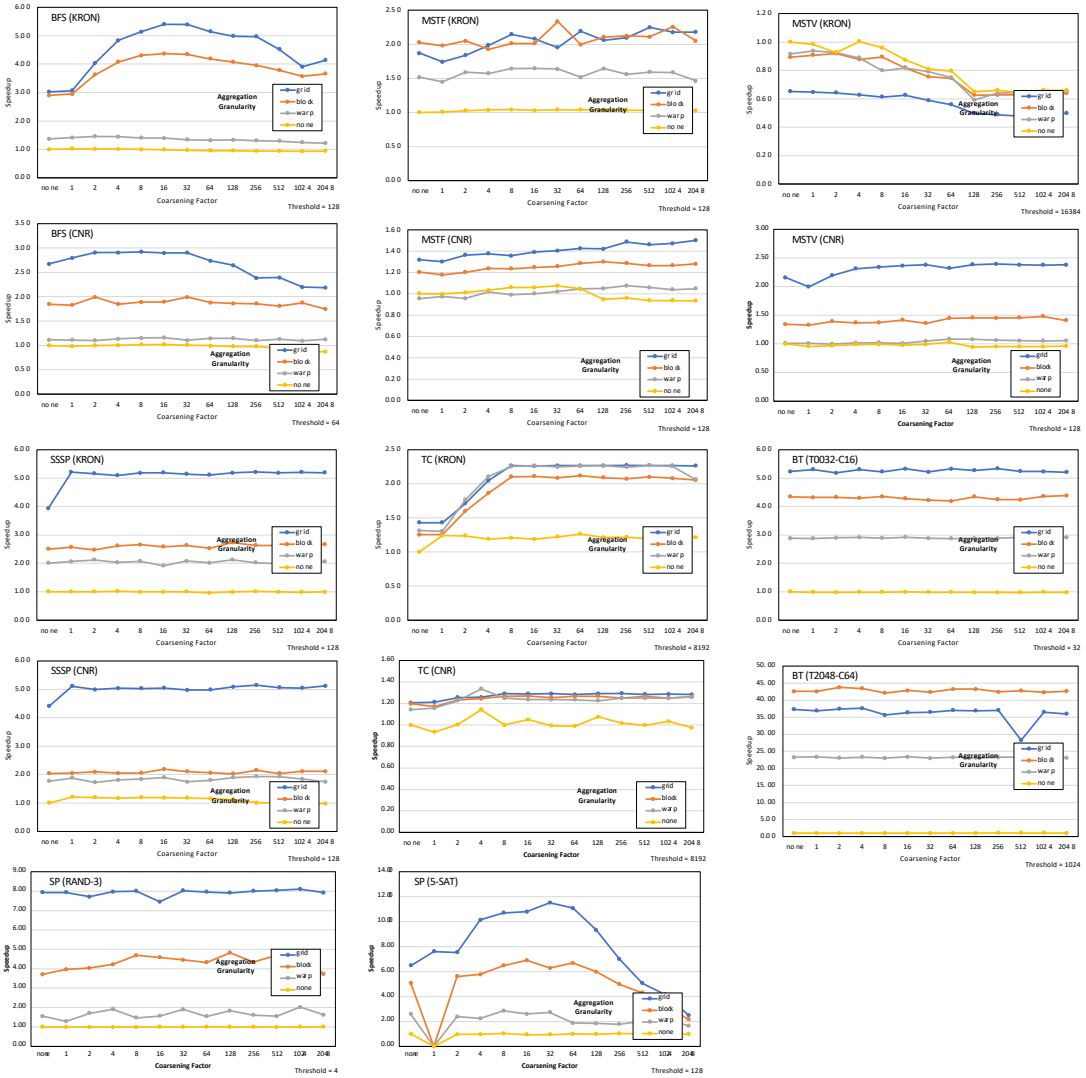


Figure 5.7: Design Space Exploration of Coarsening and Aggregation

5.4.1 Effects of Applying Different Coarsening Strategies

We implement the coarsening strategies in our compiler as discussed in section 3.2.2. We use CDP + T + C + A version because it is the most performing version, We fix the best threshold value for each benchmark, for each dataset and always use grid granularity. Fig. 5.8 shows the performance difference between coarsening strategies (1-3) using CDP + T + C + A strategy (1) as baseline. The

first observation is that performance slightly diverges as we incrementally increase the coarsening factor. The second observation is that strategy (1-2) performs almost the same on most points whereas strategy (3) performance significantly diverges to the lower end of the spectrum. The reason behind coarsening strategy (3) performing worse on high factors is because when the coarsening factor is tuned beyond the total number of blocks, due to the loop not being bounded, it would still try to execute all iterations as shown in Fig. 3.4 line(10).

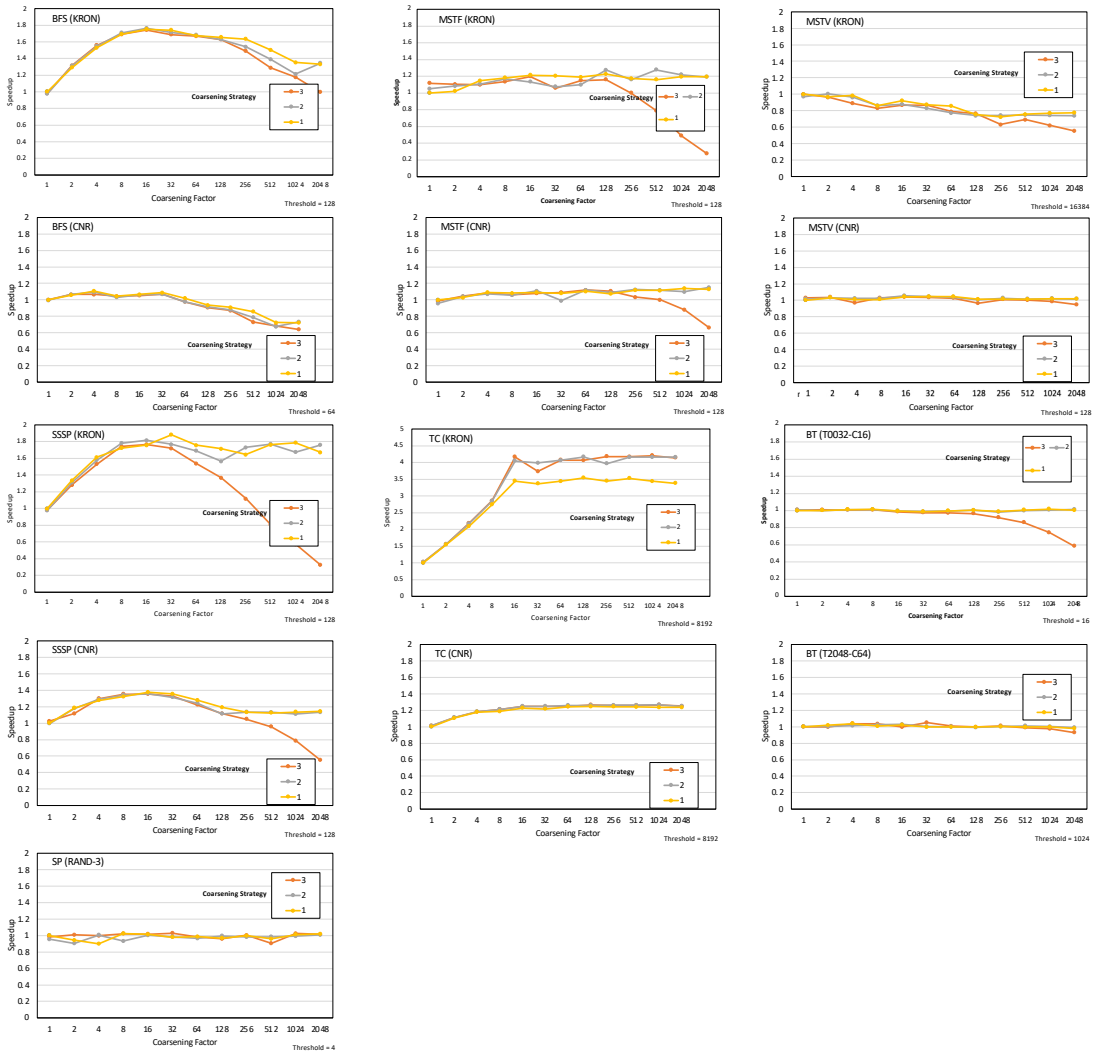


Figure 5.8: Comparing Different Coarsening Strategies

5.5 Unsuitable Cases

Dynamic Parallelism is a powerful tool when applied with the right optimizations. Most notably, when the application's parallelism requirements are high. All the aforementioned optimizations minimize the launch overhead, thresholding cancelling the small launches and serializing the work, and aggregation by consolidating small launches into a single grid. However, there are cases where the parallel requirements are too low in the application itself where even if the launches are consolidated, the launch overhead still overtake the benefit of thresholding. In particular, when applying thresholding, it would serialize all the work turning CDP to NoCDP, but it will also still incur the thresholding logic overhead. To showcase and demonstrate an example of such a case, we evaluate the graph benchmarks on a road graph (USA-road-d.NY [31]). The graph has 264,346 vertices, 730,100 edges, an average degree of 3, and a maximum degree of 8.

We follow the same methodology applied in section 5.1 by applying all our set of optimizations. Fig. 5.9 shows that the CDP version performs significantly worse than its NoCDP counterpart, and this is because that in the given graph, the degree of each vertex is too low on average, thus having very low nested parallelism requirements. Another observation is that even when all the optimizations are applied, NoCDP version still performs significantly faster for the same reason, aggregation does not find enough child threads to justify the launch and increasing the threshold any further would disallow the code from allowing any nested launch, as a matter of fact, for this particular experiment we tune the threshold beyond the maximum number of the highest launch and serialize all child work, however, the incurred overhead of applying all the optimizations in

code does not allow the performance to fully recover.

The SP benchmark in Fig. 5.4 exhibits the same behaviour on the RAND-3 data-set where NoCDP version performs significantly better than all the optimizations, that is for the same reason where the largest grid required to be launched includes only 32 threads.

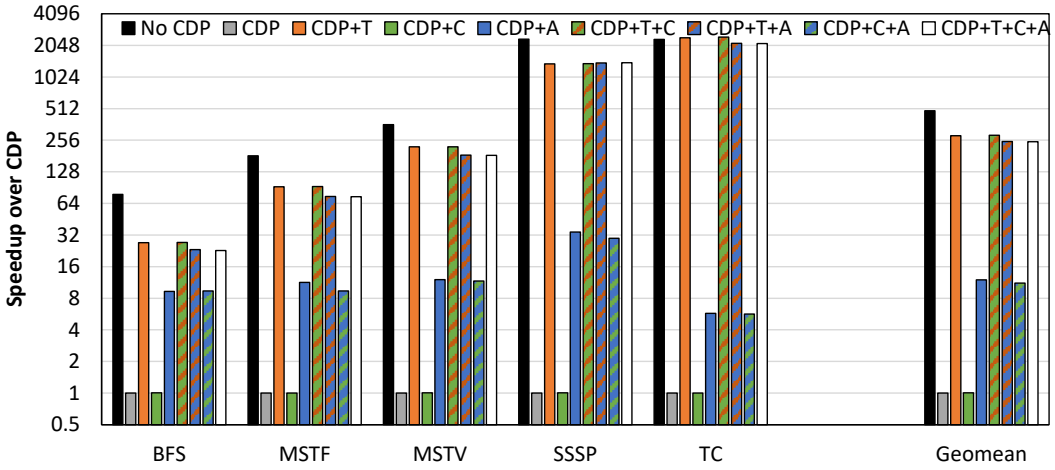


Figure 5.9: Optimizations Applied on Unsuitale Cases

Chapter 6

Related Work

The dynamic parallelism concept and implementation provided scientists with a powerful tool that would simplify their implementations and potentially make it achieve higher parallelism. However, several benchmarks had shown inefficiencies in practice caused by hardware utilization and launch overhead [3, 32, 33], which motivated multiple optimizations. In this section, we discuss typical use cases of dynamic parallelism and multiple approaches that were used to optimize dynamic parallelism through either hardware or software.

6.1 Applications using Dynamic Parallelism

In literature, there are three main categories of applications that use dynamic parallelism. The first category is applications where dynamic parallelism is used to *transfer control* from the CPU to the GPU [34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46]. There are many applications that relies on iteratively launching consecutive grids, this is usually done on a CPU thread handling the launches. This first category lets the GPU control these launches instead. This minimizes

the reliance on the CPU and reduces host-device communication while also freeing the CPU to do other work. This type of applications do not leverage nested parallelism and is not targeted by our optimizations.

The second category is applications with *regular nested parallelism*, this type of applications performs a specific level of nested parallelism where the amount of parallelism at every level required is also known before execution [47, 48, 49, 50, 51]. The usage of Dynamic parallelism may provide multiple advantages over just having the CPU iteratively launch a grid for every level of nesting such as providing a simpler interface to the programmer, launch work for multiple levels instead of waiting for the tail of one level to complete its work, and finally by transferring the control from the CPU to the GPU.

The third category is applications with *irregular nested parallelism*, the main distinction of this category is that both or one of the depth of nested parallelism and amount of parallelism required per level is unknown before execution. The use of dynamic parallelism provides a simpler interface that alleviates the complexity of programming this type of applications with only CPU launches as it would require processing techniques to either serialize all the work of the next nesting level or use complex scheduling techniques to collect work and distribute it across grid launches. Moreover, it potentially provides better performances if the right optimizations are applied. There are many applications where the amount of parallelism is unknown, most notably graph applications such as breadth-first search [52, 53], Minimum Spanning Tree Find and Verify [26], single-source shortest path [52, 53], and depth-first search [54], where the amount of nested parallelism depends on the graph size.

6.2 Hardware Optimizations

Various hardware approaches to optimize dynamic parallelism and mitigate its overhead have been developed and proposed. SPAWN [7] is a hardware controller that advises programmers if a dynamic launch is profitable or not. LASER [10] improves the locality aware scheduling enhancing dynamic parallelism. Dynamic Thread Block Launch (DTBL) [8, 9] proposes hardware support for dynamic lightweight launches of thread blocks that are added to the currently launched grids on the fly instead of launching new entire grids. LaPerm [11] extends DTBL work by adding a locality aware scheduler. The main disadvantage of these approaches is that they all have hardware requirements that are not satisfied by current GPUs motivating the need for software-based optimizations. Moreover, our proposed software optimizations would be synergistic with the hardware optimizations.

6.3 Software Optimizations

Many software approaches have been proposed to optimize dynamic parallelism. CUDA-NP [4] is a compiler approach that transforms the source code where instead of using the dynamic parallelism API, the code is partitioned into sequential and parallel sections, a large number of threads is launched upfront then these threads are segmented into two categories, master and slave threads, then using control flow and annotations, the master threads are activated when encountering sequential sections whereas slave threads would execute the parallel sections.

Free Launch [5], is another software approach that transforms the code whereby it eliminates child grid launches entirely and instead of using dynamic parallelism, the host starts by launching the maximum number of allocatable parent thread

blocks based on the device capabilities and takes a scheduling approach whereby it reuses parent threads/blocks and has them execute the child work in a load-balanced way, either sequentially or in parallel.

Both of these approaches mitigate the overhead of dynamic parallelism but they also avoid it entirely. Moreover, they require threads to be held on standby regardless of the availability of work. Li et al. [12, 13], Wu et al. [14], and KLAP [6] aggregates the work of multiple child kernels and combines them into a single aggregated grid, hence reducing the launch overhead. We leverage [6] and use it as the aggregation component in our workflow.

Chapter 7

Conclusion

7.1 Summary

In this thesis, we presented a compiler framework that includes three separately implemented key optimizations: thresholding, coarsening, and aggregation that optimizes the performance of applications that uses dynamic parallelism. First, Thresholding only allows large enough grids to be launched otherwise it serializes the work thereby the potential benefit of the launch is higher and the reduced number of launches lowers the total launch overhead. Second, coarsening assigns the work of multiple thread blocks to a single thread block thereby when combined with threshold, it reduces the schedulable blocks which could remove cases where the device is oversubscribed, or when combined with aggregation it amortizes the cost of the disaggregation logic. Third, aggregation consolidates multiple grids into a single grid thereby reducing the launch overhead and allowing for better device utilization.

We showcased the compiler implementations, then analyzed and discussed the advantages and use cases of each of these optimizations when applied separately,

or combined. Then we thoroughly evaluated the interactions between the different optimizations. Our evaluation showcases that our compiler framework provides substantial speed-ups over applications that use nested parallelism when the data sets exhibits high irregularity.

7.2 Future Work

Based on the results we observed in our evaluation and tuning, we believe that there is more to these optimizations and we can still fine-tune the performance to achieve even better results. First, we would like to expand on the aggregation granularities. As shown in section 5.3, there is a direct relationship between the optimal threshold and optimal granularity, as such, it would be interesting to explore dynamically configurable granularities where we can fine-tune even further and observe the interaction. Second, we would like to expand the thresholding optimization and apply it on aggregation, whereas we would only allow a launch to be included in the aggregation based on a certain threshold, this would allow us to further understand the interaction between the two techniques. Third, showcased in the coarsening strategies section 3.4 we feel that further analysis should be done to be able fine-tune how these optimizations are applied and look for opportunities. Fourth, as shown in our evaluation 5.3 and 5.4, different coarsening factors and thresholds across different benchmarks and datasets can yield different results, while we've explored the space and logged performance at several points exhausting the space, future work could allow these parameters to be auto-tuned through more sophisticated methods such as leveraging a machine learning model to minimize the space when searching for the best values.

Appendix A

Code Transformations

Listing A.1: BFS CDP Code

```
__global__ void BFS_child(unsigned int *levels, unsigned int
    *edgeArrayAux, int curr, int nbr_off, int num_nbr, int *flag) {
    int gid = blockDim.x*blockIdx.x + threadIdx.x;
    if(gid < num_nbr){
        int v = edgeArrayAux[gid + nbr_off];
        if(levels[v] == UINT_MAX) {
            levels[v] = curr + 1;
            *flag = 1;
        }
    }
}

__global__ void BFS_parent(
        unsigned int *levels,
        unsigned int *edgeArray,
        unsigned int *edgeArrayAux,
```

```
        unsigned int numVertices,
        int curr,
        int *flag) {
    int gid = blockDim.x * blockIdx.x + threadIdx.x;
    if (gid < numVertices) {
        if (levels[gid] == curr) {
            unsigned int nbr_off = edgeArray[gid];
            unsigned int num_nbr = edgeArray[gid + 1] - nbr_off;
            BFS_child<<< (num_nbr - 1) / CHILD_BLOCK_SIZE + 1,
                CHILD_BLOCK_SIZE>>>(levels, edgeArrayAux, curr, nbr_off,
                num_nbr, flag);
        }
    }
}
```

Listing A.2: BFS CDP + T

```
__device__ void
BFS_child__serial(
        unsigned int *levels,
        unsigned int *edgeArrayAux,
        int curr,
        int nbr_off,
        int num_nbr,
        int *flag,
        unsigned int _bDim,
        unsigned int _gDim
) {
```

```
    for (unsigned _bx = 0; _bx < _gDim; ++_bx) {
        for (unsigned int _tx = 0; _tx < _bDim; ++_tx) {
            int gid = _bDim * _bx + _tx;
            if (gid < num_nbr) {
                int v = edgeArrayAux[gid + nbr_off];
                if (levels[v] == (2147483647 * 2U + 1U)) {
                    levels[v] = curr + 1;
                    *flag = 1;
                }
            }
        }
    }
}

__global__ void BFS_parent(
        unsigned int *levels,
        unsigned int *edgeArray,
        unsigned int *edgeArrayAux,
        unsigned int numVertices,
        int curr,
        int *flag) {
    int gid = blockDim.x * blockIdx.x + threadIdx.x;
    if (gid < numVertices) {
        if (levels[gid] == curr) {
            unsigned int nbr_off = edgeArray[gid];
            unsigned int num_nbr = edgeArray[gid + 1] - nbr_off;
            unsigned int _bDim = 32;
```

```
            unsigned int __threads = num_nbr;
            unsigned int _gDim = (__threads + _bDim - 1) / _bDim;
            if (__threads >= _THRESHOLD) {
                BFS_child<<<_gDim, _bDim>>>(levels, edgeArrayAux, curr,
                    nbr_off, num_nbr, flag);
            } else {
                BFS_child__serial(levels, edgeArrayAux, curr, nbr_off,
                    num_nbr, flag, _bDim, _gDim);
            }
        }
    }
}
```

Listing A.3: BFS CDP + C

```
__global__ void BFS_child(
        unsigned int *levels,
        unsigned int *edgeArrayAux,
        int curr,
        int nbr_off,
        int num_nbr,
        int *flag,
        unsigned int _gDim) {
    for (unsigned int _bx = blockIdx.x; _bx < _gDim; _bx += gridDim.x) {
        int gid = blockDim.x * _bx + threadIdx.x;
        if (gid < num_nbr) {
            int v = edgeArrayAux[gid + nbr_off];
```

```
            if (levels[v] == UINT_MAX) {
                levels[v] = curr + 1;
                *flag = 1;
            }
        }
    }
}
__global__ void BFS_parent(
        unsigned int *levels,
        unsigned int *edgeArray,
        unsigned int *edgeArrayAux,
        unsigned int numVertices,
        int curr,
        int *flag,
        unsigned int _gDim
        ) {
    int gid = blockDim.x*blockIdx.x + threadIdx.x;
    if(gid < numVertices){
        if(levels[gid] == curr){
            unsigned int nbr_off = edgeArray[gid];
            unsigned int num_nbr = edgeArray[gid + 1] - nbr_off;
            unsigned int _gDim = (num_nbr - 1) / 32 + 1;
            BFS_child<<< (_gDim + _CFACTOR - 1)/ _CFACTOR ,
                CHILD_BLOCK_SIZE>>>(levels, edgeArrayAux, curr, nbr_off,
                num_nbr, flag, _gDim);
        }
    }
```

```
}
```

Listing A.4: BFS CDP + T + C + A (grid granularity)

```
__device__ void BFS_child__serial(
        unsigned int *levels,
        unsigned int *edgeArrayAux,
        int curr,
        int nbr_off,
        int num_nbr,
        int *flag,
        unsigned int _bDim,
        unsigned int _gDim
) {
    for (unsigned _bx = 0; _bx < _gDim; ++_bx) {
        for (unsigned int _tx = 0; _tx < _bDim; ++_tx) {
            int gid = _bDim * _bx + _tx;
            if (gid < num_nbr) {
                int v = edgeArrayAux[gid + nbr_off];
                if (levels[v] == (2147483647 * 2U + 1U)) {
                    levels[v] = curr + 1;
                    *flag = 1;
                }
            }
        }
    }
}
```

```
__global__ void BFS_child(
        unsigned int **levels_array,
        unsigned int **edgeArrayAux_array,
        int *curr_array,
        int *nbr_off_array,
        int *num_nbr_array,
        int **flag_array,
        unsigned int *___gridDimConfig_C_array,
        unsigned int *__gDim_array,
        unsigned int *__bDim_array,
        unsigned int __parentSize,
        __GridMemPool __memPool
        ) {
    unsigned int __parentIdx = __find_parent_idx_kernel(__gDim_array,
        blockIdx.x, __parentSize);
    unsigned int *levels = levels_array[0];
    unsigned int *edgeArrayAux = edgeArrayAux_array[0];
    int curr = curr_array[0];
    int nbr_off = nbr_off_array[__parentIdx];
    int num_nbr = num_nbr_array[__parentIdx];
    int *flag = flag_array[0];
    unsigned int ___gridDimConfig_C =
        ___gridDimConfig_C_array[__parentIdx];
    unsigned int __gridDim_x = __gDim_array[__parentIdx] -
        ((__parentIdx == 0) ? 0 : __gDim_array[__parentIdx - 1]);
    unsigned int __blockDim_x = __bDim_array[__parentIdx];
    unsigned int __blockIdx_x = blockIdx.x - ((__parentIdx == 0) ? 0 :
```

```
        __gDim_array[__parentIdx - 1]);
    if (threadIdx.x < __blockDim_x) {
        for (unsigned int _bx = __blockIdx_x; _bx < ___gridDimConfig_C;
            _bx += __gridDim_x) {
            int gid = __blockDim_x * _bx + threadIdx.x;
            if (gid < num_nbr) {
                int v = edgeArrayAux[gid + nbr_off];
                if (levels[v] == UINT_MAX) {
                    levels[v] = curr + 1;
                    *flag = 1;
                }
            }
        };
    }
}



__global__ void BFS_parent(
        unsigned int *levels,
        unsigned int *edgeArray,
        unsigned int *edgeArrayAux,
        unsigned int numVertices,
        int curr,
        int *flag,
        unsigned int _gDim,
        __GridMemPool __memPool
        ){
```

```
int gid = blockDim.x * blockIdx.x + threadIdx.x;
unsigned int __nb = 0;
unsigned int __nt;
int __param3;
int __param4;
unsigned int __param6;
if (gid < numVertices) {
    if (levels[gid] == curr) {
        unsigned int nbr_off = edgeArray[gid];
        unsigned int num_nbr = edgeArray[gid + 1] - nbr_off;
        unsigned int _bDim = 32;

        unsigned int __threads = num_nbr;
        unsigned int __gridDimConfig = (__threads + _bDim - 1) /
            _bDim;
        if (__threads >= _THRESHOLD) {
            unsigned int _gDim = __gridDimConfig;
            __nb = (_gDim + _CFACTOR - 1) / _CFACTOR;
            __nt = _bDim;
            __param3 = nbr_off;
            __param4 = num_nbr;
            __param6 = _gDim;;
        } else {
            BFS_child__serial(levels, edgeArrayAux, curr, nbr_off,
                num_nbr, flag, _bDim, __gridDimConfig);
        }
    }
```

```
}
{
    unsigned int **levels_array = __memPool.grid_allocate<unsigned
        int *>(2);
    unsigned int **edgeArrayAux_array =
        __memPool.grid_allocate<unsigned int *>(2);
    int *curr_array = __memPool.grid_allocate<int>(2);
    int *nbr_off_array = __memPool.grid_allocate<int>(blockDim.x *
        gridDim.x);
    int *num_nbr_array = __memPool.grid_allocate<int>(blockDim.x *
        gridDim.x);
    int **flag_array = __memPool.grid_allocate<int *>(2);
    unsigned int *_gDim_array = __memPool.grid_allocate<unsigned
        int>(blockDim.x * gridDim.x);
    unsigned int *__gDim_array = __memPool.grid_allocate<unsigned
        int>(blockDim.x * gridDim.x);
    unsigned int *__bDim_array = __memPool.grid_allocate<unsigned
        int>(blockDim.x * gridDim.x);
    union scan_counter *_sc_ = __memPool.grid_allocate<union
        scan_counter>(1);
    unsigned int __gDim = __nb;
    if (__gDim > 0) {
        union scan_counter sc_local;
        sc_local.idx = 1;
        sc_local.nb = __gDim;
        sc_local.fused = atomicAdd(&(_sc_->fused), sc_local.fused);
        unsigned int _i_ = sc_local.idx;
```

```
            nbr_off_array[_i_] = __param3;
            num_nbr_array[_i_] = __param4;
            _gDim_array[_i_] = __param6;
            __gDim_array[_i_] = sc_local.nb + __gDim;
            __bDim_array[_i_] = __nt;
        } // __gDim > 0
        if (threadIdx.x == 0) levels_array[0] = levels;
        if (threadIdx.x == 0) edgeArrayAux_array[0] = edgeArrayAux;
        if (threadIdx.x == 0) curr_array[0] = curr;
        if (threadIdx.x == 0) flag_array[0] = flag;
    };
}

void launch_kernel(
        unsigned int *d_costArray,
        unsigned int *d_edgeArray,
        unsigned int *d_edgeArrayAux,
        unsigned int numVerts,
        int iters,
        int *d_flag
) {
    unsigned int numBlocks = (numVerts - 1) / PARENT_BLOCK_SIZE + 1;
    unsigned int _gDim = numBlocks;
    {
        static __MemPool __memPool(1073741824);
        __GridMemPool __memPoolHost(__memPool);
        __GridMemPool __memPoolDevice = __memPoolHost;
```

```
unsigned int __gDim = _gDim;
unsigned int __bDim = 1024;
unsigned int **levels_array_child =
    __memPoolHost.grid_allocate<unsigned int *>(2);
unsigned int **edgeArrayAux_array_child =
    __memPoolHost.grid_allocate<unsigned int *>(2);
int *curr_array_child = __memPoolHost.grid_allocate<int>(2);
int *nbr_off_array_child =
    __memPoolHost.grid_allocate<int>(__gDim * __bDim);
int *num_nbr_array_child =
    __memPoolHost.grid_allocate<int>(__gDim * __bDim);
int **flag_array_child = __memPoolHost.grid_allocate<int *>(2);
unsigned int *_gDim_array_child =
    __memPoolHost.grid_allocate<unsigned int>(__gDim * __bDim);
unsigned int *gDim_array_child =
    __memPoolHost.grid_allocate<unsigned int>(__gDim * __bDim);
unsigned int *bDim_array_child =
    __memPoolHost.grid_allocate<unsigned int>(__gDim * __bDim);
union scan_counter *_sc_ = __memPoolHost.grid_allocate<union
    scan_counter>(1);
cudaMemset(gDim_array_child, 0,(__gDim * __bDim + __gDim *
    __bDim) * sizeof(unsigned int) + sizeof(union
    scan_counter));
BFS_parent<<< __gDim, __bDim >>>(d_costArray, d_edgeArray,
    d_edgeArrayAux, numVerts,iters, d_flag, _gDim,
    __memPoolDevice);
union scan_counter sc_child;
```

```
        cudaMemcpy(&sc_child, _sc_, sizeof(union scan_counter),
            cudaMemcpyDeviceToHost);
        unsigned int __gDim_child = sc_child.nb;
        unsigned int __bDim_child = kernel_max(bDim_array_child, __gDim
            * __bDim);
        BFS_child_h(
                __gDim_child,
                __bDim_child,
                levels_array_child,
                edgeArrayAux_array_child,
                curr_array_child,
                nbr_off_array_child,
                num_nbr_array_child,
                flag_array_child,
                _gDim_array_child,
                gDim_array_child,
                bDim_array_child,
                sc_child.idx,
                __memPoolHost
        );
    };
}
```

Appendix B

Abbreviations

DP	Dynamic Parallelism
CDP	CUDA Dynamic Parallelism
CPU	Central Processing Unit
GPU	Graphical Processing Unit
BFS	Breadth First Search
SSSP	Single Source Shortest Path
SP	Survey Probagation
TC	Triangle Counting
BT	Bezier Tessellation
MSTF	Minimum Spanning Tree Find
MSTV	Minimum Spanning Tree Verify

Bibliography

[1] D. Kirk *et al.*, "Nvidia cuda software and gpu parallel computing architecture," in *ISMM*, vol. 7, pp. 103–104, 2007.

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

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

[4] Y. Yang and H. Zhou, "CUDA-NP: Realizing nested thread-level parallelism in GPGPU applications," in *ACM SIGPLAN Notices*, vol. 49, pp. 93–106, ACM, 2014.

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

[6] I. El Hajj, J. Gómez-Luna, C. Li, L.-W. Chang, D. Milojicic, and W.-m. Hwu, "KLAP: Kernel launch aggregation and promotion for optimiz-

ing dynamic parallelism," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–12, IEEE, 2016.

[7] 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*, pp. 649–660, IEEE, 2017.

[8] 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*, pp. 528–540, ACM, 2015.

[9] J. Wang, *Acceleration and optimization of dynamic parallelism for irregular applications on GPUs*. PhD thesis, Georgia Institute of Technology, 2016.

[10] X. Tang, A. Pattnaik, O. Kayiran, A. Jog, M. T. Kandemir, and C. Das, "Quantifying data locality in dynamic parallelism in gpus," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 2, no. 3, pp. 1–24, 2018.

[11] 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.

[12] 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*, p. 5, ACM, 2015.

[13] 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*, pp. 979–988, IEEE, 2015.

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

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

[16] 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*, pp. 210–219, IEEE, 2015.

[17] J. A. Stratton, N. Anssari, C. Rodrigues, I. Sung, N. Obeid, L. Chang, G. D. Liu, and W. Hwu, "Optimization and architecture effects on gpu computing workload performance," in *2012 Innovative Parallel Computing (InPar)*, 2012.

[18] V. Volkov and J. W. Demmel, "Benchmarking gpus to tune dense linear algebra," in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008.

[19] J. Gómez-Luna, J. González-Linares, J. Benavides, and N. Guil, "An optimized approach to histogram computation on gpu," *Machine Vision and Applications*, vol. 24, no. 5, pp. 899–908, 2013.

[20] J. Gómez-Luna, J. M. González-Linares, J. I. Benavides Benítez, and N. Guil Mata, "Performance modeling of atomic additions on gpu scratchpad mem-

ory," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 11, pp. 2273–2282, 2013.

[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*, pp. 16–30, Springer, 2008.

[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*, pp. 257–268, IEEE Computer Society, 2015.

[23] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pp. 75–86, IEEE, 2004.

[24] 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*, GPGPU-3, (New York, NY, USA), p. 63–74, Association for Computing Machinery, 2010.

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

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

[27] 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.

[28] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, (Manhattan, USA), pp. 595–601, ACM Press, 2004.

[29] P. Sanders, C. Schulz, and D. Wagner, "Benchmarking for graph clustering and partitioning," 2014.

[30] A. Belov, D. Diepold, M. J. Heule, and M. Järvisalo, "Proceedings of sat competition 2014," 2014.

[31] DIMACS, "9th dimacs implementation challenge - shortest paths," 2006.

[32] 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*, ICPE '15, pp. 253–264, ACM, 2015.

[33] 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*, pp. 451–456, IEEE, 2016.

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

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

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

[37] 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*, pp. 31–40, IEEE, 2014.

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

[39] 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.

[40] 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*, pp. 4242–4245, IEEE, 2015.

[41] 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*, pp. 21–38, Springer, 2016.

[42] 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.

[43] 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*, pp. 240–245, IEEE, 2016.

[44] 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.

[45] 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.

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

[47] 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*, pp. 1–4, IEEE, 2014.

[48] 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.

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

[50] 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.

[51] 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.

[52] 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*, IA3 '15, pp. 11:1–11:4, ACM, 2015.

[53] 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.

[54] 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*, pp. 342–349, Springer, 2013.