GPU Wavefront Splitting for Safety-Critical Systems

by

Artem Klashtorny

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2022

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

The introductory paragraphs in Chapter 1 were co-written with Anirudh Mohan Kaushik. Chapters 2 to 7 were entirely written by Artem Klashtorny.

Abstract

Graphics processing units (GPUs) are compute platforms that are ideal for highly parallel workloads due to their high degree of hardware parallelism. Parallelism offered by GPUs lends itself well to machine learning and computer vision applications, including in safety-critical systems. Safety-critical systems require a guarantee of timing predictability. Guaranteeing timing predictability means being able to statically analyze the worst-case execution time (WCET) of the GPU program. Unfortunately, existing GPUs are designed for average-case performance and are thus not designed for timing predictability. Consequently, there is potential for research effort to provide these guarantees.

Prior research works have proposed several new techniques to improve performance. One such technique is wavefront splitting, which reduces the number of idle threads on the GPU and increase utilization. However, no prior work addresses the WCET of this technique. The purpose of this thesis is to develop a GPU implementation for safety-critical systems that leverages wavefront splitting and to enable analysis of the WCET in such an implementation.

Acknowledgements

I would like to thank my supervisor Professor Hiren Patel for his guidance throughout my graduate studies so far. I also acknowledge and thank the rest of the Computer Architecture and Embedded Systems Research (CAESR) team at the University of Waterloo. Special thanks to Zhuanhao Wu and Anirudh Mohan Kaushik for their continued support over the past two years.

I would also like to thank my family for their continued support of my endeavours and for helping me with getting motivated at the most challenging times.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graphics processing units (GPUs) are compute platforms that are ideal for highly parallel workloads. GPUs have a high degree of hardware parallelism compared to traditional multi-core platforms; current generation GPUs offer thousands of cores compared to the tens of cores on current generation CPUs [28, 17].

The execution model on GPUs operates on threads, known as work-items. The GPU groups work-items together into warps [19] (on NVIDIA architectures) or wavefronts [10] (on AMD architectures[1]). Wavefronts execute a single instruction at a time on vectorized data in a lockstep fashion, referred to as single instruction multiple data (SIMD) [1]. Multiple wavefronts are mapped for execution on a GPU core, which comprises of vectorized arithmetic units and register files [10].

Several key workloads in scientific computation [33], machine learning (ML) [19], and computer vision (CV) [19] feature parallel execution models. Compared to conventional multi-core platforms, GPUs are auspicious accelerators for such parallel workloads due to the abundant availability of parallel hardware compute units.

1.1 GPU Usage in Safety-Critical Systems

Among real-world parallel workloads, applications in ML and CV are now deployed in safety-critical systems. Safety-critical systems are systems in which failure to meet timing constraints can result in economic or environmental costs as well as injury or loss of life.

[1]This thesis focuses on the AMD terminology, as explained in Section 2.2.1 and Appendix A

They require a guarantee of timing predictability; it must be possible to theoretically analyze the worst-case temporal behaviour of the system and prove that timing requirements are met. Figure 1.1 demonstrates the key components for timing predictability. The theoretical worst-case guarantee provides an upper bound on any observed worst-case execution time.

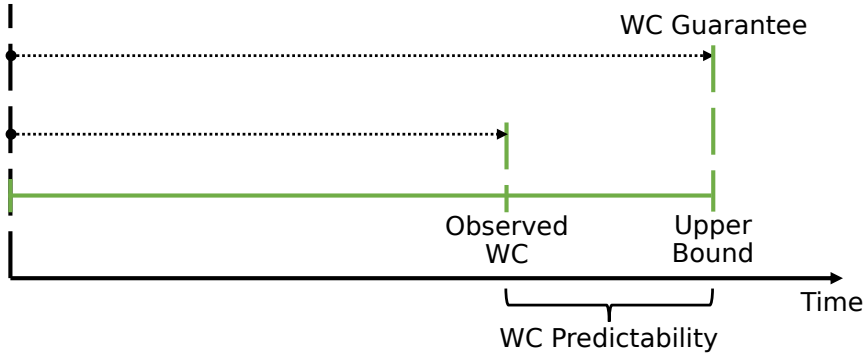
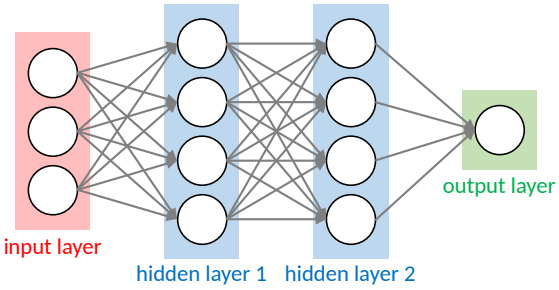
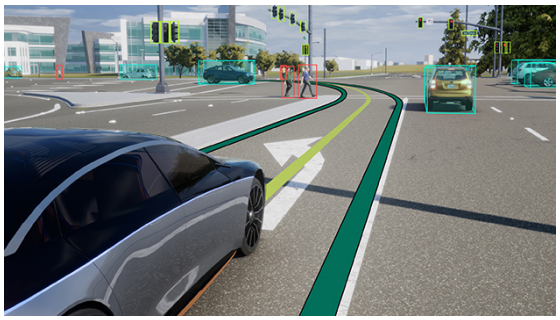


Figure 1.1: Worst-case timing predictability components [39]

Examples of domains in safety-critical systems which utilize ML and CV include advanced driver assistance systems (ADAS), industrial robotics, and avionics [34]. Taking a specific example, the autonomous driving domain deploys ML and CV workloads to process visual information about the environment to detect objects and people and to engage in procedures to safely steer the vehicle [24], as shown in Figure 1.2.



(a) GPUs are well-suited for data-intensive deep learning operations [38]



(b) NVIDIA DRIVE GPU platform for advanced driver assistance systems (ADAS) [29]

Figure 1.2: GPU application in the autonomous driving domain

Despite the safety-critical nature of these workloads, they are executed on GPUs that

maximize average-case performance as the main design focus rather than worst-case execution time, which is crucial for timing predictability of safety-critical systems [41]. To bridge this gap, there is a large body of research devoted towards analyzing and measuring the safety-critical fitness of commercial off-the-shelf GPU architectures and to propose techniques that address their shortcomings that compromise safety-critical requirements [32, 2, 18]. As ML and CV workloads become integral in safety-critical automotive and avionics domains, it is crucial that future GPU architectures are designed not only to accelerate their execution time but to do so in a manner that does not threaten safety-critical requirements. Designing such GPU architectures means simultaneously designing for high performance and timing predictability.

There exists a wealth of existing research on designing predictable general-purpose compute architecture. It may initially seem viable to apply such strategies to GPU architectures; however, the majority of these works are not compatible [41]. General-purpose compute architectures are designed to minimize instruction latency; they feature elaborate multi-level cache hierarchies, sophisticated branch prediction mechanisms, and out-of-order instruction scheduling. On the other hand, GPU architectures are designed to maximize instruction throughput; they trade the architectural optimizations of general-purpose compute for more arithmetic compute units. Details of these optimizations are often not revealed in full detail by GPU manufacturers, making analysis and assertions about timing predictability difficult [41, 40, 2]. As a result, addressing the timing predictability of GPUs requires novel analyses and approaches.

1.2 Tackling Branch Divergence

The focus of this thesis is on addressing the impact of branch divergence on the timing predictability of GPUs. Branch divergence occurs when work-items in a wavefront follow different execution paths due to a branch condition [1]. Work-items in a wavefront execute in lockstep. To reconcile the need to execute different instructions with the lockstep execution paradigm, the GPU serializes the execution of work-items in the two divergent paths [1]. Burtscher et al. show that practical workloads exhibit branch divergence contributing to 4% of total executed instructions, also establishing that branch divergence is a key impediment to average-case performance in GPUs [6]. Fung et al. show that, in the average case, branch divergence results in 20% of the instruction count per cycle of an ideal non-lockstep architecture with full utilization[2] [14].

[2]A non-lockstep architecture like this is referred to as *multiple instruction multiple data* (MIMD)

There are several research works which tackle the average-case performance loss due to branch divergence. These works explore some form of either breaking up the lockstep execution of a wavefront or regrouping the work-items within wavefronts. One such example is wavefront splitting, through which work-items from divergent paths are subdivided into two or more separate schedulable entities. The wavefront is split according to the diverging outcomes of the branch divergence; the work-items executing one path form one split and those executing the other path form another split. Thus, wavefront splitting allows the GPU to prevent the serialization of branched path execution. The technique, along with others reducing branch divergence, have been shown by prior work to improve average-case performance [26, 14, 27].

The impact of wavefront splitting and other similar techniques on the timing predictability of GPU kernel execution has received little attention from the research community [18]. As a result, they are poorly understood for the worst-case. Developers of real-time and safety-critical systems may wish to use these performance enhancements, but need to be able to statically analyze their behaviour. To that end, this thesis focuses on providing an analysis for timing predictability of wavefront splitting. The objective of this thesis is to enable wavefront splitting for safety-critical systems and to estimate the worst-case execution time (WCET) of a GPU kernel using a hybrid analysis approach.

1.3 Thesis Contributions

This thesis presents two key contributions, relating to the implementation of wavefront splitting and subsequent worst-case analysis.

1. **Explore wavefront splitting in safety-critical systems**. Prior work deployed wavefront splitting to address branch divergence for average-case performance. The main goal of this thesis is to explore whether wavefront splitting can also be leveraged in safety-critical systems to address branch divergence. The results show that wavefront splitting can be introduced in safety-critical applications given the implementation described in Chapter 4.

    - **Wavefront splitting targeting AMD hardware.** Prior work primarily focused on wavefront splitting on NVIDIA platforms. This thesis adapts wavefront splitting strategies to specifically target AMD hardware because it is open-source and hence better suited to worst-case analysis [30]. The implementation of these strategies is completed using the gem5 simulator [23], which implements previous generation AMD GPU hardware [15].

- **Static techniques for wavefront splitting.** Opportunistic splitting at runtime makes predicting the worst-case behaviour difficult. This thesis presents a strategy to statically mark branches in GPU kernels where a programmer would like the GPU to split. Splitting statically allows for worst-case modeling of the number of splits which occur per wavefront.

2. **Enable analysis of wavefront splitting.** Prior work has used hybrid analysis to estimate the WCET of GPU kernels [4]. Hybrid analysis refers to using empirical worst-case measurements of small program components to model the overall WCET of a program. However, prior work has not included wavefront splitting in the hybrid analysis model. This thesis aims to enable the hybrid analysis approach to incorporate wavefront splitting. However, blindly applying the hybrid analysis models in prior work is overly pessimistic under wavefront splitting. Hence, this thesis aims to introduce optimizations that better model the WCET when splitting. Furthermore, the presented analysis shows that for some workloads, the WCET bound is reduced when splitting.

1.4 Thesis Outline

The thesis is organized into chapters as follows.

- Chapter 1 discusses the position and contributions of the thesis in the broader context of research in GPUs for real-time systems. It outlines the problem of branch divergence and the appeal of solutions like wavefront splitting to resolve it.

- Chapter 2 presents a background on program analysis and GPU architecture. The discussion of GPU architecture covers the hardware organization as well as the software model specific to AMD GPUs.

- Chapter 3 reviews prior work aiming to reduce the performance effects of branch divergence. It also reviews works which examine GPUs as accelerators for real-time and safety-critical systems and the use of hybrid analysis on GPUs.

- Chapter 4 describes the implementation of wavefront splitting on AMD hardware. It covers a few key techniques that build on the limitations of prior work, including the static assignment of splits previously mentioned in Section 1.3.

- Chapter 5 presents the hybrid analysis framework for GPUs with wavefront splitting as described in the thesis. It explains how to extract a graphical representation of GPU code and apply it to an analytical model of program execution on the GPU hardware.
- Chapter 6 applies the implementation to a GPU benchmarking suite to determine performance compared to the baseline implementation without splitting. It uses these results to make conclusions about performance implications of the implementation.
- Chapter 7 reiterates the contributions and findings of the thesis. It outlines the limitations of the work and subsequent opportunities for future work.

Chapter 2

Background

Wavefront splitting leverages features of GPUs to reduce the impact of branch divergence, a key performance limitation. To that end, this chapter presents a background on the GPU execution model and the hardware it employs. Additionally, this chapter discusses the worst-case analysis of programs using the control-flow graph, or CFG.

2.1 WCET Analysis of Programs

Traditional WCET analysis relies on the CFG for a program. A CFG is a directed graph constructed to represent the execution of a program. The nodes of this graph correspond to basic blocks within the program; a basic block is a set of consecutive instructions in program order that do not contain any jumps. The edges in the graph correspond to sequences of basic blocks that can take place during execution. For basic programs[1], there are a few fundamental components that comprise a CFG, shown in Figure 2.1:

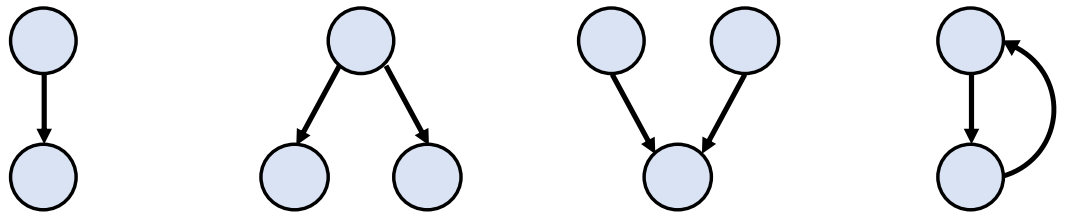
(a) a sequence with a single direct successor in program order without branches or jumps,

(b) a divergent path created by conditionally branching execution,

(c) the convergence of two paths resulting from a previous conditional branch, and

(d) a loop back to a previous basic block created by a conditional or unconditional branch.

[1]Basic programs here include any program that does not include arbitrary jump instructions, such as those generated by `break`, `continue`, or `goto` statements.



(a) Sequence (b) Divergence (c) Convergence (d) Loopback

Figure 2.1: Basic control-flow graph components

The CFG can be used to determine the WCET of the program. A common analysis technique for determining the WCET of a program is called the implicit path enumeration technique, or IPET. IPET is an integer linear programming (ILP) optimization problem; the optimization variable is an integral quantity representing the execution count of each basic block. Given a vector of latencies for each basic block, the objective of the optimization problem is to maximize the execution time of the CFG. In order to bound the optimization problem, IPET enforces constraints on the ILP based on the structure of the CFG. The execution counts must conform to the flow constraints of the graph. For instance, if two paths converge, the target node execution count must be the sum of the execution counts for the two paths entering it. Additionally, loops must be constrained by some statically known bound. This problem can be solved using an ILP solver and yields the overall WCET of a program.

2.2 Graphics Processing Units

This section details the architecture and programming model for GPUs which will be used by the remainder of the thesis.

2.2.1 Terminology

Though GPUs in some form have existed for decades, the term "GPU" was created by NVIDIA in 1999, with the release of the GeForce 256 [25]. NVIDIA introduced many of the terms used by GPU researchers today. However, the terms refer to proprietary components of which only certain details are revealed to the public. AMD is NVIDIA's major competitor and exposes more information about GPU implementation details to users. Un-

fortunately, the existence of two manufacturers independently developing GPUs that are not fully open-source has created two separate sets of common terms and definitions for the components of the GPU. Although there are some unique elements to each company's GPUs, most of these terms represent overlapping concepts.

In prior research work, NVIDIA has usually been the preferred baseline for terminology. However, the proprietary nature of much of its hardware architecture and software stack has made the open-source AMD environment more attractive for real-time applications [31]. This thesis presents an implementation of wavefront splitting in gem5, an event-driven simulator with support for modelling AMD GPU hardware. Therefore, the focus will be on AMD concepts and terminology. For a table of corresponding terms in NVIDIA architecture, refer to Appendix A.

2.2.2 Execution Model

Achieving the massively parallel model of the GPU starts with the behaviour of a single thread, or work-item. The GPU programmer describes the behaviour of each work-item in a function known as a kernel. The work-items execute the same instructions in parallel on some vectorized data. This is known as the single instruction multiple data (SIMD) execution model, and is illustrated in Figure 2.2.

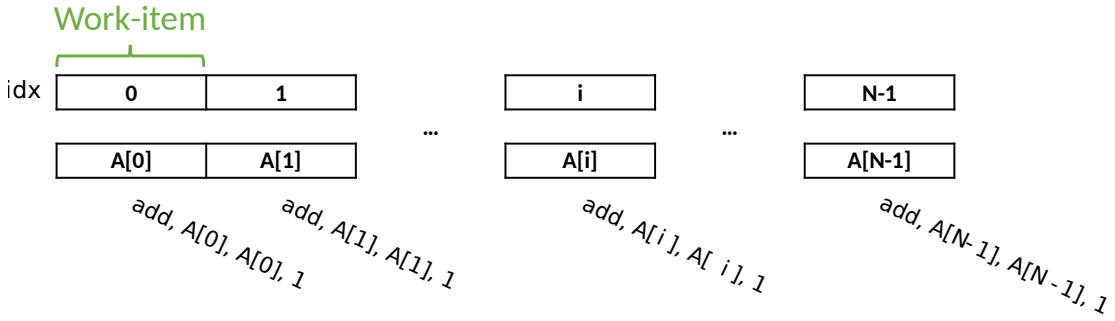


Figure 2.2: SIMD execution of an instruction by multiple work-items

The number of work-items may exceed the hardware resources of the GPU; hence, programmers group work-items into collections known as workgroups. When launching the kernel, the GPU programmer specifies the number of workgroups and work-items per workgroup. Each work-item is assigned a workgroup index and work-item index within its workgroup. As an example of a kernel, Figure 2.3 demonstrates a basic vector addition kernel, which supports integer or floating point vectors.

```
template <typename T>
__global__ void
vector_add(hipLaunchParm lp, T *A_d, size_t N){
    size_t idx = hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x;
    if(idx < N) A_d[idx] += 1;
}
```

Figure 2.3: Sample GPU kernel for vector addition

2.2.3 Hardware Architecture

The GPU consists of many simple cores which can carry out massively parallel workloads. Figure 2.4 depicts a generalized version of the GPU microarchitectural organization in modern AMD GPUs. In this thesis, wavefront splitting is implemented in the gem5 simulator. The GPU model in gem5 implements the third generation of the AMD Graphics Core Next (GCN) architecture. GCN has since been succeeded by the Radeon DNA (RDNA) microarchitecture which implements a few performance improvements. Nevertheless, this thesis maintains a focus on GCN because it is supported in gem5.

Current generation AMD GPUs are comprised of one or more processors [9, 10] as shown in Figure 2.4. Each processor is organized into four compute units (CUs). A CU consists of a set of four SIMD vector execution pipelines, a scalar execution unit, a local data share, and a shared memory interface into global memory. A SIMD unit consists of 16 ALUs, collectively known as a vector ALU (VALU) and a vector register file (VGPR), allowing for 16 parallel integer or floating point operations to occur on each SIMD simultaneously. Instructions in each SIMD are executed in lockstep: all 16 ALUs execute the same operation on distinct data elements.

The GPU dispatches workgroups to the CUs and SIMDs. Workgroup dispatch takes place in a round-robin fashion until there is no room left for a full workgroup or until there are no more workgroups. Work-items from the same workgroup share barrier and local memory resources. Sets of 64 work-items are grouped together into wavefronts. Each SIMD unit holds the execution context for 10 wavefronts, stored in wavefront slots. The execution context contains vector registers and the wavefront program counter (PC). When the SIMD's ALUs are free, it chooses a ready wavefront for execution from one of its

[2]In the more recent RDNA architecture, wavefront size is reduced to 32 work-items and there are 32 ALUs per SIMD, allowing a throughput of one instruction every cycle rather than every four cycles [10].

wavefront slots. Since there are 64 work-items in a wavefront and 16 ALUs in the vector pipeline, the operation takes four cycles to complete [2]. SIMD units execute wavefront instructions in program order.

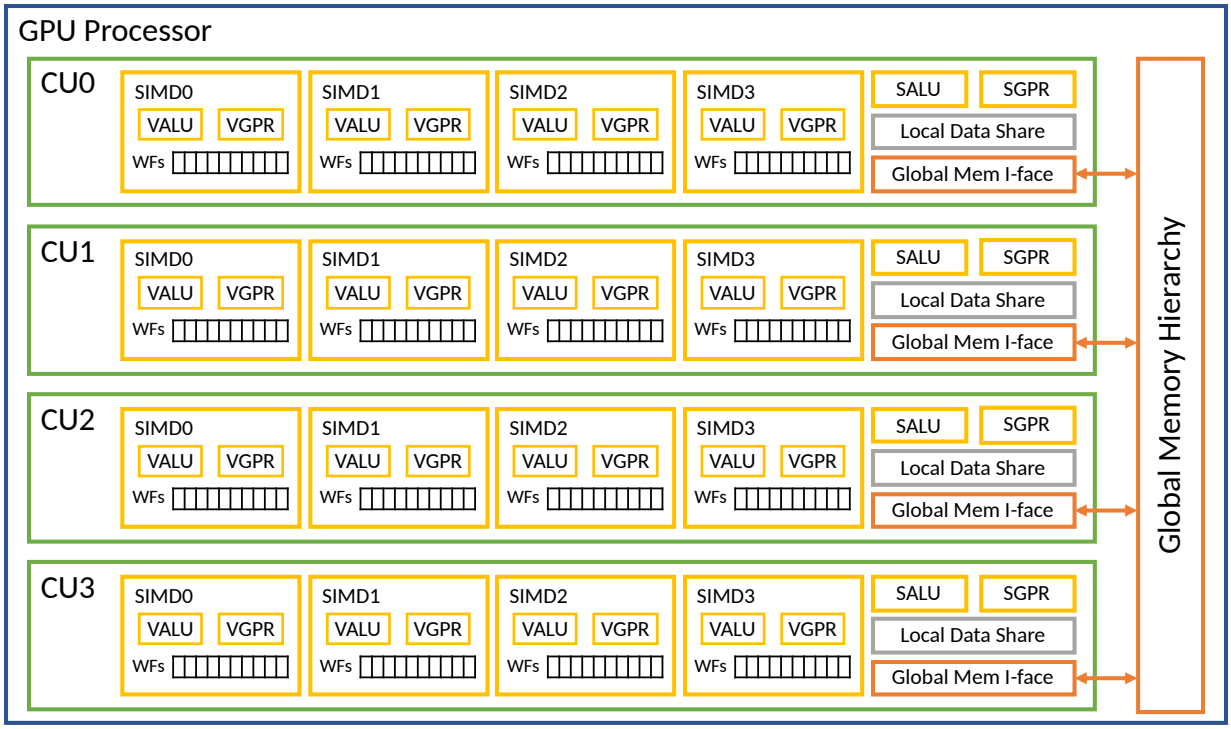


Figure 2.4: AMD Graphics Core Next architecture diagram

The benefit to having many wavefront contexts stored on each SIMD is that it allows for latency hiding. Latency hiding is a GPU design optimization which enables context switching to another wavefront when a wavefront is stalled. This optimization reduces the stall cycles for the functional units on the CU. Due to their typically long latency, instructions that have a read-after-write dependency on a previous access to global memory will trigger such a context switch.

2.2.4 HIP Programming Model

GPU programs are typically written using a model framework on top of an established programming language. The most prominent example in research and industry is CUDA, which is a software model developed by NVIDIA. AMD GPUs use the heterogeneous

interface for portability (HIP), which is very similar to CUDA, with mostly minor syntactic differences. HIP is developed as a library for C/C++ programs. Since the focus of this thesis is on AMD GPUs, the programming model is described from the perspective of HIP.

HIP programs contain a mixture of code that runs on the CPU and GPU. The CPU and GPU components share many programming features. To avoid confusion between the two, the terms "host" and "device" are used as descriptors for data and code pertaining to the CPU and GPU, respectively. HIP code is organized into functions that are designed to be executed on the GPU, known as kernels. A HIP program that executes on the GPU has at least one kernel but must start its execution on the CPU. The host code is responsible for preparing the data structures to be used by the GPU program, launching the program to the GPU, and collecting and cleaning up data once the GPU is done.

The kernel, such as the one found in Figure 2.3, falls into the category of device code. The `__global__` keyword indicates to the compiler that the function is a kernel. In HIP syntax, the workgroup index and work-item index are represented by `hipBlockIdx_x` and `hipThreadIdx_x`, respectively. The first line of the vector addition kernel computes a globally unique work-item index using the workgroup size (`hipBlockDim_x`) and local offset. The suffix `_x` corresponds to one out of three possible dimensions; however, for the sake of simplicity, the workgroup considered is one-dimensional.

Figure 2.5 depicts a simplified version of the host code for the sample vector addition kernel in Figure 2.3. The CPU will

1. Allocate and initialize the host vector `A_h`,
2. Allocate device vector `A_d`,
3. Copy the host data to the device,
4. Launch the kernel given a desired number of workgroups and work-items per group,
5. Copy the device data to the host,
6. If applicable, use the output data, and
7. Deallocate the host and device data.

HIP also supports "zero-copy GPU access" where the GPU can directly access the host vectors. However, the access latency is an order of magnitude longer due to the delay incurred by the interconnect between the CPU and GPU [11]. Therefore, most practical applications follow the aforementioned sequence.

```
1         // Allocate and initialize host vector
2         A_h = (int *)malloc(num_bytes);
3         for (int i = 0; i < N; i++) A_h[i] = i;
4
5         // Allocate device memory
6         hipMalloc(&A_d, num_bytes);
7
8         // Copy data to device
9         hipMemcpy(A_d, A_h, num_bytes, hipMemcpyHostToDevice);
10
11        // Launch kernel
12        const unsigned groups = 512;
13        const unsigned itemsPerGroup = 256;
14        hipLaunchKernel(vector_add, dim3(groups), dim3(itemsPerGroup),
15                        0, 0, A_d, N);
16
17        // Copy data to host
18        hipMemcpy(A_h, A_d, num_bytes, hipMemcpyDeviceToHost);
19
20        // Deallocate device memory
21        hipFree(A_d);
22
23        // Use output and deallocate host vectors
24        for (int i = 0; i < N; i++) printf("%d\n", A_h[i]);
25        free(A_h);
```

Figure 2.5: Host code for HIP vector addition example

2.2.5 Branch Divergence

One of the key design functionalities of GPUs that allow them to achieve high levels of parallelism and performance is the SIMD execution model. Each instruction in the GPU kernel is executed by every work-item in the wavefront in lockstep. Typically, the GPU execution hardware performs operations on vectorized data. Each work-item has a unique index which the kernel program code can access. As mentioned in Section 2.2.4, the GPU hardware uses a function of the index to determine the element in the vectorized data on

which the work-item operates. Since many of these operations take place in lockstep, the GPU is able to achieve significant parallelism.

The SIMD model works well on simple kernels without complex control flow. However, a key problem arises when conditional statements cause the control flow of the work-items within a wavefront to diverge; this is known as *branch divergence* or *control-flow divergence*. Conditions based on either the work-item index or on the vector data element can cause such divergence within a wavefront, such as in a loop or a conditional block. Traditional applications of GPUs in gaming did not have significant branch divergence; however, ML workloads have more complex control-flow patterns that lead to a greater impact from branch divergence [14]. Figure 2.6 shows an example of such divergence [1]. The program has several conditional statements that diverge based on the work-item index.

```
1         // Basic Block 1
2         if (threadIdx < 3) {
3             // Basic Block 2
4             if (threadIdx == 0) {
5                 // Basic Block 3
6             } else {
7                 // Basic Block 4
8             }
9         } else {
10            // Basic Block 6
11        }
12        // Basic Block 7
```

Figure 2.6: Sample HIP kernel code exhibiting branch divergence

Figure 2.7 shows the program in control-flow graph (CFG) form to clearly illustrate the branching paths. Each basic block is shown as a vertex BBi in the graph with edges corresponding to the control flow paths of the program.

As mentioned in Section 2.2.3, SIMD vector ALUs execute the same operation in lockstep; they do not have the ability to execute disjoint operations. As a result, the SIMD unit is unable to execute multiple branches of execution in parallel. To overcome this problem, the GPU execution flow is designed to serialize the diverging paths of the work-items. The GPU marks the work-items that should not perform the operations in each path to ensure that the serialization does not change the kernel semantics; this process is known

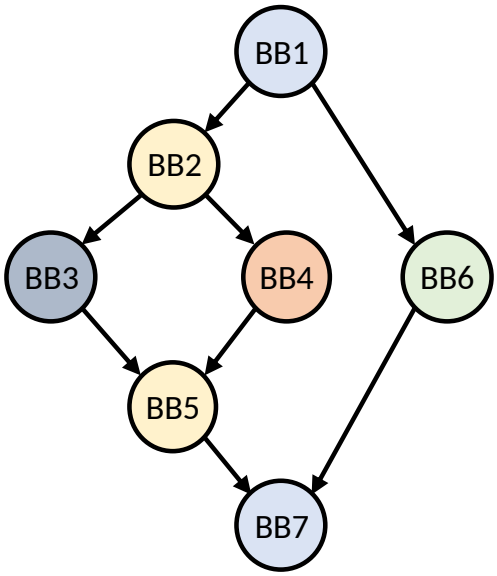


Figure 2.7: Sample CFG exhibiting divergence

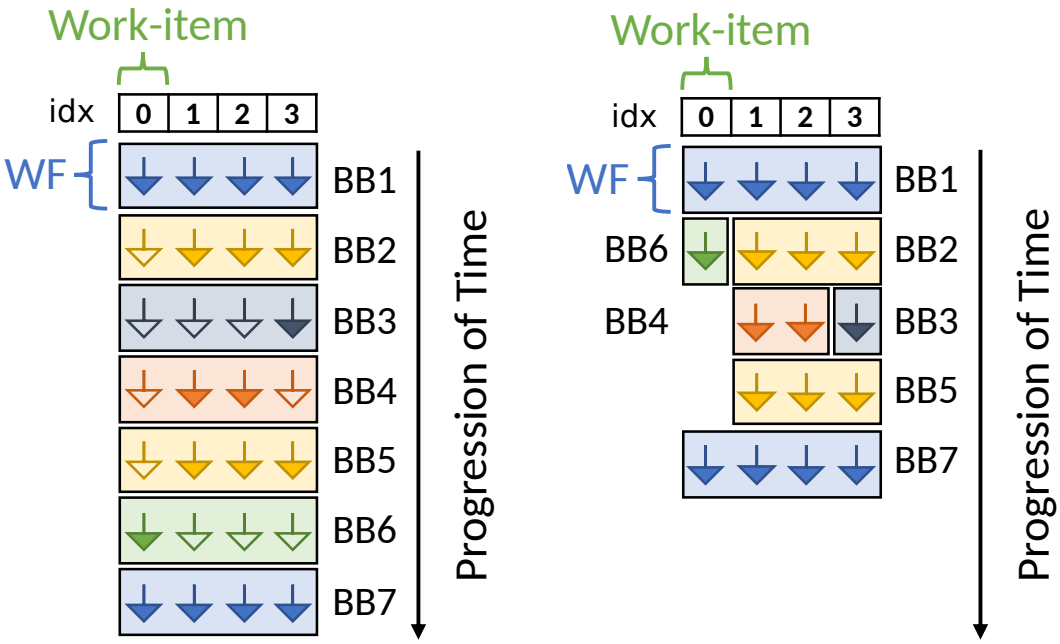
as masking. Each wavefront has an execution mask as one of its registers; each bit in the mask corresponds to a work-item in the wavefront. When executing each path, masked work-items do not perform any operation.

In cases of branch divergence, the execution mask consists of some masked and some unmasked work-items. These masks can be described as heterogeneous. Execution masks that have either all work-items masked or unmasked are described as homogeneous. For the purpose of analysis, a wavefront with a homogeneous mask can be described as *fully masked* or *fully unmasked*. A fully masked wavefront corresponds to a wavefront with an execution mask of all zeros. Similarly, a fully unmasked wavefront corresponds to a wavefront with an execution mask of all ones.

Figure 2.8a depicts the execution of a single wavefront that has four work-items, where actively executing unmasked work-items are represented by solid arrows. After completing one path, the masks are flipped and execution proceeds to the other path. For instance, the first row shows that $BB1$ can execute on all four work-items. In $BB2$, only three of the four work-items execute the instructions in the block. This approach allows the GPU to handle conditional logic based on the work-item identity but is detrimental to performance and hardware utilization.

Some prior works have introduced wavefront splitting to reduce the impact of branch divergence. With splitting, the GPU can partition work-items into smaller schedulable

entities and execute them in an interleaved fashion or in parallel. This behaviour is shown in Figure 2.8b; by overlapping multiple split wavefronts together, there is an opportunity for improved performance. More details about this technique will be described in Chapters 3 and 4.



(a) Baseline behaviour without splitting

(b) Behaviour with splitting

Figure 2.8: Wavefront execution flow with and without splitting

Chapter 3

Related Work

There exist a multitude of prior works which aim to address branch divergence. This thesis builds on the findings and proposals of several of these works. They typically argue for a hardware modification to enable better utilization of SIMD hardware or for software algorithms which reduce scheduling latencies. This chapter introduces these related works. Section 3.1 starts by taking a general look at some of the approaches which address divergence through work-item regrouping or re-organization. Section 3.2 goes into more detail with past works proposing wavefront splitting. Findings from these works inspire and direct the implementation decisions discussed in Chapter 4. Finally, Section 3.3 walks through works which analyze GPUs for applications in real-time and safety-critical systems, including some case studies and includes a description of how a prior work applied hybrid analysis to GPUs.

Note that most of these prior works deal with NVIDIA architecture, in which threads and warps are equivalent to work-items and wavefronts in the AMD environment. For more information about the terminology differences, see Appendix A.

3.1 Work-item Regrouping

A few prior works have approached the utilization problem of branch divergence by regrouping work-items. These approaches take active work-items from various wavefronts and combine them to be executed in lockstep. Some of these ideas are quite similar to the concept of wavefront splitting but are more flexible with SIMD execution. The thesis uses some of these ideas to break up the SIMD behaviour and work with individual work-items.

3.1.1 Dynamic Warp Formation

Fung et al. propose Dynamic Warp Formation (DWF) [14]. Upon encountering branch divergence, there may be many wavefronts that suffer from SIMD under-utilization. Every cycle, DWF will form new wavefronts from a pool of work-items that are ready to execute and have the same PC. Typically, work-items are assigned to a vector ALU at a specific index, known as a lane, and are unable to be moved. However, DWF allows mapping work-items to any lane; therefore, it also must expand the vector register file to allow access from any ALU. To enable this arbitrary access, DWF uses a crossbar for each SIMD unit. Alternatively, Fung et al. also propose "lane-aware" DWF, which restricts work-items to their original lane. This approach does not maximize utilization but prevents stalls due to crossbar contention. Fung et al. show that the speed-up of the basic DWF technique is 22.5% and speed-up for lane-aware DWF is 20.7% with an area overhead of 4.7%.

3.1.2 Large Warp Microarchitecture

Narasiman et al. propose the Large Warp Microarchitecture (LWM) [27], which expands upon DWF. It proposes grouping wavefronts into larger units where each wavefront is given a row index. When the units experience branch divergence, the GPU selects active work-items from each wavefront to be executed on SIMD units. Each of these lanes points to the particular row index which contains the work-item being executed. To prevent all wavefronts from arriving at the same instruction sequence causing a long-latency stall at the same time, LWM is paired with "two-level" warp scheduling, which schedules wavefront groups in a round-robin fashion. These two techniques combined provide a 19.7% performance improvement with a 2.5% area overhead cost.

3.1.3 On-GPU Thread-Data Remapping

Lin et al. propose Thread-Data Remapping (TDR) for GPUs, calling it WAGNERR [22]. WAGNERR is a static analysis tool that analyzes branch divergence in the kernel code. They propose two separate algorithms to find data-based branch divergence, arguing that divergence based solely on work-item index can be avoided statically by GPU programmers. The two algorithms, called Head-or-Tail and Data Group Indexing both remap work-items by recomputing their index computed by the kernel. The former is a simpler and more memory-efficient algorithm while the latter allows for complex control divergence patterns. Lin et al. show a 2.4% to 12.4% performance improvement using WAGNERR depending on the benchmark used.

3.1.4 Variable Warp Size

Rogers et al. propose a Variable Warp Size (VWS) architecture [37]. VWS splits wavefronts into smaller units of four work-items which can execute GPU programs entirely independently. These units are combined into groups called "gangs"; the grouping is performed by the Warp Ganging Unit (WGU). The WGU overrides local fetch, decode, and issue decisions for a gang. It maintains information about gangs in a gang table. Splitting and reconvergence of gangs is determined by heuristics, such as the occurrence of branch divergence. VWS provides a 35% performance improvement with a 5% area overhead cost.

3.2 Wavefront Splitting

This thesis builds on prior work on wavefront splitting. There have been several prior works that address branch divergence through wavefront splitting. In doing so, GPU cores can overlap branched path execution which would otherwise be serialized. This technique increases the opportunities for latency hiding and improves hardware utilization on the GPU.

GPU architectures keep track of branch serialization using a reconvergence stack. Upon encountering a conditional branch with diverging work-items, both paths are pushed onto the stack. Once the first path is completed, it is popped from the stack. The next PC is determined based on the top of the stack at this point.

3.2.1 Dynamic Warp Subdivision

The first work to propose wavefront splitting as a method to reduce branch divergence is Dynamic Warp Subdivision (DWS) by Meng et al. [26]. This work models the behaviour of the GPU by presenting all SIMD execution hardware as a component called the warp processing unit (WPU). Meng et al. modify the behaviour of the WPU for branch and memory divergence; however, the focus here is on the former. When the WPU encounters branch divergence in a wavefront, it creates two subdivisions, known as splits, to correspond to each of the paths that the work-items may take. Each split is a separate schedulable entity equivalent to a wavefront. Hence, the WPU can choose to schedule any available wavefront or split. If a split encounters a stall, the remaining threads can still be scheduled to execute because they may be free of dependencies. The WPU can therefore execute the other split, allowing it to move ahead. This behaviour can provide a performance gain in

the case where all wavefronts would be stalled executing one branch path in the baseline. Furthermore, it can allow the split that runs ahead to get a head start on its own long-latency memory operations. Thus, splitting allows the WPU to benefit from greater latency hiding and potential memory level parallelism (MLP).

GPU performance benefits from latency hiding of splits during the divergent execution. When execution is no longer divergent, the splits can become detrimental to performance because they execute the same instructions out of lockstep. The break in lockstep execution results in a loss to thread level parallelism (TLP). Meng et al. refer to this phenomenon as unrelenting subdivision. To reduce the effects of this problem, they propose two techniques to reconverge multiple splits back to a full wavefront or to another split. First, DWS maintains a warp split table (WST) which stores entries containing the split work-item masks, current PCs, and parent wavefronts. Notably, if the wavefront is split, the branching paths are not pushed to the reconvergence stack to allow for arbitrary interleaved execution of either path. When all splits in the WST reach the PC at the top of the reconverge stack, they are reconverged to execute in SIMD lockstep once again.

One important caveat of DWS is that the reconvergence stack is frozen at the time of split to allow for interleaved execution of the splits. This means that there is some interleaving of the post-dominator of the branching paths which are chosen for the split. Since the post-dominator does not have divergent instructions, interleaving the splits' execution results in reduced SIMD utilization over the baseline. The second technique to reconverge is an optimization which expedites the reconvergence process to an extent. It is applied in the case that the PCs of the two splits align. This PC-based reconvergence reduces the interleaving of the post-dominator.

DWS allows for nested branches, meaning that it can support as many splits as there are work-items in a wavefront. In this scenario, each split executes a single work-item of the application. As expected, this level of splitting reduces the TLP and hurts the performance in cases where there are few or no opportunities for latency hiding. Meng et al. refer to this as oversubdivision. They therefore use a heuristic to limit the number of splits that are performed: split only when the post-dominator of the current branch is followed by a short basic block of fewer than fifty instructions. Overall, Meng et al. show that DWS with PC-based reconvergence gives a 1.13x speedup on average over the baseline architecture.

3.2.2 Dual Path Execution

DWS uses the WST to keep track of existing splits. When the WST is being used, the reconvergence stack is frozen. Any further nested branches can result in additional splits

for which post-dominators are not pushed to the reconvergence stack. Many of the opportunities for reconvergence in the DWS approach are missed. This behaviour results in lost TLP and as a result lost performance. To address this problem, Rhu and Erez propose the Dual-Path Execution (DPE) Model [35].

Instead of adding a data structure to keep track of splits, the DPE model modifies the reconvergence stack. In the baseline stack, each entry contains the PC of each basic block, the thread mask, and the post-dominator. The DPE stack keeps track of both branch paths in the same stack entry, with a shared post-dominator. Rhu and Erez also modify the wavefront scheduler and scoreboarding mechanism to allow two schedulable entities per wavefront. At any given point, DPE can select either of the two splits at the top of the DPE stack for execution. Reconvergence takes place when both splits reach the common post-dominator for the entry at the top of the stack. By having a mechanism to keep track of all post-dominators while allowing for splitting, this approach allows reconvergence to happen much sooner than with DWS. Rhu and Erez show that DPE provides an average 15% speedup over the baseline, while never degrading performance in the tested benchmarks. DWS on the other hand, exhibits some degraded performance in some workloads due to lower SIMD utilization.

There are some limitations to the DPE approach. The major notable aspect of DPE in comparison to DWS is that it does not support any selection of where wavefronts should be split. Instead, it creates splits on any conditional branch execution. This behaviour can lead to a similar problem of oversubdivision of wavefronts that DWS aimed to avoid. Additionally, as the name suggests, DPE only allows two splits to be interleaved at any point. This restriction limits the total number of schedulable entities which can be chosen for execution by the wavefront scheduler.

3.2.3 Simultaneous Branch and Warp Interweaving

DWS and DPE allow for interleaving execution of diverging work-items of a wavefront. A logical extension of this interleaving is to instead allow for simultaneous parallel execution through additional GPU hardware modifications. Brunie et al. propose such parallel execution with Simultaneous Branch Interweaving (SBI) and Simultaneous Warp Interweaving (SWI) [5]. These techniques allow for parallel execution between work-items within a wavefront for SBI and across wavefronts for SWI.

SBI allows work-items from the same wavefront to execute distinct instructions simultaneously. To accommodate this behaviour in hardware, Brunie et al. propose doubling the number of work-items in a wavefront while cutting the number of wavefronts in half.

Additionally, DWS, DPE, and SBI all share a limitation in handling branched execution with only a single path, exemplified by an `if` block without an `else` block. For these workloads, Brunie et al. propose SWI. SWI allows for parallel execution of splits from different source wavefronts as long as the active work-items do not overlap. This mechanism allows for greater parallelism and SIMD lane utilization. Overall, SBI combined with SWI provide a 40% performance improvement for irregular applications[1].

3.2.4 Subwarp Interleaving

A more recent work by Damani et al. explores the hardware requirements for implementing wavefront splitting, which they refer to as subwarp interleaving (SI), on NVIDIA hardware [8] for raytracing applications. Raytracing applications require each thread to compute the path of a ray of light that reflects off of objects in three-dimensional space. The nature of this computation means that these applications have a significant amount of branch divergence. Therefore, breaking up the lockstep execution as done with wavefront splitting provides some promise of improved performance.

SI proposes splitting with a fundamentally different mechanism than all prior works. While it still conceptually splits wavefronts as done in prior work, SI performs scheduling at the work-item level, in addition to the wavefront level. The status of each work-item is tracked using a finite state machine and the status information is kept in a thread status table (TST). All work-items which are ready constitute a split. The wavefront scheduler selects the wavefront which should execute based on if any of its work-items are ready for execution. As a further optimization, SI allows for splits to yield execution to another split. This functionality allows for splits to coordinate their context-switching behaviour to optimize their MLP.

The SI approach is fundamentally different from prior works because each split is confined to the wavefront slot of the parent wavefront, rather than being treated as a separate schedulable entity. Overall, Damani et al. demonstrate an average 6.3% performance improvement over the baseline with the best proposed configuration.

To summarize, the past works in wavefront splitting provide several varying implementations. The key design features from each are compared in Table 3.1.

[1]Irregular applications are defined as those with an average instruction count per cycle (IPC) of less than 30.

Table 3.1: Comparison of key design features in prior wavefront splitting works

	DWS	DPE	SBI	SWI	SI
Splits as separate schedulable entities	✓	✓	✓	✓	
Parallel execution of splits			✓	✓	
Parallel execution across different wavefronts				✓	
Reconvergence at some post-dominators	✓	✓	✓	✓	
Reconvergence at any post-dominator		✓	✓	✓	
Two-level scheduling					✓

3.3 GPUs in Real-Time Applications

The last set of related works aim to understand GPUs as platforms for real-time systems. Section 3.3.1 looks at some prior work investigating GPUs in safety-critical systems. Next, Section 3.3.2 presents the key ideas of hybrid analysis used to estimated the worst-case execution time of GPU programs. These ideas are vital to the analysis in Chapter 5.

3.3.1 Case Studies

A few recent case studies have looked at the use of GPUs in safety-critical systems, such as avionics. Kosmidis et al. present the findings of GPU for Space (GPU4S), a European Space Agency funded study of GPU use in space workloads [20]. They find that existing open-source GPU solutions are incomplete or model outdated GPU hardware, but that the RISC-V movement shows promise for the GPU domain. Out of current commercial off-the-shelf platforms, they indicate that AMD embedded products are preferred due to their better programming and open-source support. Finally, they indicate that techniques for reliability applied in the automotive sector can be applied to space workloads as well. Benito et al. note that existing programming models, such as CUDA and OpenCL, are limited for safety-critical workloads and seek to study the feasibility of OpenGL SC and Brook Auto, alternative GPU software models defined with safety-criticality in mind [3]. They find that the two alternatives provide sufficient performance without significant development overhead costs.

3.3.2 Estimating GPU WCET using Hybrid Analysis

Analyzing the worst-case execution time (WCET) of GPUs is subject to many architectural complexities and implementation details, making a full system model massive in scope. Betts et al. [4] use a mix of modelling and analysis with some empirical measurement-based values to compute a worst-case kernel execution time. The analysis starts with the construction of a control-flow graph (CFG) from the GPU program. The WCET of a CFG can typically be solved with the implicit path enumeration technique (IPET), as described in Section 2.1.

The GPU hybrid analysis approach determines the individual CFG edge latencies using empirical measurements. Instrumentation points (IPTs) are inserted at various points in the program to be able to measure the latency of all possible paths through the CFG. The CFG with IPTs can be converted into an instrumentation point graph (IPG). The vertices of the graph are the IPTs and the edges are the paths between IPTs that do not have another IPT along the path. The challenge with applying IPET to GPUs is that branches are serialized. To that end, Betts et al. proposes an algorithm for adding edges to the IPG to represent this serialized behaviour.

After running a series of experiments, the IPTs must be extracted. Betts et al. show that the IPTs can be separated by wavefront into a wavefront-specific trace. Furthermore, a series of test vectors across multiple inputs and experiments are used to get a more accurate measurement of the worst case for each kernel segment. Using this data, a solver can determine the WCET for a single wavefront.

Finally, there is some initialization delay between the starting IPT of each wavefront. In the worst case, the wavefront under analysis must wait for all other wavefronts to execute their starting IPT. This work aims to quantify this behaviour using two separate approaches, called dynamic and hybrid. The dynamic approach adds the time between the start of GPU execution to the first IPT of the final wavefront to get the total WCET estimate. The hybrid approach attempts to determine the workgroup groupings of wavefronts and use this information. It provides a more pessimistic bound, but includes the workgroup organization as a parameter.

Chapter 4

Implementation

Chapter 1 outlined two key contributions: to enable wavefront splitting for safety-critical systems and to enable its hybrid analysis. The purpose of this chapter is to address the former; it outlines the architectural changes made to the GPU to accommodate wavefront splitting for safety-critical systems and to enable subsequent analysis, with some ideas developed from the prior works described in Chapter 3.

4.1 Limitations of Prior Work

The prior wavefront splitting works all provide a solid foundation for targeting and addressing branch divergence. However, there was no analysis of the worst-case execution time in those works. Hence, there are two key limitations of prior wavefront splitting implementations which can result in an overly pessimistic worst-case upper bound.

1. **SIMD vector pipelines are shared**. When splits are created, they function as separate schedulable units that compete with every other split. This causes contention for scheduling resources. In the worst case, a split must wait for all other wavefronts, including other splits, to complete. As a result, the analysis must assume that all split execution is serialized, which corresponds to a bound that increases roughly linearly with the number of splits. Therefore, there is a need to create separate resources for each split to enable a guarantee of parallel execution.

2. **There is no static way to tell which branches split**. In the absence of information about where wavefronts will split, the worst-case analysis must assume that the

wavefront splits at all branches. Additionally, there may be some hardware overhead constraints for splits. If splitting is performed opportunistically, then the analysis must assume that the splitting takes place in program order until hardware resources are fully utilized. This approach may forgo some of the benefit of splitting at later branches in both the average and worst case. Therefore, the implementation needs to allow the GPU programmer to statically mark at which branches the wavefront should split.

Before getting to how these limitations will be addressed, it is first necessary to outline the fundamental architectural modifications to support splitting and the mechanisms by which splits are created and execute.

4.2 Architectural Support for Splitting

Wavefront splitting requires several hardware modifications to the AMD GPU architecture. This section presents some hardware additions to the SIMD unit to realistically enable wavefront splitting, without yet fully addressing the limitations in Section 4.1. It concludes with some definitions which will be used in later sections to refer to wavefronts with different properties related to splitting.

4.2.1 Additional Wavefront Slots

The first component under analysis for implementing wavefront splitting is the SIMD unit. Recall from Chapter 2 that each CU consists of a set of four SIMD units. Each SIMD unit contains a vector ALU, a vector register file, and ten slots for storing wavefront context. This context includes general purpose registers, the program counter, and the execution mask. The process of splitting a wavefront requires maintaining two separate contexts for the resulting splits. To minimize the additional hardware overhead from splitting, the split wavefronts are identical to the base wavefronts, with some additional state elements.

Vector ALUs must support partial execution of wavefronts or smaller wavefront sizes and the SIMD must maintain the context of every in-flight split. Every split in the implementation is treated as a fully schedulable unit equivalent to a wavefront. The baseline SIMD configuration is capable of maintaining multiple split contexts with wavefront slots. However, splits occupying existing wavefront slots limits the number of wavefronts which can be inflight on the SIMD. If every wavefront is allowed and expected to split $\mathcal{S}$ times

during program execution, then the number of wavefronts supported by the hardware is cut by a factor of $\mathcal{S}+1$. This trade-off is unacceptable because it hurts performance of the GPU, which is counter-productive to the goal of wavefront splitting. Instead, the SIMD unit must have an increased number of wavefront slots for splits.

Wavefront splitting requires additional SIMD slots to maintain context and program state without adding scheduling contention. To that end, additional wavefront slots are provisioned for each SIMD which are held in reserve for splits. Since the additional wavefront slots are reserved for splits, wavefronts that have not been split cannot use the additional slots. When a wavefront is split, one of the splits can reside in the original wavefront slot. The new split must occupy one of the reserve slots. Additionally, hardware limitations dictate how many concurrent splits are supported. All wavefronts are allowed to split the same number of times; the number of such splits is denoted $\mathcal{S}$, which can take any integral value in the range $[0, 63]$. Therefore, the design supports $\mathcal{S}$ additional sets of reserve wavefront slots, as shown in green in the SIMD diagram in Figure 4.1. This allows the GPU designer flexibility in determining how much additional area overhead is acceptable as a trade-off for increasing the splitting capacity.

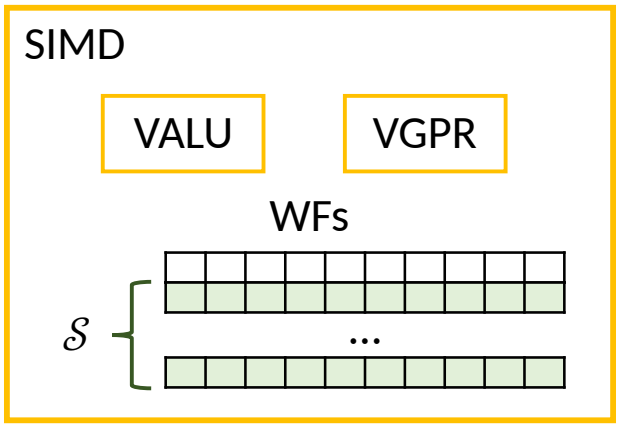


Figure 4.1: SIMD unit with additional wavefront slots for splitting

In addition to extra wavefront slots, each slot must keep track of new state elements. The first new state element is a read-only bit to indicate whether a slot is reserved for splitting or not. The second element is a "split mask", which is used to supplement the execution mask. The split mask is necessary because it keeps track of how the wavefront is split. These new state elements allow the existing wavefront slots to behave as splits when required.

4.2.2 Split Masks

The baseline AMD architecture includes an execution mask for each wavefront. The mask features one bit per wavefront lane that can be modified by instructions in the kernel. As mentioned in Section 2.2.5, a lane is masked when its corresponding bit is set to zero. Conversely, the lane is unmasked when its corresponding bit is set to one. For any given instruction, all unmasked lanes execute the instruction or operation, while masked ones do not execute, executed as a NOP instruction.

The execution mask is sufficient in the baseline architecture. However, when splitting, some work-items in the resulting wavefronts are masked for the duration of the split's existence. At the same time, each split may encounter further branch divergence where splitting is not possible (e.g. not enough resources for splitting). Hence, the splitting mechanism cannot simply rely on the existing execution mask. Thus, the split mask is introduced to serve as a representation of the active work-items in the split. This mechanism allows for a full-size wavefront of 64 work-items to be used for any smaller wavefront size to accommodate splitting. Each lane only executes if both the corresponding split mask and execution mask bits are set.

4.2.3 Definitions

Having introduced the concepts of reserve wavefront slots and split masks, it is possible to define some descriptions for wavefronts that will be useful when outlining the implementation and analysis. The purpose of this section is to define these terms.

- A *stopped* wavefront is not permitted to execute instructions. It may only start execution when the CU places it in the running state.
- A *running* wavefront is executing or waiting to execute instructions and is available to be chosen by the SIMD for execution.
- A *full* wavefront refers to a wavefront which has not been split, which corresponds to a fully unmasked split mask.
- A *split* wavefront refers to a wavefront which has been split; thus, it has a homogeneous split mask. Alternatively, a split wavefront can be referred to in abbreviated form as a *split*.
- A *reserve* wavefront refers to a wavefront which is not launched at the start of the

kernel but is instead held in reserve as the target for a split. A reserve wavefront cannot be a full wavefront and it begins kernel execution in the stopped state.

- A *launch* wavefront refers to one of the wavefronts which begin execution as a full wavefront, rather than a reserve wavefront. If a launch wavefront is split, it is still considered a launch wavefront. A launch wavefront begins kernel execution in the running state.
- A wavefront is a *parent* wavefront for a split wavefront if and only if the split wavefront began execution as a result of a split operation on the parent. The wavefront that triggers the split is its own parent. A parent wavefront may be a launch or split wavefront, or both.
- Similarly, a split wavefront is its parent wavefront's *child* wavefront.
- When multiple nested splits occur, there is a parent-child hierarchy created. All wavefronts which share the same launch wavefront at the root of the hierarchy are called *siblings*.

4.3 Splitting Mechanism

There are several key design decisions that guide the process of deciding under which conditions to split a wavefront. With the architectural support in place, this section presents the mechanism to perform wavefront splitting. There are several key conditions which must be met for wavefront splitting to occur. Subsequently, there are some important considerations for how the resulting splits are prepared for execution.

4.3.1 Conditions for Splitting

Firstly, and perhaps most fundamentally, the program counter for the wavefront under analysis must point to a conditional branch instruction. In the GCN3 ISA, there are several of these conditional branch instructions; however, the most relevant to the problem of branch divergence is the pair of instructions which check the wavefront execution mask:

- `s_cbranch_execz` and
- `s_cbranch_execnz`.

These instructions are an optimization designed to skip a basic block if all work-items are masked. If there is no branch divergence, one of the branching paths can be skipped. The key idea with wavefront splitting is to leverage this optimization to enable one wavefront split to skip ahead and continue with the other branching path. When the SIMD executes one of these two conditional branch instructions for the wavefront, it may have a candidate for splitting.

Splitting on any suitable conditional branch instruction may not always be the best choice, particularly if the execution mask for the wavefront is homogeneous. When there is a homogeneous execution mask, there is no branch divergence and, hence, no reason to split. By contrast, a heterogeneous execution mask indicates that there is branch divergence and that splitting may be helpful. Therefore, if the SIMD encounters a conditional branch instruction but the execution mask is homogeneous, it can and should skip the split operation.

4.3.2 Creation of a Split

Wavefront splitting is implemented by adjusting the semantics of the relevant `cbranch` instructions. Splitting must take place before the branching is performed to properly leverage the branching behaviour and skip the desired basic blocks. Therefore, splitting takes place at the start of the cycle when the instruction is at the execute stage of the pipeline.

When splitting occurs, there are several steps that must be taken, which are illustrated in Figure 4.2. First, the SIMD chooses an available wavefront slot in reserve. This information is available statically, so if the programmer has set the mode register bit for splitting, then it is assumed that there are always available reserve wavefront slots. Otherwise, the hardware will choose the first $\mathcal{S}$ branches in program execution order. Along with finding the wavefront slot, the SIMD reserves vector and scalar register resources for the new slot and links it with the local data share for the CU. Additionally, the new wavefront slot is linked with the same barrier resources of the original wavefront slot.

Next, the SIMD sets the split masks for the original and new wavefront slots. The original slot split mask is set to the current execution mask of the wavefront. The new slot split mask is set as the negation of the current execution mask; in other words, all other lanes that are masked by the execution mask. Finally, the new slot is ready to enter the SIMD execution pipeline, which happens when the SIMD sets the status of the new slot to "running". In effect, as shown in Figure 4.2, the parent wavefront executes the `if` path,

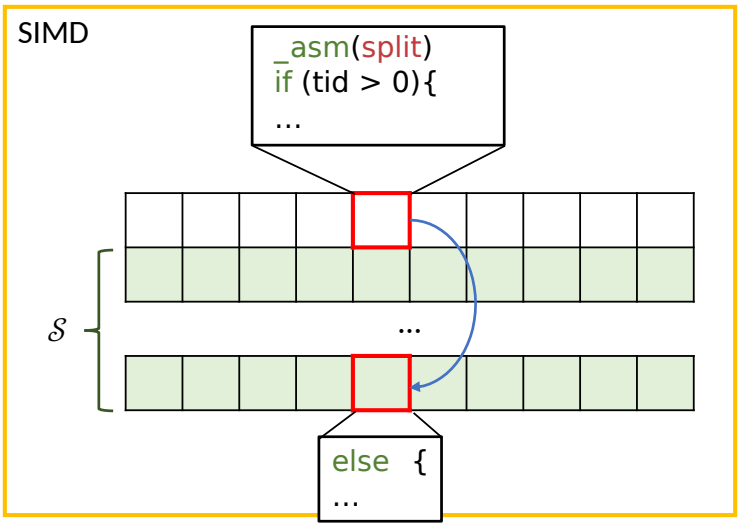


Figure 4.2: The splitting process where each split executes one of two branch paths

while the child wavefront executes the `else` path. With all of these factors in place, the two splits are ready to be scheduled and executed by the SIMD pipeline.

4.3.3 Splitting Example

Consider as an example the very simple kernel in Figure 4.3. The HIP compiler will convert the code to GCN3 assembly code, shown in Figure 4.4, which has two conditional branch instructions. This code is a variation on the assembly code presented by Gutierrez et al. [15]. Table 4.1 depicts the baseline execution masks at these two instructions, as well as the split masks for each of the resulting wavefront splits.

```
1        if (threadIdx < 3) {
2            // ...
3        } else {
4            // ...
5        }
```

Figure 4.3: Simple HIP kernel with branch divergence

```
1       BB8:
2             s_cmp_le vcc, 4, v0
3             s_and_saveexec s[2:3], vcc
4             s_cbranch_execz BB10
5       BB9:  # ...
6       BB10:
7             s_andn2 exec, s[2:3], exec
8             s_cbranch_execz BB12
9       BB11: # ...
10      BB12:
11            s_endpgm
```

Figure 4.4: Compiled GCN3 assembly code from Figure 4.3

The first basic block, `BB8`, performs the vectorized comparison in the `if` statement and stores this value in the execution mask. Basic block `BB9` contains code in the `if` body of the HIP code. Basic block `BB10` flips the execution mask and `BB11` contains code in the `else` body. Finally, basic block `BB12` ends the program. Notice that the instruction on Line 4 branches to `BB10` if the execution mask is all zeros. Similarly, the instruction on Line 9 branches to `BB12` if the execution mask is all zeros. However, since the execution mask is heterogeneous, these branches do not occur in the absence of splitting.

When splitting is introduced into the example in Figure 4.4, two splits are created at Line 4: Split 1 and Split 2. The split mask for Split 1 is set to be the execution mask and the split mask for Split 2 is set to the negation of the execution mask, as shown in Table 4.1. Subsequently, the execution mask for each split is set using the bitwise AND of the original execution mask and the split mask. Let

- e_b refer to the baseline execution mask determined by the program execution,
- e_{si} refer to the execution mask of split i, and
- s_i refer to the split mask for split i.

Taking Split 2 at Line 4, for instance, the resulting execution mask is derived as follows

$$
\begin{aligned}
e_{si} &= e_b \wedge s_i \\
&= 0111 \wedge 1000 \\
&= 0000
\end{aligned}
$$

This result produces an execution mask for Split 2 that is all zeros, meaning that the split can skip execution of `BB9`. Similarly, at Line 9 of Split 1, the execution mask is all zeros, allowing the split to skip `BB11`. All of the combinations of split masks and execution masks are shown in Table 4.1. By skipping the blocks, the two splits are able to independently execute the two branching paths and benefit from additional potential latency hiding beyond the baseline.

Table 4.1: Split and execution masks for branch instructions in Figure 4.4

	`s_cbranch_execz BB10`	`s_cbranch_execz B12`
e_b	0111	1000
s_1	0111	0111
e_{s1}	$0111 \wedge 0111 = 0111$	$1000 \wedge 0111 = 0000$
s_2	1000	1000
e_{s2}	$0111 \wedge 1000 = 0000$	$1000 \wedge 1000 = 1000$

4.3.4 Reconvergence

Having dealt with splitting at divergent points in kernel execution, some discussion of reconvergence is required. Reconvergence from a program analysis standpoint refers to the meeting of two divergent paths. For instance, in Figure 2.7, there is an example of branch divergence at $BB2$, with paths to $BB3$ and $BB4$. These paths reconverge at $BB5$. As discussed in Chapter 3, prior works in wavefront splitting have looked at different strategies for reconvergence. Out of the proposed methods, dual-path execution (DPE) provides the most robust solution to this problem with the best performance results [35]. In that solution, both divergent paths are maintained in the reconvergence stack. This allows the merging of splits to take place at any reconvergence point in the program, including for nested branches.

The reconvergence stack is used to detect the PC at which reconvergence occurs. Instead of maintaining reconvergence PCs for both of the paths as in DPE, this implementation adds a split mask stack for each wavefront. At the start of kernel execution, the stack simply contains one fully unmasked entry. Every time a split occurs, the old split mask is pushed onto the stack. Similar to the derivation of the execution mask of the split wavefront, these split masks can be used to detect reconvergence. Every time the base execution mask changes, the GPU hardware will check whether reconvergence has been achieved using the split mask stack. If the base execution mask (before applying the exclusive-OR with the split mask) matches an entry in the split mask stack, then a point of reconvergence has been reached. It is notable that multiple reconvergence points can occur at the same PC; therefore, it is not enough to solely consider the split mask at the top of the stack.

Although reconvergence prevents unrelenting subdivision [26], there are benefits for worst-case analysis to avoiding reconvergence. The analysis, presented in Chapter 5, examines the effect of each configuration of split wavefronts. Furthermore, the first split to arrive at a reconvergence PC must wait for its sibling to reconverge. In the worst case, it must wait for the sibling to execute all divergent instructions. Therefore, reconvergence is left unimplemented in favour of a simpler worst-case analysis. In the future, it may be useful to expand the implementation and analysis to support reconvergence.

4.4 Parallel Execution of Splits

The components that have been covered up to this point have enabled wavefront splitting with splits as separate schedulable entities equivalent to regular wavefronts. The creation of splits allows the GPU programmer to context-switch to a different wavefront split when encountering load-use stalls. In those situations, the GPU utilization is improved. However, there is still an opportunity for further improvement to utilization. As suggested by Brunie et al. [5], the SBI and SWI techniques propose executing wavefront splits simultaneously. This section will describe the method of implementation of a similar parallel execution model. However, as stated in Section 4.1, if the vector pipeline resources are not duplicated, then split execution is serialized in the worst case. Thus, the vector pipeline resources will be duplicated to *guarantee* parallel execution of splits.

4.4.1 Hardware Organization

As mentioned in Chapter 2, the GPU hardware executes based on the SIMD pattern. Therefore, in order to support wavefront splits executing simultaneously in parallel, the SIMD execution must be broken up. To accomplish this, the SIMD is expanded with new vector pipelines. With unlimited wavefront splitting, a wavefront can split into 64 distinct splits, each with one active work-item. However, as discussed in Section 4.2.1, the finite number of wavefront slots limits this to $\mathcal{S}$ splits per wavefront. Therefore, the vector ALU only must support $\mathcal{S}$ distinct instructions in the same cycle to enable parallel execution. This guarantees the parallel execution of all ready splits corresponding to the launch wavefront.

Additionally, wavefronts execute scalar instructions on the CU scalar pipeline. If the resource is not expanded, splits will compete for it. Therefore, each CU is expanded with $\mathcal{S}$ scalar pipelines.

4.4.2 Wavefront Scheduling

With a SIMD unit that can execute multiple splits with distinct program counters in parallel, the next step is to enable scheduling of multiple splits. This requires some modifications to the scheduling algorithms used at various points in the SIMD execution pipeline. This pipeline is illustrated in Figure 4.5.

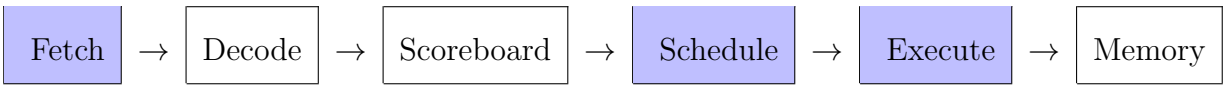


Figure 4.5: SIMD execution pipeline with modified stages highlighted

The first stage in the pipeline is the fetch stage, which consists of a fetch unit for each SIMD unit. Each cycle, the fetch unit attempts to fill an instruction buffer (IB) for a wavefront slot, which is chosen using a scheduling policy, either oldest-first or round-robin. If successful, it adds the instruction to the scheduling list. When dealing with splits, the scheduling policy should select as many splits as possible, as long as the split masks do not overlap, as in SWI. Hence, multiple fetch operations can happen in parallel.

After the fetch stage, the instructions in the IB are decoded in the decode stage and evaluated in the scoreboard stage. These stages do not require any modifications to enable parallel execution of splits. This is because they perform their respective operations on all wavefront slots with at least one entry in their respective IBs. The scoreboard stage

is responsible for reading the operands of the instruction at the front of the buffer and determining if there are any read-after-write (RAW) dependencies. If not, the instruction is marked a ready to execute.

The schedule stage performs scheduling of wavefronts with ready instructions to available functional units. This includes the are four SIMD units with $\mathcal{S}+1$ vector pipelines each, $\mathcal{S}+1$ scalar pipelines, and one memory unit each for the global memory, scalar memory, and local data share. Again, the scheduling policy used is either oldest-first or round-robin, where a single wavefront is selected. In order to enable parallel split execution, the scheduling policy chooses as many splits as possible such that the split masks do not overlap, the same as with the fetch stage. In fact, an identical scheduling policy is used to select splits in the fetch and schedule stages. Once this is done, the execute stage can execute as many splits as are scheduled rather than just one, enabled by the SIMD hardware modifications detailed in Section 4.4.1.

The focus of the simultaneous parallel execution implementation is on optimizing the ALU side of operations. The memory bandwidth remains unchanged in this implementation. As a result, for memory operations, while address computation can be done in parallel for different splits, the actual memory accesses may be serialized.

4.5 Splitting Statically

As mentioned in Chapter 3, prior work dynamically split the wavefronts when branch divergence is encountered. In the worst case, branch divergence causes each work-item to execute a different path in the program. Hence, every work-item in the wavefront is serialized and, as a result of dynamic splitting, every split consists of only one work-item. With a finite number of wavefront slots in reserve for splitting, some wavefronts may be prevented from splitting. Therefore, the WCET is difficult to model.

In addition to difficulties with the WCET analysis, opportunistic dynamic splitting can be detrimental to performance. Some divergent paths suffer more from serialized execution than others. For example, consider the CFG with nested divergence from Figure 2.7. An opportunistic splitting mechanism would split on the outer branch first because it is encountered first. However, the inner branch may have a long latency memory operation followed by a dependent ALU operation. This pattern can receive a large benefit from splitting compared to simpler instances of branch divergence. If the SIMD unit is only capable of one split per wavefront, then splitting on the outer branch limits the additional latency hiding benefit that splitting on the inner branch would bring. Thus, being able to

choose where to split is important for worst-case analysis, but may also benefit performance. To that end, deciding which branches trigger a wavefront split should be a static choice.

There may be one or more optimal decisions for where wavefronts should split to minimize the worst-case upper bound. This is an interesting problem, but it is out of the scope of this thesis.

4.5.1 Hardware Support

To enable static assignment of branches which should trigger a split, there is a need for a corresponding compile-time programming capability. There are two potential methods to achieve this, both of which modify the instruction set architecture (ISA). This modification could be

1. a new or modified conditional branch instruction that splits the wavefront along the divergence boundary, or
2. a new register or bits of an existing register which can be set to indicate that the SIMD should split the wavefront upon branch divergence.

There are some trade-offs with respect to each choice. Choosing to add a new conditional branch instruction means modifying the ISA. Reusing and setting existing register bits does not require an ISA modification, but adds overhead when setting and resetting the bit. For this implementation, the latter was chosen. Fortunately, such an option exists with modern AMD GPU ISAs. The mode register, a read and write status-type register in the GCN3 and RDNA2 architectures, contains some unused bits. Table 4.2 depicts the used upper bits for both architectures. The least significant bits (LSBs) are omitted because all of them are used for other purposes. The unmarked bits are all unused.

Table 4.2: Used mode register bits (bits 31-16)

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
GCN	•	•	•	•	•									•	•	•
RDNA				•				•				•	•	•	•	•

Since the implementation is on GCN hardware, any of the unused bits of the GCN mode register would be suitable. For the purpose of consistency with newer architectures, the suitable bits are narrowed down to the five shared between GCN and RDNA. For this implementation, bit 21 was selected as the wavefront splitting control bit.

4.5.2 Software Support

The mode register bit must be set to mark a branch for splitting. To do so, a simple set register instruction can be used before and after the branch. The instruction can be inserted into HIP code with the `__asm` keyword. The register field should be set before the branch and and reset when splitting is no longer desired. This pattern is shown in Figure 4.6.

```
__asm("s_setreg_imm32_b32 hwreg(HW_REG_MODE, 21, 1), 1");
if (threadIdx < 3) {
    // Basic Block 1
} else {
    // Basic Block 2
}
__asm("s_setreg_imm32_b32 hwreg(HW_REG_MODE, 21, 1), 0");
```

Figure 4.6: Sample HIP code with mode register assignment for splitting

If the wavefront is executing a conditional branch instruction with a heterogeneous execution mask, the SIMD must check bit 21 of the mode register for the wavefront under analysis. Additionally, the SIMD should have sufficient resources for splitting. Namely, it should have an available wavefront split in reserve with sufficient register file resources. For the purpose of simplifying the WCET analysis, the mode register bit should be set carefully. This is to avoid encountering a situation in which the programmer desires a split, but there are insufficient resources to actually perform the split. With these conditions met, the SIMD may proceed with splitting the wavefront.

Chapter 5

Worst-Case Analysis

Chapter 4 detailed the wavefront splitting implementation on AMD GPU hardware. The purpose of this chapter is to enable hybrid analysis for GPU programs under splitting. Prior work in GPU hybrid analysis [4] does not model the WCET when splitting. Simply blindly applying the hybrid analysis models from prior work results in an overly pessimistic estimate. Hence, this chapter introduces some optimizations to the model to better support splitting.

This chapter will begin in Section 5.1 by discussing the scenario that elicits the worst-case behaviour, known as the critical instance. Next, Section 5.2 will outline the process of determining the WCET estimate for a single wavefront. Finally, Section 5.3 will detail how the analysis is aggregated for all wavefronts executing the kernel to get an overall execution time estimate.

5.1 Critical Instance

The worst-case behaviour of any program happens under a certain set of conditions. These conditions typically involve a maximum amount of contention for resources and scheduling interference. The scenario that elicits the worst-case behaviour is known as the critical instance. In the case of wavefront splitting on GPU hardware, the critical instance is defined by two conditions.

1. **All launch wavefront slots, $\mathcal{W}$, are utilized.** To ensure maximum scheduling interference between wavefronts, the wavefront slot hardware across all SIMDs must be fully utilized.

(a) For some workloads, fully utilizing the hardware may not be possible. If the number of wavefronts required by the kernel, $\mathcal{W}_{kernel}$ is less than the maximum amount of wavefronts available in hardware to execute the kernel, $\mathcal{W}_{HW}$, then the critical instance is limited. Hence,

$$\mathcal{W} = \min\left(\mathcal{W}_{HW}, \mathcal{W}_{kernel}\right)$$

(b) If $\mathcal{W} = \mathcal{W}_{HW}$, the number of work-items per workgroup specified by the programmer, T_{WG}, is important to utilize as many of the available wavefronts as possible. Hence,

$$(T_{WF} \times \mathcal{W}) \mod T_{WG} = 0$$

where T_{WF} is the number of work-items per wavefront. Take Figure 5.1 as an example (where $G = 5$ is the number of workgroups). The choice of T_{WG} is made such that the total number of available work-items on the hardware, $T_{WF} \times \mathcal{W}$ divides evenly into workgroups. As a result, all hardware wavefront slots are provisioned.

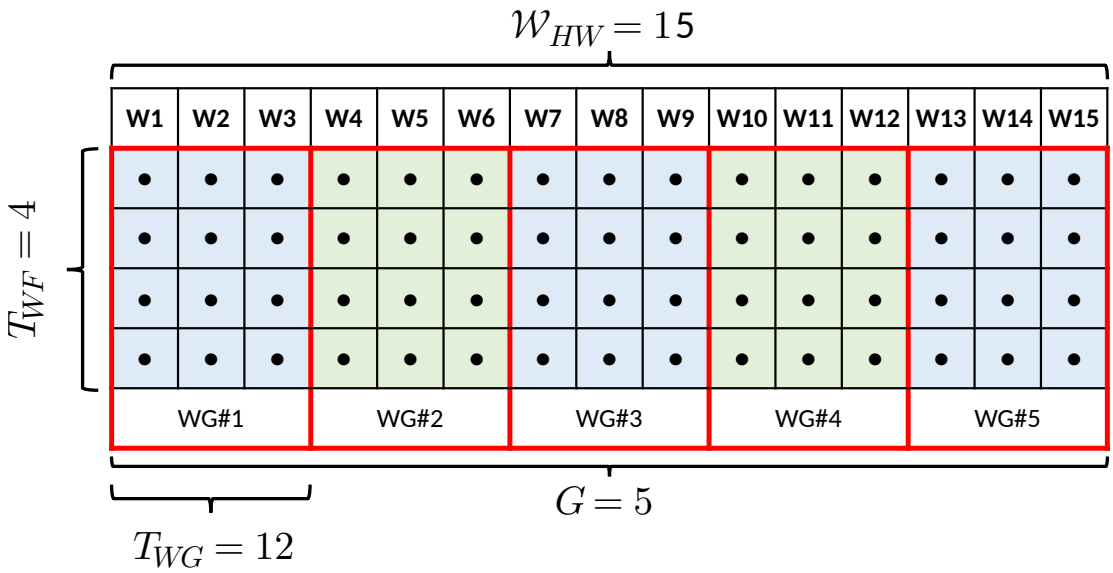


Figure 5.1: Workgroup dispatch maximizing use of available wavefront slots

2. **Each launch wavefront splits $\mathcal{S}$ times.** In order to ensure maximum interference between splits, the kernel should generate as many splits as possible. Hence, each

launch wavefront should split into each of its reserve $\mathcal{S}$ wavefront slots. To accomplish this, the kernel should have at least $\mathcal{S}$ branching paths when the mode register bit is set. This will be examined in more detail and proven in Section 5.2.5.

5.2 Modeling Individual Wavefronts

To model the WCET of a GPU kernel executing on GPU hardware, it is first useful to determine the WCET of a GPU kernel executed by a single wavefront, L_w. To do so, hybrid analysis uses empirical timing measurements of small segments within the CFG and stitches them together to get a WCET estimate. This section describes the process of extracting the CFG of the kernel, strategically inserting IPTs for measurement, and combining the empirical information to compute a WCET estimate. These methods are all well established in prior literature; however, there are some implementation-specific considerations which are discussed here.

5.2.1 Extracting CFGs from GPU Kernels

The first step in the hybrid analysis process is to extract a CFG from the GPU kernel. Extracting the CFG involves processing the GPU kernel to determine these basic blocks and paths between them. Typically, a compiler forms basic blocks dynamically as it builds an intermediate representation [13]. The HIP compiler, `hipcc`, performs this basic block formation when generating GCN assembly code. It gives users an option to view the GCN assembly code that it generates via the `-save-temps` flag. The code uses numbered basic blocks as labels which can be targets of branch instructions, similar to what is shown in Figure 4.4. Therefore, formation of basic blocks does not need to be performed again.

In order to extract control-flow information, a Python script extracts labels from the assembly code and infers the possible control-flow paths of the kernel. The script uses branch instructions and labeled basic blocks in the program to infer the edges in the CFG. The script steps through each line in the GCN assembly program and performs one of three actions.

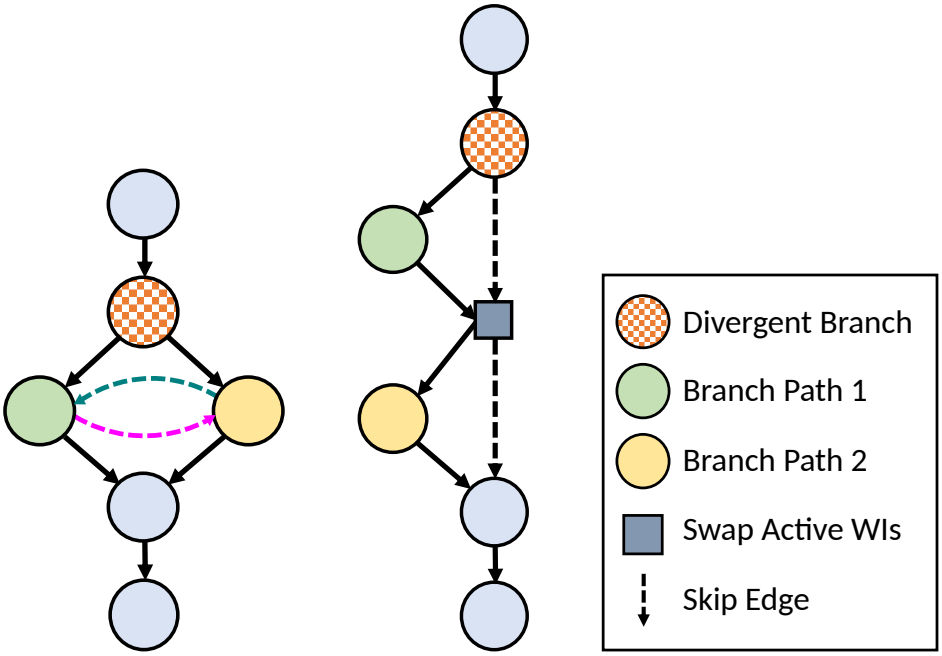
1. If the script encounters a conditional branch instruction, there are two possible outcomes: either control proceeds to the next PC or it jumps to the basic block indicated by the label of the branch. Therefore, the script infers two edges in the CFG.

2. If the script encounters a new basic block (indicated by a label), it connects the previous basic block to the new one with an edge.

3. If the script encounters an unconditional branch, it infers a single edge to the basic block destination of the branch. This type of branch is present when a kernel contains a loop.

5.2.2 Representing Branch Divergence

Traditional IPET problems do not account for GPU branch serialization during branch divergence. Consider an example branch divergence pattern, highlighted in Figure 5.2a.



(a) CFG with added edges

(b) CFG in GCN code organization

Figure 5.2: Branch divergence in a traditional CFG compared to GCN

Ignoring the dashed line edges, execution branches to either the green or yellow blocks, but there is no way to represent the GPU executing both. To accommodate this behaviour, Betts et al. propose an algorithm which inserts the dashed line edges into the CFG. These

edges originate from any basic block that immediately precedes a post-dominator of a branch. The destination of the edges are the first branching basic block of any other branching path. These edges correspond to additional constraints in the IPET problem formulation which allows for a model of branch divergence and serialization.

Fortunately, the algorithm to insert branch serialization edges is unnecessary for GCN code. The HIP compiler encodes the serialization through the organization of its instructions and basic blocks. Consider again the example control-flow divergence pattern, highlighted in Figure 5.2. Compiler literature [13] and Betts et al. [4] present CFGs with a structure similar to Figure 5.2a. As a result, the dashed edges must be added in a post-processing step to properly model serialized branch execution.

GCN, on the other hand, places branching paths on the same execution path, shown in Figure 5.2b. This structure arises corresponding to the assembly code structure shown in Figure 4.4. At the divergent branch, a wavefront with any active work-items will proceed to the first branch path (in green). If the wavefront is fully masked, then the `cbranch` instruction allows the wavefront to skip the green basic block. At the navy square, the wavefront swaps its active work-items by inverting the execution mask relative to the reconvergence stack. Then, the process repeats for the second branch path (in yellow). The green and yellow basic blocks, previously in alternate paths, can be visited in one pass without inserting additional edges to the CFG. Therefore, no modifications to the CFG are required.

5.2.3 Inserting Instrumentation Points

Hybrid analysis uses empirical measurements of execution times of small paths within the CFG. Gathering these measurements requires the insertion of probes into the program; these probes are referred to as instrumentation points (IPTs). Each probe should give information about the point in time when it is executed.

IPT probes can be inserted in software as some software function call or as a hardware probe. For the purpose of this thesis, since wavefront splitting is implemented in gem5, the probes are executed in hardware at specific points during program execution. The simulator outputs logging information at these key points in the kernel and can be pieced together for clear runtime WCET values. They are represented as IPT nodes added to the CFG between two basic blocks connected by an edge.

As an example, the diagram on the left of Figure 5.3 depicts a CFG with IPTs inserted. It is important to discuss the strategy of IPT insertion to get timing information about all paths in the CFG.

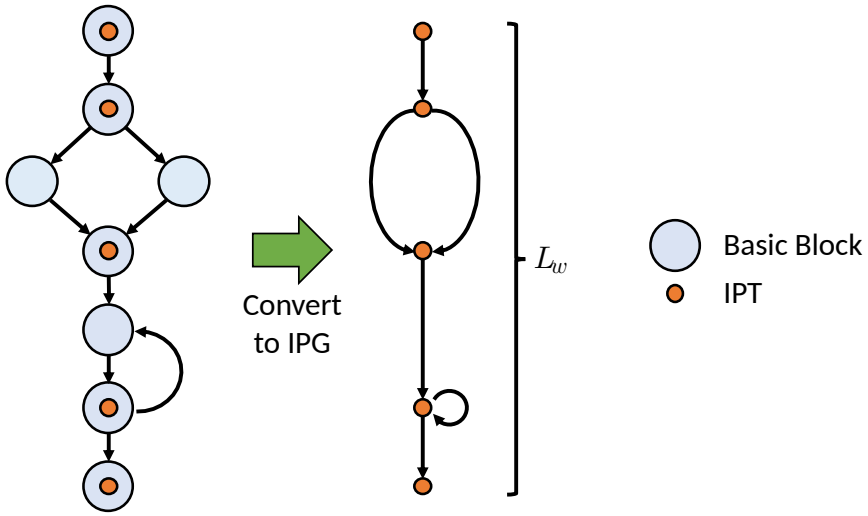


Figure 5.3: Conversion of a CFG with IPTs to an IPG

IPTs are inserted when a wavefront encounters

- the start of the kernel,
- a diverging branch,
- a loopback branch,
- reconvergence of two branching paths, and
- the end of the kernel.

The above IPTs may not be fine-grained enough to provide execution time measurements for each edge in the CFG. For example, there may be several basic blocks between the start of a kernel and a divergent branch. IPET requires that each edge WCET is known to determine the overall WCET. However, since the purpose of hybrid analysis is to stitch together the segments of the CFG, it is not essential to have such fine-grained measurements, as long as each path is represented. The only requirement is to transform the CFG. As shown by Betts et al., a control flow graph with IPTs can be converted to an IPT graph (IPG) of a wavefront. The nodes of the IPG correspond to the IPTs. Edges of the IPG correspond to all paths between IPTs that are free of other intermediate IPTs. When solving the IPET problem, the IPG can be used as a substitute for the CFG.

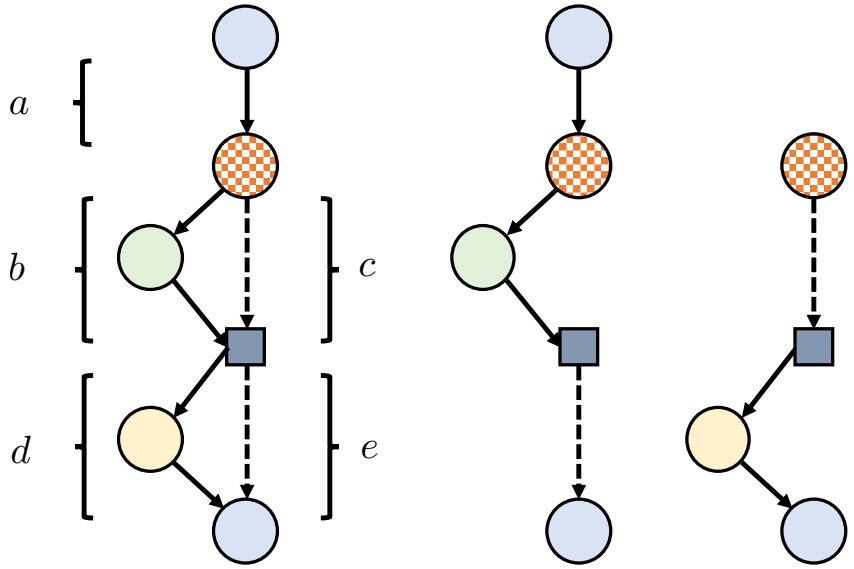
5.2.4 Incorporating Splitting

When putting the WCET measurements together across wavefronts, Betts et al. use a wavefront initialization delay which is scaled by the number of wavefronts. Intuitively, before doing any analysis across multiple wavefronts, it is clear that adding splits will increase the contribution of this initialization delay. Therefore, the worst-case estimate will be overestimated and will be worse than when not splitting. However, by reducing the worst-case overestimation while still maintaining model correctness, there is an opportunity to optimize the CFGs to tighten the worst-case bound.

There is an important consideration to be made to incorporate wavefront splitting when solving for the WCET of the CFG and IPG. A split is guaranteed to skip certain basic blocks whereas a full wavefront will serialize all branch execution in the worst case. When looking at groups of wavefronts, some may be launch wavefronts and some may not. Therefore, it is important to analyze the worst-case behaviour of all splitting configurations. The first observation is that some paths in the CFG and IPG can be omitted for splits. Since split wavefronts execute on a subset of lanes to match the branch divergence boundary, some paths are guaranteed to be skipped when the wavefront executes. These omissions give a clearer picture of WCETs for individual wavefronts.

Figure 5.4 illustrates a set of CFG nodes corresponding to the execution of a single `if-else` branch, with Figure 5.4a representing the execution in the absence of splitting. Due to the way a wavefront is subdivided into two splits, there are two distinct behaviours for wavefront splits. Splits are created when a wavefront encounters a divergent branch where the programmer has enabled splitting. Instead of allocating a new wavefront slot for each split, the GPU continues to use the parent wavefront for one of the splits. The GPU will still allocate a new slot for the child wavefront, which begins execution at the same branch instruction. This leads to the first IPG optimization: for a newly allocated split wavefront, all basic blocks prior to the branch instruction can be omitted from the CFG, as shown in Figure 5.4c. Corresponding IPTs can then be removed from the IPG.

In addition to the basic blocks leading up to a branch, the worst-case analysis can remove some paths corresponding to branches in the IPG. When a wavefront is split at a branch with divergence, the parent wavefront's split mask is updated to match its execution mask. This means that it will not skip the next basic block or set of basic blocks via the conditional branch. Next, the wavefront flips its execution mask. However, since the split mask prevents this from happening, the execution mask is fully masked. Hence, it skips execution of the second branch. To summarize, a wavefront that is split is guaranteed to continue to the first branch and skip the second branch. Similarly, its split is guaranteed to do the opposite. This optimization yields the simplified CFGs in Figures 5.4b and 5.4c.



(a) Without Splitting

(b) With Splitting, Parent Wavefront

(c) Without Splitting, Child Wavefront

Figure 5.4: CFG without splitting and CFGs with paths removed for splitting

Without splitting, the WCET is $L_w = a + b + d$. Since the dashed edges correspond to skipping the green or yellow basic blocks, then $c \leq b$ and $e \leq d$. Hence, the WCETs for the parent and child wavefronts are $L_w = a + b + e$ and $L_w = c + d$, respectively. Combined with information about how many splits are executed, this optimization provides a more accurate model for the worst case.

The CFG and IPG can also be modified for wavefronts that have a homogeneous execution mask. If an execution mask at a branch marked for splitting is guaranteed to be homogeneous, then the split does not need to happen. However, because the execution mask is homogeneous, one of the branch paths can be eliminated in the CFG, similar to one of the optimized CFGs in Figure 5.4. For example, a fully unmasked wavefront at the checkered node in Figure 5.4 will follow the CFG in Figure 5.4b while a fully masked wavefront will follow Figure 5.4c. This optimization reduces the behaviour of a wavefront to that of a split, for the subset of the CFG corresponding to the branch divergence.

5.2.5 Splitting in the Worst Case

Section 5.1 claims that the worst-case scenario is when all wavefronts split $\mathcal{S}$ times. It is not immediately obvious whether a wavefront that splits at a branch corresponds to the worst-case scenario. This is subject to proof, which is presented here. First, a preliminary claim is demonstrated by Lemma 5.2.1, which can be used by later assertions.

Lemma 5.2.1. *In the worst case, a wavefront will have a heterogeneous execution mask at all branches.*

Proof. To prove this statement, we can examine the two cases whether or not the branch is marked for splitting.

1. The branch is not marked for splitting. If the mask is homogeneous, only one of the branching paths is taken. If the mask is heterogeneous, both branch paths are taken in sequence. The sum of execution times for the two branch paths must be greater than or equal to either path individually. Therefore, the mask is heterogeneous in the worst case.

2. The branch is marked for splitting. If the mask is homogeneous, only one of the branching paths is taken. If the mask is heterogeneous, the wavefront splits and the total execution time is the maximum of the two branch paths. The maximum of the two branching paths must be greater than or equal to either path individually. Therefore, the mask is heterogeneous in the worst case.

Both cases above show that execution masks must be heterogeneous in the worst case. □

A GPU kernel may have multiple branches in sequence or in a nested configuration. Given the available resources (i.e. the number of reserve wavefront slots per wavefront, $\mathcal{S}$), the wavefront can only support a limited number of nested splits, which may be fewer than the maximum nested divergence. These splits occur on a first-come-first-served basis according to program order. There are a few deductions which can be made about this splitting in the worst case.

Given this result, if a wavefront has a heterogeneous execution mask when it reaches the branch selected for splitting by the programmer, it is guaranteed to split.

Theorem 5.2.2. *In the worst case, wavefronts are guaranteed to split at the first $\mathcal{S}$ branches in wavefront execution order marked for splitting.*

Proof. The objective of this proof is to show that, in the worst case, all branches marked for splitting are reachable, looking at non-nested followed by nested branches.

1. Consider branch $j \leq \mathcal{S}$ which is marked for splitting that is not nested inside another branch. There is at least one additional wavefront slot free for a split because $j \leq \mathcal{S}$. By Lemma 5.2.1, the execution mask is heterogeneous. Hence, a new split is allocated. Therefore, in the worst case, the split is guaranteed to be performed.

2. Consider branch $k \leq \mathcal{S}, k \neq j$ which is marked for splitting that is nested inside another branch. This branch can only be split if execution is not skipped at runtime. Due to branch serialization, the only way for the branch to be skipped is if the wavefront had a homogeneous execution mask at one of the outer branches. By Lemma 5.2.1, this is a contradiction. Therefore, branch k must be reachable. Following the same line of reasoning as with branch j, branch k must also be split.

By exhaustively showing that a wavefront must split at a branch whether or not it is nested within another branch, we show that all wavefronts are guaranteed to split at the first $\mathcal{S}$ branches. □

With a set of IPGs corresponding to different split configurations, it is possible to attain the WCET for an individual wavefront slot, L_w. The WCET for simple programs may be deduced by inspection. More complex structures require the use of an ILP solver to solve the IPET problem. Using the WCET values from these CFGs and IPGs of individual wavefronts, it is now possible to put them together into a model spanning multiple wavefronts.

5.3 Modeling Multiple Wavefronts

Section 5.2 deals with determining the WCET for a single wavefront using a hybrid analysis approach. In particular, the analysis supports estimating a worst-case bound for wavefronts that implement wavefront splitting. This model is helpful for understanding the behaviour of wavefronts in isolation; in reality, as seen in Chapter 2, wavefronts execute among other wavefronts. There may be scheduling interference from other wavefronts on the same SIMD unit and interference from wavefronts on other SIMDs or even CUs for memory instructions. Therefore, it is important to extend the hybrid analysis model to support splitting on the fully parallel GPU hardware as presented in Chapter 2 with splitting as presented in Chapter 4. This section aims to present such a model and derive a WCET for the kernel as a whole across all workgroups, L_{kernel}.

5.3.1 Workgroup Batching

If there are not enough hardware resources to execute all workgroups, some workgroups must wait for others to complete. In the worst case, workgroups are executed serially in batches. Consider again the configuration in Figure 5.1. The number of workgroups, G, and the number of work-items per workgroup, T_{WG}, were chosen such that the workgroups fully utilize the hardware. Suppose that the programmer specified more workgroups; in other words,

$$GT_{WG} > \mathcal{W}T_{WF}$$

In this case, the workgroups must execute in two or more batches. The number of batches, B, is given by

$$B = \left\lceil \frac{G}{\left\lfloor \frac{\mathcal{W} \times T_{WF}}{T_{WG}} \right\rfloor} \right\rceil \tag{5.1}$$

Each batch consists of the maximum number of wavefronts that can execute in parallel on the GPU hardware. This serialized behaviour must be taken into account when grouping all wavefronts together to get the overall WCET, L_{kernel}.

5.3.2 General Model

The GPU scheduler interleaves execution of wavefronts. Consequently, wavefront IPTs are also interleaved. Figure 5.5 shows the interleaved execution times of three wavefronts. To properly analyze the wavefronts individually, it is necessary to separate them into traces of IPTs exclusive to a wavefront. The time interval between IPTs of a wavefront indicates the level of interference from other wavefronts. However, L_w alone does not account for the start time of the first instruction of a wavefront. Since the SIMD can only execute one wavefront at a time, there is some latency before the final wavefront has had a chance to begin execution. There is an initialization delay, D, that must also be considered and taken into account in the model.

From Figure 5.5, the overall L_{kernel} is split up over the B batches. The worst-case latency for completion of each batch corresponds to the final wavefront that completes

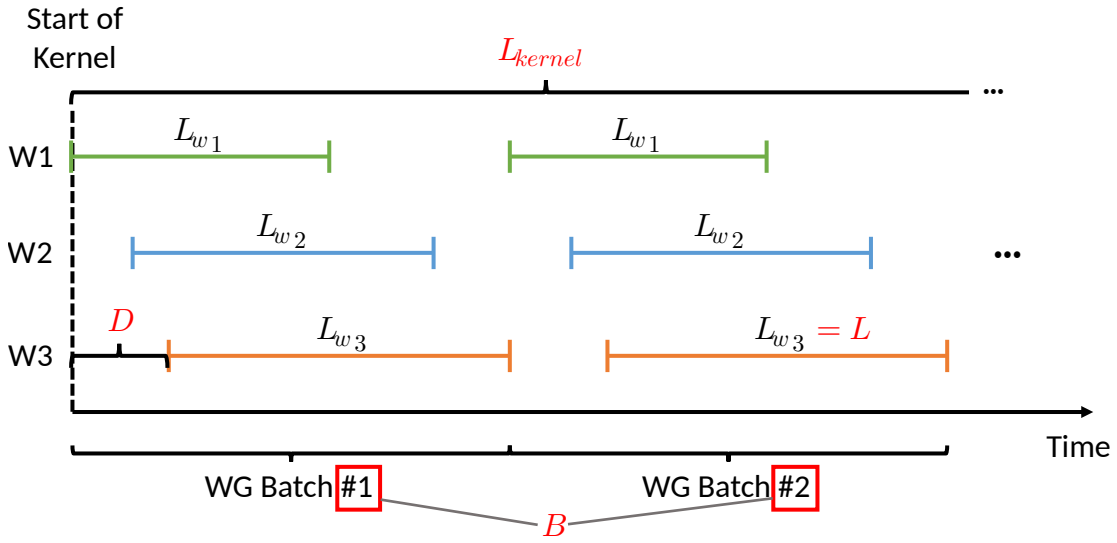


Figure 5.5: Timing diagram demonstrating key components of the general model

execution. The final wavefront to complete execution completes at a time $L + D$ after the start of the batch, where L is equal to the worst-case L_w of all wavefronts. Hence, the general model for L_{kernel} is

$$\underbrace{L_{kernel}}_{\substack{\text{WCET of all WFs} \\ \text{executing kernel}}} = \underbrace{B}_{\substack{\text{number of} \\ \text{WG batches}}} \times (\underbrace{L}_{\text{worst-case } L_w} + \underbrace{D}_{\text{initialization delay}}) \tag{5.2}$$

5.3.3 Expanding the Model

The general model in Equation 5.2 can be made more explicit. Equation 5.1 already expands upon the B factor. The $L + D$ factor can also be expanded, as shown in Equation 5.3.

$$L + D = \max_{\substack{1 \le i \le \mathcal{W} \\ 0 \le j \le \mathcal{S}}} \left(\delta_i^j + {L_w}_i^j \right) \tag{5.3}$$

The δ_i^j component represents the measured worst-case initialization delay for each wavefront relative to the start of a batch and ${L_w}_i^j$ is the computed WCET of each wavefront, as

presented in Section 5.2. Figure 5.6 presents how Equation 5.3 is used to determine $L + D$ for a batch.

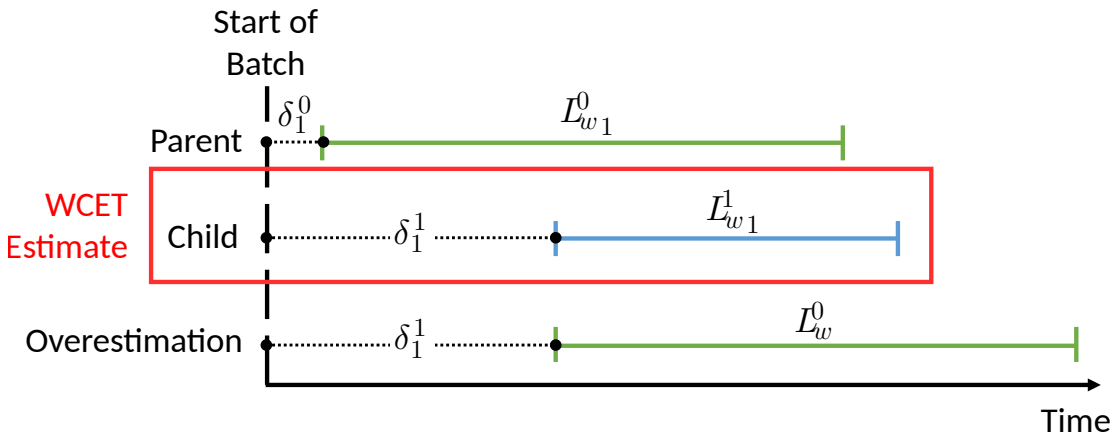


Figure 5.6: Example in which the model avoids overestimation

This model actually avoids some overestimation. Consider the case where a wavefront splits near the end of the kernel. This corresponds to the start time of the newly created split, meaning the child will have a significantly larger δ_i^j value. However, the child wavefront will almost certainly have a smaller $L_{w_i}^{\ j}$ than the parent.

Blindly combining the maximum δ_i^j and maximum $L_{w_i}^{\ j}$ will result in an overestimation, as shown in the third line of the timing diagram in Figure 5.6. Applying the max operator in Equation 5.3 after the sum ensures that the mismatch is avoided, and instead a more realistic estimate is computed. In the case of Figure 5.6, that is the second line in the timing diagram.

Putting Equations 5.1, 5.2, and 5.3 together gives

$$L_{kernel} = \left\lceil \frac{G}{\left\lfloor \frac{\mathcal{W} \times T_{WF}}{T_{WG}} \right\rfloor} \right\rceil \max_{\substack{1 \le i \le \mathcal{W} \\ 0 \le j \le \mathcal{S}}} \left(\delta_i^j + L_{w_i}^{\ j} \right) \tag{5.4}$$

The model represents a consolidation of two different approaches presented by Betts et al. and now supports wavefront splitting. It can be applied to applications executing using the splitting implementation to verify its capability.

5.3.4 Modeling without a Parallel Execution Guarantee

The addition of $\mathcal{S}$ SIMD vector pipelines guarantees parallel execution of splits. This is incorporated in the model. As a final exercise, it will be useful to compare the expression from Equation 5.4 with one in which there is no guarantee of parallel execution of splits. In other words, the splits must be serialized. This is summarized in Equation 5.5.

$$L_{kernel} = \left\lceil \frac{G}{\left\lfloor \frac{\mathcal{W} \times T_{WF}}{T_{WG}} \right\rfloor} \right\rceil \max_{\substack{1 \leq i \leq \mathcal{W} \\ 0 \leq j \leq \mathcal{S}}} \left(\delta_i^j + \sum_{0 \leq j \leq \mathcal{S}} {L_{w_i}}^j \right) \tag{5.5}$$

The summation in this model will likely lead to an increased worst-case estimate. It underscores the importance of guaranteeing parallel execution and the resulting model for L_{kernel}. Chapter 6 will continue to examine the effect both of these models have on the computed worst case bound.

Chapter 6

Results

Chapters 4 and 5 presented an implementation and analysis of wavefront splitting on AMD GPUs. The purpose of this chapter is to demonstrate the implementation and analysis and assess their benefits with benchmarks. All implementation was completed in the gem5 simulator. Hence, all results come from simulation traces corresponding to the instrumentation points presented in Chapter 5.

Section 6.1 will start with an overview of how GPU benchmarks are supported on AMD GPUs and on gem5 specifically. Verifying that the implementation still provides some performance benefit is addressed in Section 6.2. Finally, Section 6.3 applies the analysis to the benchmarks to verify that the computed WCET provides an upper bound and examines if splitting can result in an improved execution time in the worst case compared to not splitting.

6.1 Benchmarks

The Rodinia benchmark suite was chosen to assess the implementation. The Rodinia suite is designed for heterogeneous systems, particularly CPU-GPU systems similar to the GCN architecture used in gem5 [7]. It features a diverse set of programs and algorithms that test various program and data structure organizations as well as synchronization patterns. Before being able to run the suite, there are some challenges with compatibility and measurement of the benchmarks with gem5. These issues are applicable even outside the scope of the Rodinia suite; namely,

1. few benchmarks are written natively in HIP,

2. the constructs supported by the simulator are poorly documented, and

3. simulation takes a long time to complete.

Most GPU benchmarks are written in CUDA. They can be converted to HIP using a tool known as `hipify` [16]. There are two `hipify` tools: `hipify-perl` and `hipify-clang`. The `hipify-perl` tool has no dependencies and performs a simple regular expression replacement, which may miss some constructs. The `hipify-clang` tool is more robust but requires the CUDA library as a dependency [12]. Fortunately, to bypass the need for this tool, several authors have developed HIP versions of the Rodinia suite. One such translated suite is used for this thesis [36]. Other benchmarks either must be written natively in HIP or converted from CUDA.

The Rodinia benchmarks use several constructs which are unsupported in the simulator. However, most benchmarks work with some slight modifications. Those that do not are `lavaMD`, `lud`, `myocyte`, `pathfinder`, and `streamcluster`. Additionally, the `leukocyte` benchmark in the HIP repository does not compile. Finally, some benchmarks have multiple versions, of which at least one works correctly. In total, 16 of 22 benchmarks from the Rodinia suite execute without issues in gem5. Table 6.1 presents the Rodinia benchmarks used for performance measurements. Each benchmark is split up into three categories based on the size of the data input to the kernel. These sizes provide some insight into what configurations and input sizes benefit most from splitting.

6.2 Performance Results

First and foremost, wavefront splitting is a performance optimization; all prior works examine and verify it as such. For this thesis, there are specific implementation details which modify wavefront splitting to be applicable to the worst case. In particular, these details pertain to the static selection of branches which trigger splitting. Furthermore, reconvergence is avoided in favour of simplified worst-case analysis. Therefore, the purpose of this section is to measure the performance benefits of the wavefront splitting technique as it is implemented.

Figure 6.1 presents the speedups observed for the Rodinia suite for the small, medium, and large input sizes[1] listed in Table 6.1. For these measurements, the start of the kernel is

[1]There are a few missing data entries for the medium and large input sizes. Some applications take several hours to complete on the simulator and run into timeout errors.

Table 6.1: Rodinia benchmark suite with input parameters used in testing

Name	Small Inputs	Medium Inputs	Large Inputs
`backprop`	16x16 input layer	512x512 input layer	4096x4096 input layer
`bfs`	4096 vertices	16384 vertices	65536 vertices
`b+tree`	100 elements	1000 elements	10000 elements
`cfd`	16 elements	1024 elements	65536 elements
`dwt2d`	4x4 image	16x16 image	192x192 image
`gaussian`	16 unknowns	64 unknowns	256 unknowns
`heartwall`	1 frame	10 frames	100 frames
`hotspot`	64x64 matrix	512x512 matrix	1024x1024 matrix
`hybridsort`	16 elements	128 elements	1024 elements
`kmeans`	100 points	1000 points	10000 points
`nn`	100 entries	1000 entries	10000 entries
`nw`	16x16 matrix	64x64 matrix	256x256 matrix
`particlefilter`	16x16 matrix	64x64 matrix	512x512 matrix
`srad`	10x10 image	50x50 image	100x100 image

determined when the first wavefront reaches the start of kernel IPT. For the end of program, a new IPT is inserted when a CU is put to sleep after all wavefronts finish execution. The last of these IPTs signifies the end of the kernel.

From the results it is clear that the performance benefit is constrained to a handful of the benchmarks and to varying extent by their input sizes. The mean performance speedup for all applications is 1.02x. The best case improvement, a 1.18x speedup, is exhibited by the Needleman-Wunsch (`nw`) benchmark, which is a dynamic programming algorithm to determine similarities between genetic sequences [21]. The program contains multiple loops that each contain one or more conditional statements and reads from and writes to memory. The conditional statements are based on the thread index, indicating branch divergence. Therefore, there is a structural opportunity for splitting. Otherwise, there are a few benchmarks which demonstrate a marginal improvement and others which demonstrate approximate parity with the baseline.

Overall, this outcome is a slightly lessened speedup compared to prior work in wavefront splitting. However, the key difference with this work is the absence of reconvergence of splits. Without reconvergence, wavefront splits are more likely to execute identical instructions out of lockstep with each other. Reconvergence would restrict the latency overhead from the increase in active wavefront slots strictly to the branch divergent blocks in the kernel. Furthermore, it would allow splitting again at different branches with different

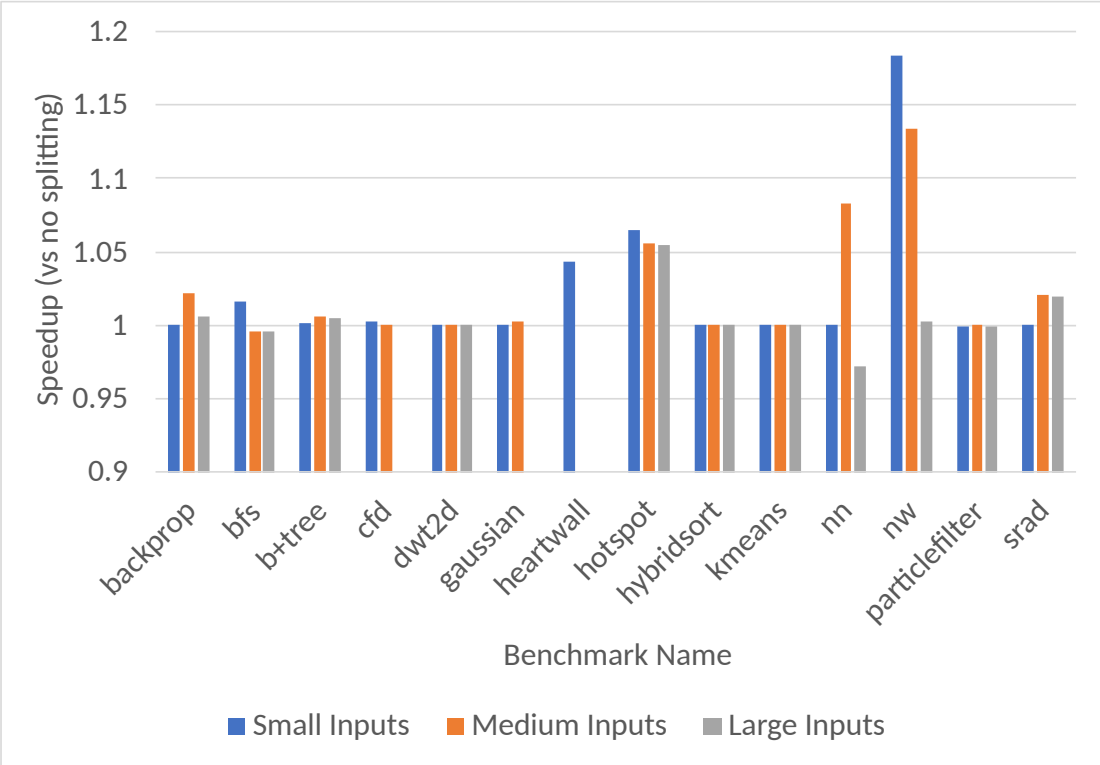


Figure 6.1: Speedup exhibited by Rodinia benchmarks at various input sizes

execution mask values.

A significant cost associated with splitting is memory contention. Although the implementation provides duplicated SIMD vector pipelines to handle each of the $\mathcal{S}$ splits, there is no increase to the memory bandwidth. This memory contention works against the level of branch divergence in a workload. Notably, some Rodinia benchmarks benefit significantly more than others from splitting. These are mostly benchmarks which have branching structure within loops with large bounds. Therefore, there is significant branch path serialization in the baseline. Damani et al. showed that raytracing benchmarks show much greater branch divergence than other workloads. However, there are few raytracing benchmarks, especially those targeting AMD hardware with HIP. As a result, wavefront splitting is not a universally performant strategy; it is limited to workloads that exhibit large quantities of branch divergence.

These results show that performance gained from wavefront splitting, while it does not degrade performance in most cases, does not provide a significant performance benefit.

However, for some benchmarks with some input sizes, there is a measurable improvement. This means that it is important to consider the application and the inputs when determining if wavefront splitting is beneficial. Simply applying it to any workload may result in needless hardware cost. If a workload does not benefit from splitting, it is better to use the additional split hardware for additional parallelism (i.e. increase the number of inflight wavefronts on SIMD hardware).

6.3 Applying the Analysis

The purpose of this section is to assess whether wavefront splitting can reduce the execution time in the worst case. The data is obtained by running the benchmarks on the wavefront splitting implementation, measuring the program segments, and applying the analysis from Chapter 5. The analysis results in some computed analytical WCET for the kernels. This information is helpful to make conclusions about the approach to the analysis and the resulting worst-case estimates.

6.3.1 Synthetic Benchmark

The Rodinia kernels may not achieve the critical instance as outline in Section 5.1. Hence, an additional synthetic benchmark is used to verify the analysis. This synthetic benchmark contains $\log_2(64) = 6$ layers of conditional blocks. Execution of each block is unique to the work-item index with respect to the wavefront. Therefore, each work-item takes a unique path through the resulting kernel CFG. If splitting is enabled globally and $\mathcal{S} = T_{WF}$, the kernel maximally splits each wavefront, meaning the second critical instance condition can be met for any valid value of $\mathcal{S} \in [0, T_{WF}]$. By matching the total number of work-items executing the benchmark to the total available hardware, the first critical instance condition is also met.

Additionally, the synthetic benchmark does not contain any branches that are conditional upon data; all conditions are based on the work-item index, a static property. This means that all invocations of the benchmark will statically follow the same path regardless of the input data. With hybrid analysis, the measurements should be obtained across a wide variety of data inputs to exercise as many of the possible paths as possible. This provides more realistic data for computing the worst-case bound. By making all conditions dependent on work-item index, the benchmark forgoes the need to test across multiple inputs.

Figure 6.2 presents the data for the kernel WCET in graphical form. Figure 6.2a presents the observed worst case and the computed worst case given a guarantee of parallel execution. Figure 6.2b presents the same data as Figure 6.2a, but also includes the computed worst case if the splits are serialized. These graphs depict two types of data:

- **Observed WC**. The observed worst case represents the execution time from the start of the first wavefront of the first workgroup to the completion of the final wavefront in the last workgroup, as measured by the respective IPTs. It is shown in blue in Figure 6.2. For the synthetic benchmark, there is only a single execution of the program needed since the execution paths are all independent of the input data. The benchmark runs under each value of $\mathcal{S}$ and produces IPT measurements. Additionally, the value represented by the yellow bar is the observed WCET in an alternative implementation without the guarantee of parallel execution.
- **Computed WC**. The computed worst-case is the value of L_{kernel} using the produced IPT measurements for each value of $\mathcal{S}$. Given the guarantee of parallel execution in the implementation, L_{kernel} is defined by Equation 5.4 and is shown in orange in Figure 6.2. Without a guarantee of parallel execution of splits, splits are executed serially in the worst case. This means that a split will at worst need to wait for all other splits across all wavefronts to complete before executing for every batch. This value is represented by the grey bar in Figure 6.2b. Both computed worst-case data points are computed using data from the observed worst case.

There are a couple key findings from this data. First, the computed worst case serves as an upper bound on all observed worst case values, as desired. Second, the guarantee of parallel execution is crucial to have a reasonably low upper bound. Furthermore, only with the parallel guarantee does the execution time improve in the worst case with wavefront splitting. In the serialized case, the computed worst case increases roughly linearly with the number of splits.

6.3.2 Rodinia Benchmarks

Along with performance results, the Rodinia benchmarks are also used to demonstrate the analysis. Figure 6.3 presents the same data as Figure 6.2 for nine of these benchmarks. In these graphs, the observed worst case is based on a single input. Ideally, this value would be the maximum across a variety of inputs. In some cases, such as `bfs`, the computed worst case follows a similar pattern to the synthetic benchmark: a reduction in the WCET

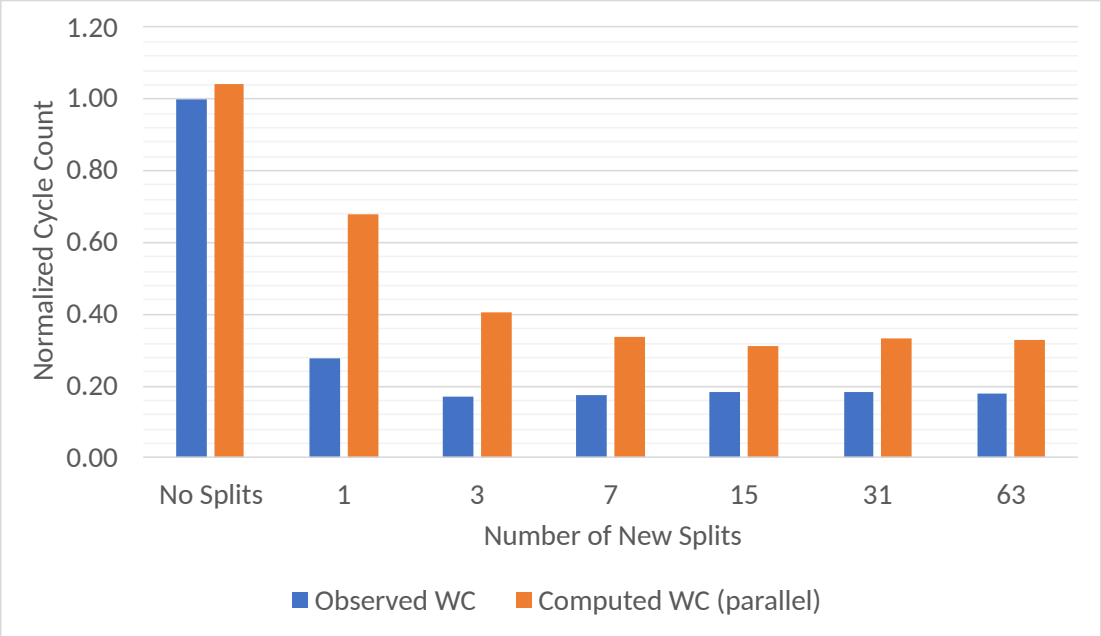
when splitting. In other cases, such as `gaussian`, the computed worst case actually increases. This is due to the aforementioned memory contention and a relative lack of branch divergence in the kernel.

Additionally, computed WCETs for some of the benchmarks are significantly larger than the observed times. This is impacted by two factors:

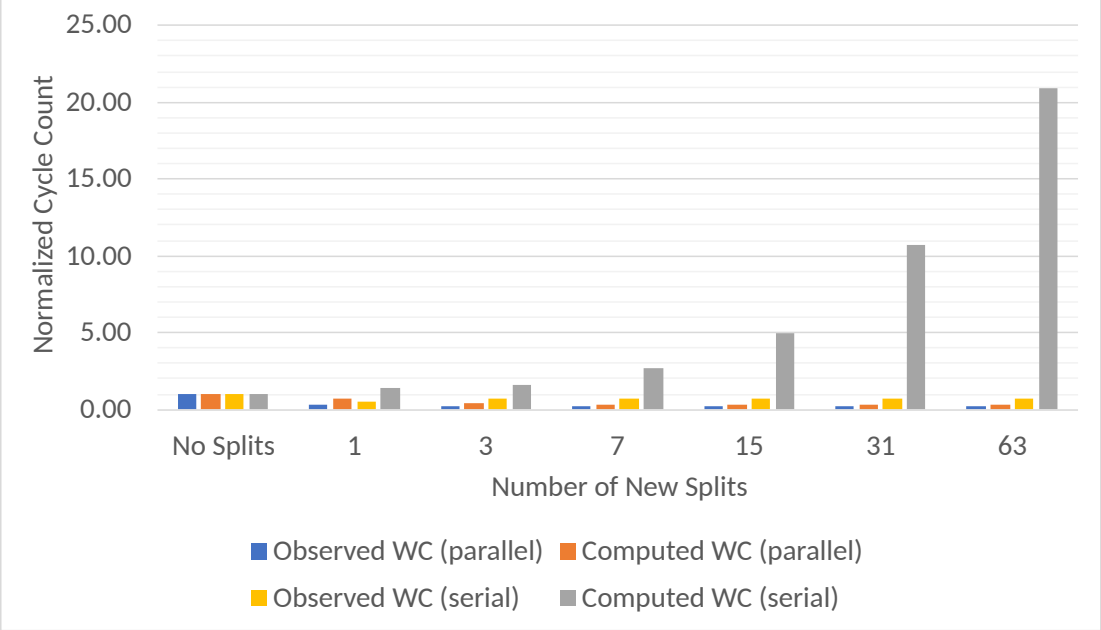
1. There are many variations in execution times of certain segments of the IPG. Using the maximum value naturally results in an larger upper bound.

2. The benchmarks for which computed WCETs are significantly larger than the observed worst case had the largest number of workgroups executing. The multiplicative behaviour of the batch count in Equation 5.4 amplifies the effect of minor variations in the program segments.

To summarize, as with the performance benefit, the benefit in the computed worst-case is largely application-dependent. The parallel execution guarantee is crucial to obtain a reduced WCET when splitting. By providing an upper bound on the observed worst case, the model presented serves to provide the timing predictability for GPUs with wavefront splitting.



(a) Observed and computed WCET with parallel execution guarantee



(b) Same as Figure 6.2a but including data without parallel execution guarantee

Figure 6.2: Observed and analytical WCET for synthetic benchmark

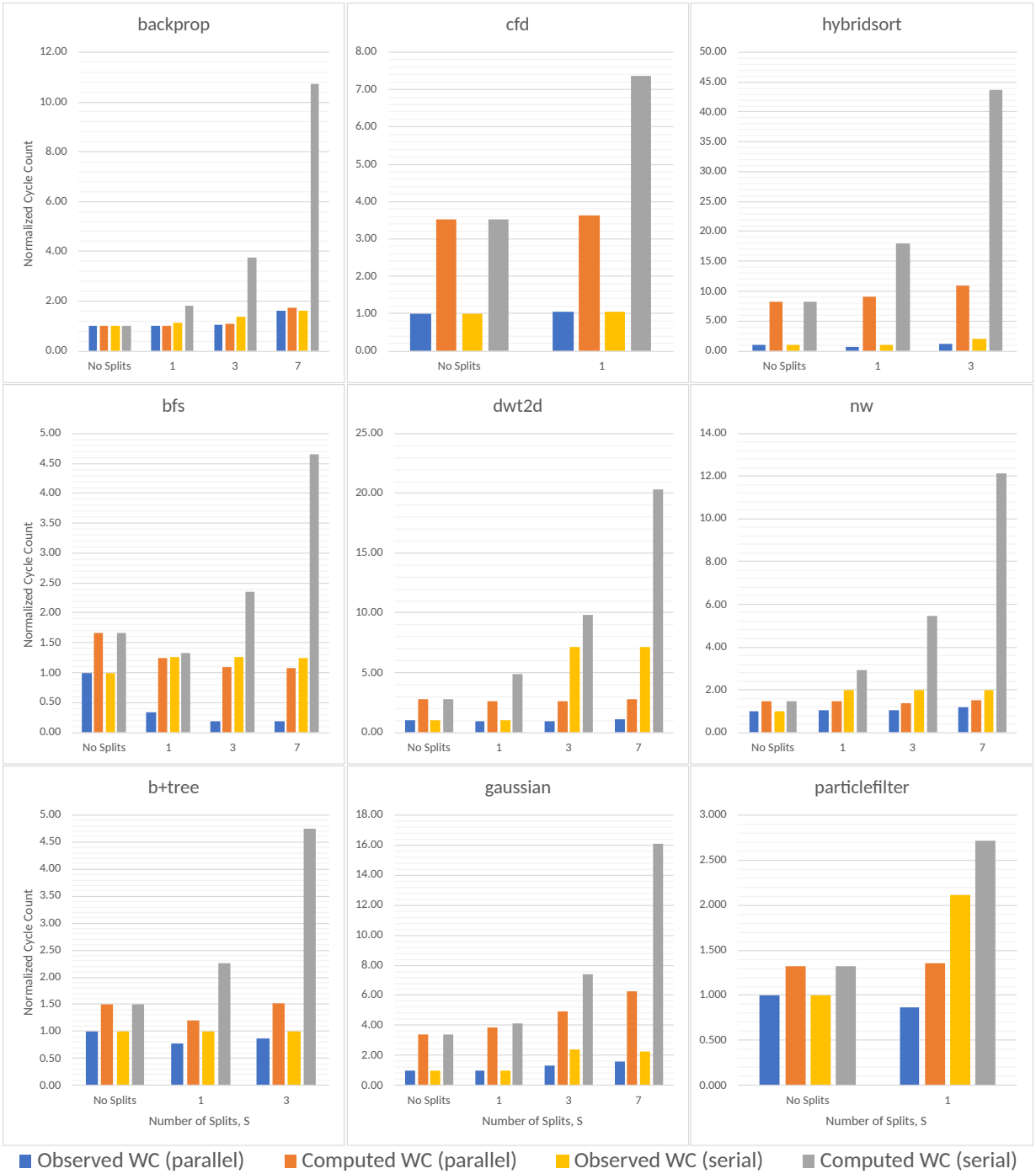


Figure 6.3: Observed and analytical WCET for Rodinia benchmarks

Chapter 7

Conclusions

The purpose of this chapter is to summarize the discussion and findings in Chapters 4 through 6, address the limitations of the implementation and analysis, and present opportunities for future work.

7.1 Discussion

The objective of the implementation was to leverage wavefront splitting to address branch divergence in AMD GPUs for real-time and safety-critical systems.

Chapter 4 discusses the wavefront splitting implementation. Fundamentally, wavefront splitting allows wavefronts to split upon encounter with a divergent branch, when the work-items within a wavefront must execute two different outcomes of the branch. To avoid serializing the branch outcomes, wavefront splitting creates two separate schedulable entities, known as wavefront splits. These separate entities may be scheduled and executed in an interleaved fashion. Additionally, through the addition of vector pipelines to each SIMD, the implementation guarantees parallel scheduling and execution of splits. As long as the active work-items of each split do not overlap, the splits can be executed together. In order to allow for more robust analysis of wavefront splitting, the implementation only triggers splits based on a static assignment to a bit of a GPU register. This strategy allows for a quantitative understanding of the number of splits which can and will occur based on the number of branches that occur when the register bit is set. Together, these techniques form the base implementation of this thesis.

Chapter 5 describes the GPU hybrid analysis framework specifically applied to wavefront splitting. The analysis in this thesis aims to quantify the behaviour of GPUs with wavefront splitting. The scheduling details for GPU kernels are complex; a full-featured analysis must take into account this behaviour for individual wavefronts as well as the interaction between wavefronts on the GPU processor. This motivates the use of hybrid analysis, which takes empirical timing measurements corresponding to edges in the kernel CFG. Using these measurements, the analysis stitches together to create a cohesive estimate of the WCET. This thesis uses this approach and incorporates wavefront splitting as described in the implementation.

Chapter 6 demonstrates the impact of wavefront splitting on various benchmarks in the worst and average cases to verify the analysis and implementation with empirical measurements. Since reconvergence was avoided to simplify the analysis, the speedups seen are less dramatic than those found in some prior work. However, for some workloads, there is still a performance improvement value provided by splitting. Additionally, the analytical worst case was shown to be decreased for certain benchmarks while still serving as an upper bound on the observed worst case.

7.2 Limitations and Future Work

This thesis presents wavefront splitting as seen in prior work with some modifications and analysis to support the worst-case for real-time systems. While these are novel contributions, there are some limitations and opportunities for future work in the area.

The first major challenge is that hybrid analysis does not provide a guarantee of worst-case behaviour, only an estimate. Ideally, it would be replaced with a fully static analysis framework. Future work will need to assess how to incorporate the complexities of GPU kernel execution into fully static worst-case analysis.

The implementation in this thesis avoids reconvergence of wavefront splits. This approach is taken to avoid additional complexity to the analysis. However, it does not allow for the best performance in benchmarking; some benchmarks have very small performance benefits from wavefront splitting as a result. Therefore, there is an opportunity for future work to incorporate reconvergence into the implementation and analysis. Additionally, there is an opportunity to solve the problem of which branches to choose to split. Choosing where to split can be made into an ILP and solved, though there are some complexities with representing branch divergence and splitting.

Finally, wavefront splitting is one technique in the broader domain of GPU hardware.

As previously mentioned and explored in prior work, modern GPUs are not designed with the worst case in mind. Therefore, there is an opportunity for research work to redesign GPU hardware and corresponding open-source and well-documented software support to better serve real-time and safety-critical systems. Such a system would allow developers of real-time systems to avoid current closed-source or poorly documented GPU options. Additionally, it would help with the above goal of a fully static analysis framework.

7.3 Conclusion

This thesis presents an implementation and analysis framework for wavefront splitting to target real-time and safety-critical systems. It introduces some new implementation details that enable worst-case analysis. To analyze, the implementation is evaluated with a hybrid analysis framework, in which smaller components of GPU kernel execution are measured and combined using an analytical model. This approach provides a wavefront splitting method that can be accompanied by an estimate of the WCET of the kernel and in some cases a reduction in the WCET compared to the baseline without splitting. By providing an upper bound on the observed worst case, the model presented serves to provide the timing predictability for GPUs with wavefront splitting.

The contributions of this thesis are part of an initial push to re-engineer GPU architecture to better suit workloads that are sensitive to worst-case behaviour. As expected, there are many opportunities for future work in the area. Eventually, this work will be a part of the path to a fully open-source GPU architecture and associated software stack that can be certified for safety-critical systems.

References

[1] Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, and Margaret Martonosi. *General-Purpose Graphics Processor Architecture*. Morgan & Claypool, 2018.

[2] Tanya Amert, Nathan Otterness, Ming Yang, James H. Anderson, and F. Donelson Smith. GPU scheduling on the NVIDIA TX2: Hidden details revealed. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 104–115, 2017.

[3] Marc Benito, Matina Maria Trompouki, Leonidas Kosmidis, Juan David Garcia, Sergio Carretero, and Ken Wenger. Comparison of GPU computing methodologies for safety-critical systems: An avionics case study. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 717–718, 2021.

[4] Adam Betts and Alastair Donaldson. Estimating the WCET of GPU-accelerated applications using hybrid analysis. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 193–202, 2013.

[5] Nicolas Brunie, Caroline Collange, and Gregory Diamos. Simultaneous branch and warp interweaving for sustained GPU performance. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 49–60, 2012.

[6] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A quantitative study of irregular programs on GPUs. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*, pages 141–151, 2012.

[7] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.

[8] Sana Damani, Mark Stephenson, Ram Rangan, Daniel Johnson, Rishkul Kulkarni, and

Stephen W. Keckler. GPU subwarp interleaving. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, 2022.

[9] Advanced Micro Devices. Graphics core next architecture reference guide, 2016.

[10] Advanced Micro Devices. Introducing RDNA architecture, 2019.

[11] Advanced Micro Devices. HIP programming guide, 2021.

[12] Advanced Micro Devices. HIPify reference guide v5.1. https://docs.amd.com/bundle/HIPify-Reference-Guide-v5.1/page/HIPify.html, 2022.

[13] Charles N. Fischer, Ronald K. Cytron, and Richard J. LeBlanc. *Crafting A Compiler*. Addison-Wesley Publishing Company, USA, 1st edition, 2009.

[14] Wilson W.L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 407–420, 2007.

[15] Tony Gutierrez, Sooraj Puthoor, Tuan Ta, Matt Sinclair, and Brad Beckmann. The AMD gem5 APU simulator: Modeling GPUs using the machine ISA. ISCA, 2018.

[16] Ruobing Han, Blaise Tine, Jaewon Lee, Jaewoong Sim, and Hyesoon Kim. Supporting CUDA for an extended RISC-V GPU architecture. *CoRR*, abs/2109.00673, 2021.

[17] Alex Herrera. AMD Ryzen Threadripper PRO processors: Introducing the third wave of workstation computing, 2020.

[18] Yijie Huangfu and Wei Zhang. Static WCET analysis of GPUs with predictable warp scheduling. In *2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*, pages 101–108, 2017.

[19] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[20] Leonidas Kosmidis, Iván Rodriguez, Alvaro Jover-Alvarez, Sergi Alcaide, Jérôme Lachaize, Olivier Notebaert, Antoine Certain, and David Steenari. Gpu4s: Major project outcomes, lessons learnt and way forward. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1314–1319, 2021.

[21] Vladimir Likic. The needleman-wunsch algorithm for sequence alignment. *Lecture*

given at the 7th Melbourne Bioinformatics Course, Bi021 Molecular Science and Biotechnology Institute, University of Melbourne, pages 1–46, 2008.

[22] Huanxin Lin, Cho-Li Wang, and Hongyuan Liu. On-GPU thread-data remapping for branch divergence reduction. *ACM Trans. Archit. Code Optim.*, 15(3), oct 2018.

[23] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jerónimo Castrillón, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Marjan Fariborz, Amin Farmahini Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc S. Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. The gem5 simulator: Version 20.0+. *CoRR*, abs/2007.03152, 2020.

[24] Piergiuseppe Mallozzi, Patrizio Pelliccione, Alessia Knauss, Christian Berger, and Nassar Mohammadiha. *Autonomous Vehicles: State of the Art, Future Trends, and Challenges*. Springer International Publishing, 2019.

[25] Chris McClanahan. History and evolution of GPU architecture, 2010.

[26] Jiayuan Meng, David Tarjan, and Kevin Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 235–246, New York, NY, USA, 2010. Association for Computing Machinery.

[27] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. Improving gpu performance via large warps and two-level warp scheduling. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 308–317, 2011.

[28] NVIDIA. NVIDIA Ampere GA102 GPU architecture, 2021.

[29] NVIDIA. NVIDIA DRIVE end-to-end solutions for autonomous vehicles, 2022.

[30] Nathan Otterness and James H. Anderson. AMD GPUs as an alternative to NVIDIA for supporting real-time workloads. In *ECRTS*, 2020.

[31] Nathan Otterness and James H. Anderson. Exploring AMD GPU scheduling details by experimenting with "worst practices". In *29th International Conference on Real-Time Networks and Systems*, RTNS'2021, pages 24–34, New York, NY, USA, 2021. Association for Computing Machinery.

[32] Nathan Otterness, Ming Yang, Sarah Rust, Eunbyung Park, James H. Anderson, F. Donelson Smith, Alex Berg, and Shige Wang. An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 353–364, 2017.

[33] M. Papadrakakis, G. Stavroulakis, and A. Karatarakis. A new era in scientific computing: Domain decomposition methods in hybrid CPU-GPU architectures. *Computer Methods in Applied Mechanics and Engineering*, 200(13):1490–1508, 2011.

[34] Ana Pereira and Carsten Thomas. Challenges of machine learning applied to safety-critical cyber-physical systems. *Machine Learning and Knowledge Extraction*, 2(4):579–602, 2020.

[35] Minsoo Rhu and Mattan Erez. The dual-path execution model for efficient GPU control flow. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 591–602, 2013.

[36] Corbin Robeck and Aryan Salmanpour. ROCm developer tools: HIP examples. https://github.com/ROCm-Developer-Tools/HIP-Examples, 2016.

[37] Timothy G. Rogers, Daniel R. Johnson, Mike O'Connor, and Stephen W. Keckler. A variable warp size architecture. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 489–501, 2015.

[38] Stanford University. Convolutional neural networks, 2022.

[39] Lothar Thiele and Reinhard Wilhelm. Design for timing predictability. *Real-time systems*, 01 2004.

[40] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In

2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS), pages 235–246, 2010.

[41] Ming Yang, Nathan Otterness, Tanya Amert, Joshua Bakita, James H. Anderson, and F. Donelson Smith. Avoiding pitfalls when using NVIDIA GPUs for real-time tasks in autonomous systems. In Sebastian Altmeyer, editor, *30th Euromicro Conference on Real-Time Systems, ECRTS 2018, July 3-6, 2018, Barcelona, Spain*, volume 106 of *LIPIcs*, pages 20:1–20:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

APPENDICES

Appendix A

AMD and NVIDIA Terminology

Table A.1 lists some relevant translations between AMD and the associated NVIDIA concepts and terms.

Table A.1: Mapping between equivalent terms in AMD and NVIDIA GPU models

AMD Term	**NVIDIA Term**
Compute Unit (CU)	Simultaneous Multiprocessor (SM)
SIMD Unit	Streaming Processor (SP)
Wavefront	Warp
Workitem	Thread
Workgroup	Thread Block