Inter-warp Divergence Aware Execution on GPUs

A Thesis Presented
by

Chulian Zhang

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

Master of Science

in

Electrical and Computer Engineering

Northeastern University
Boston, Massachusetts

April 2016
To my family.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>v</td>
</tr>
<tr>
<td>List of Tables</td>
<td>vii</td>
</tr>
<tr>
<td>List of Acronyms</td>
<td>viii</td>
</tr>
<tr>
<td>Abstract of the Thesis</td>
<td>ix</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Problem Definition</td>
<td>4</td>
</tr>
<tr>
<td>1.2 Contribution</td>
<td>5</td>
</tr>
<tr>
<td>1.2.1 Programming Level</td>
<td>5</td>
</tr>
<tr>
<td>1.2.2 Architecture Level</td>
<td>6</td>
</tr>
<tr>
<td><strong>2 Related Work</strong></td>
<td>7</td>
</tr>
<tr>
<td>2.1 Background Subtraction on GPU</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Thread Scheduling on GPU</td>
<td>9</td>
</tr>
<tr>
<td><strong>3 Background</strong></td>
<td>11</td>
</tr>
<tr>
<td>3.1 GPU Programming</td>
<td>11</td>
</tr>
<tr>
<td>3.2 GPU Architecture</td>
<td>12</td>
</tr>
<tr>
<td>3.2.1 Overview</td>
<td>12</td>
</tr>
<tr>
<td>3.2.2 CPU-GPU Interaction</td>
<td>14</td>
</tr>
<tr>
<td>3.2.3 Single Instruction Multiple Thread</td>
<td>14</td>
</tr>
<tr>
<td>3.2.4 Memory Hierarchy &amp; Divergence</td>
<td>15</td>
</tr>
<tr>
<td>3.2.5 Occupancy</td>
<td>16</td>
</tr>
<tr>
<td>3.2.6 Scheduler</td>
<td>17</td>
</tr>
<tr>
<td>3.3 GPGPU-Sim</td>
<td>18</td>
</tr>
<tr>
<td>3.3.1 GPGPU-Sim Architecture Overview</td>
<td>19</td>
</tr>
<tr>
<td>3.3.2 Running Application on GPGPU-Sim</td>
<td>19</td>
</tr>
<tr>
<td>3.3.3 GPGPU-Sim Output</td>
<td>20</td>
</tr>
<tr>
<td>3.4 Mixture of Gaussian</td>
<td>21</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>CPU &amp; GPU Hardware Threads and PC Trace</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>CPU &amp; GPU Architecture Comparison</td>
<td>3</td>
</tr>
<tr>
<td>3.1</td>
<td>GPU Thread Hierarchy &amp; Memory Hierarchy (Software View)</td>
<td>13</td>
</tr>
<tr>
<td>3.2</td>
<td>GPU Architecture Overview</td>
<td>13</td>
</tr>
<tr>
<td>3.3</td>
<td>CPU &amp; GPU Interconnection</td>
<td>15</td>
</tr>
<tr>
<td>3.4</td>
<td>GPU Memory Hierarchy (Hardware View)</td>
<td>16</td>
</tr>
<tr>
<td>3.5</td>
<td>Schedulers in GPU</td>
<td>18</td>
</tr>
<tr>
<td>3.6</td>
<td>GPGPU-Sim Overall Architecture</td>
<td>19</td>
</tr>
<tr>
<td>3.7</td>
<td>API Call Interception</td>
<td>20</td>
</tr>
<tr>
<td>3.8</td>
<td>MoG Flow</td>
<td>21</td>
</tr>
<tr>
<td>4.1</td>
<td>Memory Access for Coalesced and non-Coalesced Data Placement</td>
<td>26</td>
</tr>
<tr>
<td>4.2</td>
<td>Concurrency of Data Transfer &amp; Kernel Execution</td>
<td>27</td>
</tr>
<tr>
<td>4.3</td>
<td>Architectural Impact of General Optimizations</td>
<td>28</td>
</tr>
<tr>
<td>4.4</td>
<td>Performance for Alg-specific Optimization</td>
<td>31</td>
</tr>
<tr>
<td>4.5</td>
<td>Performance for Different Optimizations</td>
<td>33</td>
</tr>
<tr>
<td>4.6</td>
<td>Frame Group Optimization for Share Memory</td>
<td>34</td>
</tr>
<tr>
<td>4.7</td>
<td>Performance Over Group Size</td>
<td>34</td>
</tr>
<tr>
<td>5.1</td>
<td>PC Trace and Cache Misses for MoG</td>
<td>38</td>
</tr>
<tr>
<td>5.2</td>
<td>Standard Benchmark Evaluation</td>
<td>40</td>
</tr>
<tr>
<td>5.3</td>
<td>Conceptual IPC / Inter-Warp Divergence Trade-off</td>
<td>41</td>
</tr>
<tr>
<td>5.4</td>
<td>Overall Flow of WPS Calculation</td>
<td>42</td>
</tr>
<tr>
<td>5.5</td>
<td>WPS Calculation</td>
<td>43</td>
</tr>
<tr>
<td>5.6</td>
<td>Measured WPS for MoG and DXTC</td>
<td>43</td>
</tr>
<tr>
<td>5.7</td>
<td>Architecture View of a Streaming Multiprocessor (SM)</td>
<td>45</td>
</tr>
<tr>
<td>5.8</td>
<td>MoG WPS Contributors</td>
<td>46</td>
</tr>
<tr>
<td>5.9</td>
<td>WPSaS Structure</td>
<td>49</td>
</tr>
<tr>
<td>5.10</td>
<td>WPS / IPC Trade-off</td>
<td>51</td>
</tr>
<tr>
<td>5.11</td>
<td>WPS / IPC Trade-off</td>
<td>52</td>
</tr>
<tr>
<td>5.12</td>
<td>Comparisons of PC Trace and Cache Misses for MoG with &amp; without WPSaS</td>
<td>53</td>
</tr>
<tr>
<td>5.13</td>
<td>IPC, Temporal Under-utilization and Stall Comparisons</td>
<td>54</td>
</tr>
<tr>
<td>5.14</td>
<td>IPC Improvement Comparisons</td>
<td>56</td>
</tr>
</tbody>
</table>
List of Tables

3.1 HW Configuration .................................................. 14
3.2 Compute Capability for Fermi .................................. 17
3.3 GPGPU-Sim General Statistics ................................. 20

4.1 General Optimization Levels .................................... 25
4.2 Algorithm-Specific Optimizations ............................. 29

5.1 GPGPU-Sim Configuration ....................................... 36
List of Acronyms

**GPU**  Graphics Processing Unit.
**CPU**  Central Processing Unit.
**CUDA**  Compute Unified Device Architecture.
**OpenCL**  Open Computing Language.
**CV**  Computer Vision.
**WPS**  Warp Progression Similarity.
**WPSaS**  WPS aware Scheduler.
**MoG**  Mixture of Gaussians.
**PC**  Program Counter.
**SIMD**  Single Instruction Multiple Data.
**SIMT**  Single Instruction Multiple Thread.
**DMA**  Direct Memory Access.
**CTA**  Cooperative Thread Arrays.
**RF**  Register File.
**SM**  Streaming Multiprocessor.
**SRR**  Strict Round Robin.
**LRR**  Loose Round Robin.
**GTO**  Greedy then Oldest.
Abstract of the Thesis

Inter-warp Divergence Aware Execution on GPUs

by

Chulian Zhang

Master of Science in Electrical and Computer Engineering
Northeastern University, April 2016
Dr. Gunar Schirner, Adviser

GPUs have appeared as very efficient many-core platform to execute applications with massive thread-level parallelism. GPUs achieve high throughput by running many threads concurrently and switching between them rapidly to hide memory latency. With the introduction of general-purpose programming models such as CUDA and OpenCL, many applications are motivated to use GPUs to accelerate compute intensive kernels and see impressive speedups. With the trend toward using GPUs for a diverse range of applications (e.g. vision and scientific computing) new challenges have been raised. New challenges have been raised for both algorithm and architecture designer when targeting GPUs for general-purpose applications.

From an architecture perspective, one of the main challenges is inter-warp conflict in the shared resources including I$, D$ and compute units. The conflicts in the shared resources mainly caused by inter-warp divergence which is uneven execution progress across the concurrent warps. Excessive inter-warp divergence may hinder GPUs to achieve their peak throughput. This motivates the need for approaches that manage inter-warp divergence, avoiding I$ conflicts, for divergence-sensitive benchmarks. From an algorithm perspective, the challenge is how to understand algorithm-specific optimizations to achieve maximum utilization enhancing the overall performance.

This thesis primarily focuses on the architecture perspective. From an architecture perspective, this thesis quantitatively studies the benefits of inter-warp divergence aware execution on GPUs. To that end, the thesis first proposes a novel approach to quantify the inter-warp divergence by measuring the temporal similarity in execution progress of concurrent warps, which we call Warp Progression Similarity (WPS). Based on the WPS metric, this thesis proposes a WPS-aware Scheduler (WPSaS) to optimize GPU throughput. The aim is to manage inter-warp divergence to hide memory access latency and minimize resource conflicts and temporal under-utilization in compute units allowing GPUs to achieve their peak throughput. Our results demonstrate that WPSaS improves throughput by 10% with a pronounced reduction in resource conflicts and temporal under-utilization.
From an algorithm perspective, this thesis demonstrates a GPU implementation of background subtraction MoG (Mixture of Gaussians) algorithm that surpasses real-time processing for full HD resolution. This implementation applies both general and algorithm-specific optimization to achieve 101x speedup over sequential implementation without impact to the output quality.
Chapter 1

Introduction

On a CPU platform, at each point of time, many applications are running concurrently. Each application usually consists of only one or a few threads. Since the number of hardware cores is limited, only a few threads will be able to execute each time (Figure 1.1a shows a simplified illustration). The operating system is responsible for scheduling the threads to achieve fairness. The scheduling is performed in coarse-grain, i.e. in tens of milliseconds, as context switch between threads is pretty expensive. Applications are executing different code. Hence, the PC traces are completely different from each other.

CPU has a latency oriented design (shown in Figure 1.2a). Its goal is to finish each thread as soon as possible. Due to the memory wall, accessing off-chip memory is becoming more of a bottleneck, especially for applications with lots of data access. To hide this memory latency, CPU adopts on-chip cache to store frequently used data. Cache works very well for applications with data locality because most of the access will hit in the cache. For applications with many conditional statements, single thread performance may suffer from pipeline flush penalty associated with branch misprediction. To solve this, CPU usually comes with very complicated branch predictor to increase prediction accuracy. To further reduce single-thread execution latency, CPUs pipeline is designed to be out-of-order, which utilizes instruction-level parallelism within each thread. While one instruction is stalled due to data dependence (e.g. waiting for data to come back from memory), other independent instructions can go ahead and execute. This helps to hide the memory access latency. Though all these techniques contribute to reducing single thread execution latency, they require a lot of extra hardware logic to implement the functionality, hence occupying a lot of on-chip area and consuming a significant amount of power, especially the big cache. Nowadays, with so many transistors on the die and consuming power, some of them even has to be turned off due to too
CHAPTER 1. INTRODUCTION

much heat dissipation. Because of this, it becomes more and more difficult to increase application performance using latency-oriented design while keeping the same power budget.

![HW Threads](image)

**Figure 1.1: CPU & GPU Hardware Threads and PC Trace**

In applications that expose significant parallelism the latency of an individual thread is less important, and the overall throughput (when the last thread finishes) is the crucial measure. In a throughput-oriented design, there will be massively parallel threads for one application (illustrated in Figure 1.1b). All these threads will execute the same code, but on different data (the PC trace also proves this). On hardware side (shown in Figure 1.2b), it will have many parallel hardware cores. Whenever running threads stall on long latency operations, such as memory access, the hardware scheduler will pick other threads to run to keep the execution unit busy. If there are enough threads, memory access latency can be entirely hidden. Different from the CPU design, context switch here happens in a much finer-grained granularity, i.e. in cycle level. With the help of large register file, all contexts reside in the register file. Hence, there’s no need to do the context switch. Instead of using cache, branch predictor, and out-of-order execution, throughput-oriented designs use the overlap between thread execution and memory access of different threads to hide latency. By doing so, the
space occupied by the big cache and branch predictor can be used to place more in-order cores on chip. Hence, more threads can run in parallel. Even though latency for each thread becomes much bigger, the overall throughput is much better compared to CPU. Moreover, without the big cache, branch predictor and out-of-order execution (all of them are very power-hungry), this design also consumes much less power, resulting in much better power efficiency.

However to benefit from this design, applications need to have a huge amount of inherent parallelism. One great example of throughput-oriented design is GPU, which is traditionally customized for running graphics applications. Graphics applications inherently have a lot of parallelisms as there is a large number of pixels in each image/frame. Also, operations on each pixel are usually similar and independent from each other. Besides graphics applications, many other applications also have inherent parallelism, such as particle simulation, weather simulation, etc. These applications are ideal candidates for GPUs acceleration. However, programming GPUs was not easy before since all objects need to be described as triangles and actions need to be converted to graphics operations on those triangles. This process is a very counter-intuitive and time-consuming. Later, with the introduction of general-purpose GPUs programming model such as CUDA [1] and OpenCL [2], we see a significant boost in GPUs adoption. Effectively, CUDA / OpenCL is an extension to high-level language such as C/C++, enabling programmers to use GPUs by writing their algorithms.
CHAPTER 1. INTRODUCTION

in C/C++. Even though it’s not pure C/C++ and requires extra efforts, it’s way easier than writing in “graphics language”. Since then, many general applications have seen significant speedup by GPUs acceleration as shown in [3]. Nevertheless, as more diverse applications are ported to GPUs, there are some challenges to get the most from GPUs. Following is a more detailed analysis of those challenges.

1.1 Problem Definition

Success of application acceleration is continuously attracting more and more general purpose applications to use GPU, particularly computer vision and machine learning. These applications have a significant amount of parallelism and repeat the same operations over many data. For example, in computer vision applications, the same operations will be executed on each consecutive frame in the video. These applications are very computation intensive as it needs to extract information, build the model, and update the model on the fly from the raw data. Meanwhile, accessing input data and updating model also incur a lot of memory accesses. Furthermore, input frames are getting higher and higher resolution, which further increases both the computation demand and memory demand.

Due to the algorithm complexity, these applications need to perform many different tasks. Therefore, they tend to have larger code sizes, hence larger binary sizes. As the algorithms involve more intelligent decision making based on the condition of input data, they exhibit more control flow diversity. These control flows also make the memory access more irregular as code in different branches will access different regions of memory. Branch divergence hurts performance because GPUs execute programs in a manner like SIMD (Single Instruction Multiple Data) which means that every thread will do the same thing but on different data. When divergence happens, to preserve correctness, both paths will be executed, but threads only commit results on their own paths. That means many threads are doing useless work when divergence happen.

Besides branch divergence, memory divergence also hinders performance. As GPUs work in SIMD mode, many requests will be generated when a memory access instruction is being executed. If these requests access locations that are next to each other, they coalesce into a fewer number of requests (ideally only one request is needed) to the first level cache. Otherwise, a huge amount of requests will be sent to the memory subsystem (also called memory divergence). With more requests, more conflicts occur in the memory subsystem, especially in the data cache. With so many threads running concurrently on GPUs and rapidly switching, cache thrashing is more likely to
happen. Cache thrashing occurs a lot in computer vision and machine learning applications since their memory access patterns are often irregular.

Overall, we see that these applications run very inefficiently on GPUs since there are many branch divergences, memory divergences and conflicts in shared resources during GPUs execution. Therefore, to fully utilize GPUs programmers need to be aware of the underlying architecture while designing and implementing their algorithms. At the same time, GPUs designers also need to optimize the architecture to support better execution of more general purpose applications.

1.2 Contribution

In order to streamline the application development and improve GPUs performance. This thesis makes contributions in two aspects: programming level and architecture level. From the programming level, the thesis demonstrates some general and algorithm-specific optimizations for a computer vision application that can improve performance significantly and these optimizations are generally applicable to other applications. From the architecture level, this thesis first identifies the importance of thread-level similarity and defines a new metric Warp Progression Similarity (WPS) to capture the similarity. Then different contributors to WPS are studied. Finally, a WPS-aware scheduler is proposed that can achieve better resource usage and higher performance.

In more detail, this thesis makes the following contributions:

1.2.1 Programming Level

Background subtraction is an essential first stage in many vision applications differentiating foreground pixels from the background scene, with Mixture of Gaussians (MoG) being a widely used implementation choice. Due to this algorithm complexity, MoG is highly computation intensive. MoG’s high computation demand renders a real-time single threaded realization on CPU infeasible. With its pixel level parallelism, deploying MoG on top of parallel architectures such as a Graphics Processing Unit (GPU) is promising. However, MoG poses many challenges such as a significant amount of control flow (potentially reducing GPUs execution efficiency) as well as a large memory bandwidth demand.

In this thesis, a GPU implementation of Mixture of Gaussians (MoG) is proposed that surpasses real-time processing for full HD (1080x1920, 60 Hz fps). This thesis describes step-wise optimizations starting from general GPUs optimizations (such as memory coalescing, computation &
CHAPTER 1. INTRODUCTION

communication overlapping), via algorithm-specific optimizations including control flow reduction and register usage optimization, to windowed optimization utilizing shared memory. For each optimization, this thesis evaluates the performance potential and identifies architectural bottlenecks. Our CUDA-based implementation improves performance over sequential implementation by 57x, 97x and 101x through general, algorithm-specific, and windowed optimizations respectively, without impact to the output quality.

1.2.2 Architecture Level

When an application runs on a GPU, many warps will be launched. All the warps will execute the same code and start from the beginning. After some time, there will be a divergence in execution progress (which we call inter-warp divergence) among all the warps. In other words, some warps are far ahead while some others are lagging behind. The inter-warp divergence happens mainly because uneven memory access latency (whether data is in the cache or not) and branch divergence. This divergence has significant performance impact on GPUs as it can bring more conflicts to shared resources such as D$ and I$. Previous work [4, 5, 6, 7, 8, 9, 10] also observes conflicts in the memory subsystem, in particular, the data cache. [7, 8, 9, 10] try to change thread scheduling policy as such to preserve data locality. However, they overlook execution locality between warps and its performance impact on shared resources especially instruction cache, which is a huge victim as it’s directly related to inter-warp divergence.

To fully understand the performance impact, this thesis quantitatively studies the benefits of inter-warp divergence aware execution on GPUs. To that end, the thesis first proposes a novel approach to quantify the inter-warp divergence by measuring the temporal similarity in execution progress of concurrent warps, which we call Warp Progression Similarity (WPS). Using this metric, we analyzed all the factors that can influence WPS including algorithm-intrinsics, bounded cache, scheduling policy and number of schedulers. The result shows that among all the factors, scheduling policy is the biggest one. Then, based on the WPS metric, this thesis proposes a WPS-aware Scheduler (WPSaS) to optimize GPUs throughput. WPSaS will slow down ahead warps and speedup lagging warps by taking into account the age of each instruction cache block. The goal is to manage inter-warp divergence to better hide memory access latency and minimize resource conflicts and temporal under-utilization in compute units allowing GPUs to achieve their peak throughput. Our results demonstrate that WPSaS improves throughput by 10% with a pronounced reduction in resource conflicts and temporal under-utilization.
Chapter 2

Related Work

In this chapter, existing work on GPU acceleration of background subtraction will be discussed first. Next, previous work on how to improve thread scheduling on GPU will be compared and analyzed.

2.1 Background Subtraction on GPU

With GPUs evolving to support general-purpose applications, researchers have started to adapt vision applications to GPU platforms [11] to improve throughput and efficiency. A subset of OpenCV vision APIs [12] have been realized in CUDA for parallel execution [13]. Key algorithms including image segmentation [14], feature extraction [15], object detection and tracking [16] have been mapped to GPUs platforms. Most of these approaches mainly rely on general GPU optimizations (see Section 4.2). Our techniques presented in this thesis can help to identify opportunities for algorithm tuning with respect to GPU architecture to further improve performance.

Few research teams focused on a GPU-parallel realization of MoG [17, 18, 19, 20, 21]. However, these approaches mainly focused on general GPU optimizations largely independent of the MoG algorithm. Thus, these approaches achieve a limited speedup of about $20\times$ over serial CPU execution. The optimizations applied are almost similar across these approaches.

A common optimization for GPU is overlapping data transfer and kernel execution. In [17, 18], the input frame and output foreground pixels are streamed using DMA in parallel to executing the kernel. However, compare to [18], [17] imposes much more traffic to the system bus, as Gaussian parameters need to be fetched and written back to the main memory per frame execution. For each pixel, MoG reads the Gaussian parameters (with up to 120 bytes assuming 5 Gaussian
CHAPTER 2. RELATED WORK

with three double-precision parameters per Gaussian) and writes them back after updating. Three of the four approaches (except [17]) store the Gaussian parameters in GPU global memory avoiding repeatedly transferring Gaussian parameters back and forth.

Many approaches [17, 18, 20] employ memory coalescing to increase memory access efficiency when accessing GPU memory. Furthermore, memory coalescing potentially reduces the volume of the transaction to GPU memory, utilizing GPU memory bandwidth. However, even in a perfect GPU memory utilization, updating of Gaussian parameters is still subject to memory access latency, limiting the GPU utilization. Some other approaches [17, 18, 20] improve efficiency by utilizing shared memory to keep part of the Gaussian parameters on-chip avoiding off-chip access and its latency. However, [17, 18, 20] did not reveal shared memory details (e.g. amount) and did not analyze that optimization individually for its performance improvement potential. Our work specifically isolates the effect of shared memory, investigates into the trade-off between segment size, window depth, and performance.

The approach [21] aims to reduce the amount of computation in MoG based on the algorithm in [22]. [21] eliminates Standard Deviation along with its associated costly operations and utilizes a variable number of Gaussian components per pixels. This boosts the performance at the cost of quality loss. Overall, using a variable number of Gaussian components seems a promising for a CPU-bound MoG realization. However, when targeted to a GPU, it may only yield limited benefits. The parallel threads in a GPU execute in lock-step mode. All threads perform the same amount of computation even with a variable number of Gaussian components. In result, the thread with the most Gaussian components determines the latency of all parallel threads. Furthermore, an unbalanced memory access pattern within the parallel threads potentially reduces the memory access efficiency, limiting the achievable speedup.

Overall, the previous approaches focused on GPU general optimizations not utilizing algorithm-specific optimization opportunities. This leads to limited speedup over serial implementation. In contrast, our work further expands the optimization potential including register usage reduction and reducing MoG branch divergence which requires thorough algorithm understanding. Also, our step-wise optimization provides insight into the potential of each optimization, highlights architectural features involved, and thus guides future parallelization efforts.
CHAPTER 2. RELATED WORK

2.2 Thread Scheduling on GPU

Though there is lots of locality analysis of data cache on GPU, only a few work targets instruction cache. [23] observed that on deep-multithreaded GPUs, there exists inter-warp instructions temporal locality, resulting in much instruction fetching redundancy among different warps. To remove this redundancy, they proposed a filter cache to store recently fetched instructions. The filter cache achieves 60% hit rate for benchmarks that have more regular control flow, resulting in up to 19% energy reduction in the front-end.

Contrary to the assumption that running more threads on GPU would be better, [4, 5, 6, 7, 8, 9, 10] have all observed that in many applications, fewer threads have better performance since more threads will introduce additional conflicts in the shared resources such as cache, interconnection, and DRAM. To understand the performance impact of the number of concurrent threads, [4] developed an abstract model to study the behavior of parallel workloads on architecture that combines both caching and multithreading. Their results show that there will be a performance “valley” when adding more threads will hurt the performance as cache size is not big enough to capture all accesses. As a result, too many requests contend with each other for the cache which leads to cache thrashing. [5] used a detailed simulator to study the characteristics of applications on GPU. They came to similar results that reducing concurrency can yield better performance for applications such as Advanced Encryption Standard (AES) [24], MUMmerGPU (MUM) [25].

One way to solve this problem is to enhance the scheduler to find the right number of threads to run. [6] proposed a dynamic cooperative thread arrays (CTA) scheduling mechanism to allocate the optimal of CTAs by monitoring two key metrics of memory systems: C-idle, idle cycles when cores are not utilized; C-mem, stalled cycles where all CTAS are waiting for data. [7] explained that commonly used instruction fetch scheduling such as round-robin give equal priority to each warp, making all warps reach the same long latency operations at the same time. Therefore, GPU is unable to hide memory latency. To solve this problem, they propose a 2-level warp scheduling where all warps are split into fetch groups, prioritize warps from a single group first until long latency operations are reached, then switch to another group. To get resource contention information, [8] used victim cache tag to capture the locality lost for each warp, and give high priority to the warp that loses the most locality. [10] identifies the correlation between branch divergence and memory divergence, then use the active mask in the branch to predict the number of cache blocks needed by each warp, use it to guide the scheduler to throttle warps. [9] also introduces a 2-level scheduler, but in CTA granularity, besides that, the scheduler will prioritize non-consecutive CTA groups on
CHAPTER 2. RELATED WORK

different cores to reduce contention on memory banks.

Apart from using scheduling to solve the resource contention issue, another way is to change the cache management. \cite{26} introduced a memory request prioritization buffer (MRPB) to do request reordering and cache bypass. Reordering requests from the same source (such as warp, block) together can produce a more cache friendly access stream, therefore, retain the intra-warp locality. Instead of memory request reordering, \cite{27} and \cite{28} divide threads into cached group and bypass group and adjust the number in each group dynamically to make sure the working set won’t exceed the cache capacity.

Different from all previous work, this thesis identifies that I-$ is also one of the major victims of the conflict brought by too many threads running simultaneous, and I-$ performance has a significant performance impact.
Chapter 3

Background

To put the contributions of this thesis into perspective, this chapter will first introduce GPU programming model and how it supports general purpose programming. Then GPU architecture and how the software is executed will be explained to understand some optimization challenges on GPU. After that, GPGPU-Sim, a cycle accurate GPU simulator, will be explained. In the end, it’s a detailed introduction of Mixture of Gaussians (MoG), a frequently used background subtraction algorithm.

3.1 GPU Programming

There are two languages available to program GPU: CUDA and OpenCL, both support a very similar heterogeneous programming model. Each program written in this model consists of host code and device code (also called kernel). The host code runs on CPU and is responsible for program initialization and data transfer between host and device. Device kernel is executed on GPU and responsible for the acceleration of compute-intensive task. Typically, the host code will do some initialization including input data setup and transferring input data to device memory. Then it calls the device kernel. Device kernel will start executing once data is ready in device memory. When the device kernel finishes, the control is returned to the host, which will also transfer the results back to host memory. An example is shown in Algorithm 1: cudaMemcpy is used to copy memory between host and device. The direction of copy is configured by the last argument, either cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost. Dim3 is a custom data structure to hold the thread hierarchy information. To call the kernel, besides the input arguments, thread hierarchy also needs to be specified (surrounded by triple angular brackets)
Algorithm 1 Sample CUDA Host Code

1: function SAMPLE_CUDA(in input, out output) 
2:     cudaMemcpy(input_d, input, memSize, cudaMemcpyHostToDevice) 
3:     dim3 dimGrid(NUM_BLOCK, 1) 
4:     dim3 dimBlock(numThread, 1) 
5:     mix_kernel <<<dimGrid, dimBlock>>> (input_d, output_d) 
6:     cudaMemcpy(output_d, output, memSize, cudaMemcpyDeviceToHost) 
7: end function

Each device kernel will run many threads, which is configured by the programmer. As shown in Figure 3.1, each kernel has a grid of threads. The grid consists of many thread blocks, which can be up to three dimensions. Each block has a unique ID. Each block can also be multi-dimensions, composed of threads. Each thread also has a unique ID. All threads will run the same code but operate on different data. Typically the execution order of each thread is non-deterministic, but CUDA/OpenCL provides intrinsics for synchronizing threads within each block.

Variables in device kernel can reside in different parts of the memory hierarchy given by the programming model (shown in Figure 3.1). Global memory is shared by all threads but its accessing speed is very slow. Shared memory is shared by all threads within each thread block. Since it’s implemented by hardware on-chip cache, access is very fast. To use it, the programmer needs to first explicitly declare shared memory variable, then fetch data from global memory to shared memory. When the processing finishes, data needs to be stored back to global memory. Otherwise, the data will be gone as the life time of shared memory variables is the same as the corresponding thread-block. Local memory is private for each thread, so usually is used for store per-thread data. It’s backed by global memory.

3.2 GPU Architecture

3.2.1 Overview

Modern Graphics Processor Units (GPUs) platform offer a clustered-SIMD architecture with massive thread-level parallelism to support data-intensive high-throughput applications. Figure 3.2 conceptually highlights the main architecture elements of an NVIDIA GPU. For our experiments, we use an NVIDIA Tesla C2075 [29]. Table 5.1 compares its main architecture features against the selected CPU (Intel Xeon E5-2620 [30]). The Tesla C2075 consists of 14 Streaming
CHAPTER 3. BACKGROUND

Multiprocessor (SM) units. Each SM contains 32 cores (compute units), a shared scheduler, 48KB shared memory, and a large 32K register file (32-bit each) that is shared among the cores in the SM.

Figure 3.2: GPU Architecture Overview

Cores within an SM execute in SIMD fashion. They share a common Program Counter (PC) and are controlled by a common scheduler (two schedulers per SM). Multiple threads (up to 1536) are scheduled in HW by the SM scheduler and a large register file facilitates zero-overhead context switch. This allows hiding the high latency of off-chip access by rapidly switching to another
Table 3.1: HW Configuration

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Intel Xeon E5-2620</td>
<td>NVIDIA Tesla C2075</td>
</tr>
<tr>
<td>Cores</td>
<td>6</td>
<td>448</td>
</tr>
<tr>
<td>Frequency</td>
<td>2.5 GHz</td>
<td>1.15 GHz</td>
</tr>
<tr>
<td>FLOPS (single)</td>
<td>120.3 GFLOPS</td>
<td>1.03 TFLOPS</td>
</tr>
<tr>
<td>FLOPS (double)</td>
<td>515 GFLOPS</td>
<td></td>
</tr>
<tr>
<td>Cache</td>
<td>L2 (256K), L3 (15M)</td>
<td>L1 (16/48K), L2 (768K)</td>
</tr>
<tr>
<td>Memory BW</td>
<td>12.8GB/s (DDR3)</td>
<td>144GB/s (GDDR5)</td>
</tr>
</tbody>
</table>

ready-to-run thread. Shared memory offers some on-chip fast access storage for shared data among parallel threads within an SM. The explicitly controlled shared memory helps to reduce accesses to the slower off-chip memory.

### 3.2.2 CPU-GPU Interaction

Figure 3.3 conceptualizes integration of a CPU and a discrete GPU interconnected via the system bus (PCIe). More tightly integrated solutions exist (e.g. AMD’s APUs), but high-end (high-performance) GPUs are discrete. In the discrete scenario, CPU and GPU have own memories with dedicated links. The GPU does not have access to the CPU’s memory. To transfer the necessary data to / from CPU memory and GPU memory, the CPU initiates DMA transactions before and after GPU kernel execution. Modern GPU typically supports asynchronous concurrent execution, i.e. computation on the device and memory transfer between host and device can be performed in parallel.

### 3.2.3 Single Instruction Multiple Thread

GPU supports Single Instruction Multiple Thread (SIMT) execution model which is very similar to SIMD. A group of threads will share the same instruction, but operate on different data. In doing so, the overhead of instruction fetch and decode can be amortized across the whole groups of threads. In NVIDIA GPU, this group of threads is called a warp, which usually consists of 32 threads.
CHAPTER 3. BACKGROUND

In AMD GPU, this group of threads is called a wavefront, which usually consists of 64 threads. Different from traditional SIMD, SIMT also supports control flow. That means threads in the same warp/wavefront can execute control flow instructions and go to different paths. However supporting control flow also comes with additional cost. If threads within the same warp go to different paths, it’s called branch divergence. When that happen, all threads within the same warp / wavefront will execute both paths, but each thread only commits its result in the path that belongs to it. As a result, branch divergence causes inefficiency in GPU execution because some threads are doing unnecessary and useless work. This not only increases the execution time but also adds to the overall power consumption.

3.2.4 Memory Hierarchy & Divergence

Figure 3.4 shows the memory hierarchy from the hardware perspective. On the top, every SM has its register file (RF). Local variables usually reside in RF and latency is a few cycles. Next is the L1 memory, which is divided into two parts: data cache and shared memory. Data cache is transparent to the programmer and can benefit application with data locality. Shared memory is similar to the cache, but it’s designed to be managed by programmers since they know what data exhibits locality. Data stored in shared memory is only shared by threads within the same thread group. L1 memory is private to each SM. In the next level is the L2 data cache that’s shared by all SMs. The last level is the off-chip global memory. It’s usually big, so is the access latency.

When a warp executes a memory instruction, all threads within this warp will issue requests. Hence, there will be 32 requests. If each request is asking for 4-byte data and they are accessing neighboring locations, they will coalesce into one request for 128-byte data. Then this request will be sent to L1 data cache. This is because typical block size in L1 data cache is 128 bytes. If the data is
in the cache, the whole block will be returned immediately. This is called coalesced memory access. However, if those 32 requests are not asking for neighboring locations, multiple requests will be sent to L1 data cache, which is called memory divergence. During execution, coalesced memory access is always desirable. Otherwise, there will be too many requests overwhelming the memory subsystem.

### 3.2.5 Occupancy

GPU uses hardware multi-threading to hide memory latency. Therefore, it’s better to have more threads running at the same time so as to keep the execution unit busy. Each generation of GPU has a hardware constraint that limits the maximum number of threads/warps/blocks that run at the same time in one SM (shown in Table 3.2). Assume one application can run W warps per SM, and the maximum number of warps that the hardware can support is MAX, then the occupancy is \( W / \text{SM} \). It’s always ideal to reach the hardware limit. However, due to resource usage by each thread, applications usually can not reach 100% occupancy. One resource usage is the number of registers per thread. For example, in Fermi architecture, if one application allocates YY registers per thread, then only XX threads can be running at the same time, which is smaller than the optimal. Another resource usage is shared memory. Since shared memory size is a limit, if each block uses too much, then only a small number of blocks can be running at the same time. Typically, 70% - 80% occupancy is needed to achieve good performance. Therefore, the programmer needs to be aware of occupancy while developing their GPU applications.
CHAPTER 3. BACKGROUND

Table 3.2: Compute Capability for Fermi

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum number of threads per SM</td>
<td>1536</td>
</tr>
<tr>
<td>Maximum number of blocks per SM</td>
<td>8</td>
</tr>
<tr>
<td>Maximum number of warps per SM</td>
<td>48</td>
</tr>
<tr>
<td>Maximum number of 32-bit registers per SM</td>
<td>32 K</td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td>1024</td>
</tr>
</tbody>
</table>

**Potential Occupancy Limiter:**

Register Usage, SMEM Usage, Block Size

3.2.6 Scheduler

GPU is multi-threaded architecture. So the number of concurrent threads is much bigger than the amount of hardware resources available. In each cycle, a scheduler is needed to pick the next warp to use those resources. As shown in Figure 3.5, there are two types of schedulers in GPU pipeline: front-end scheduler and back-end scheduler. Front-end scheduler sits in the first stage of the pipeline and picks the next warp to fetch an instruction from I-$. Back-end scheduler sits in the issue stage and picks the next ready warp to execute. Typically, the policy adopted by front-end scheduler policy is Round-Robin, which fetches instruction for each warp one after another. Every warp has equal priority. In Fermi architecture, there’s one front-end scheduler and two back-end schedulers in each SM. Policies in back-end scheduler can have many different flavors. In the simulator used in our experiments, there are four different policies:

**Strict Round Robin (SRR):** SRR gives the same priority to all warps. All warps take turns to issue an instruction. Each time only one instruction will be issued. In case the current warp is stalled on certain long-latency operation, the scheduler will have to wait.

**Loose Round Robin (LRR):** LRR is similar to round robin which gives priority to each warp one after another, except that LRR will skip warps that are not ready (e.g. waiting for long latency memory operation). LRR is typically better than round robin as it avoids unnecessary waiting time.

**Greedy then Oldest (GTO):** GTO will keep on giving priority to one warp until it reaches long latency operation. Then it will pick the oldest warp. Warp’s age is counted from the time when the warp is assigned to the current SM. In case there are multiple warps with the same oldest age, warp with the smaller ID will be picked.
CHAPTER 3. BACKGROUND

Two Level: In this scheme, all warps are divided into fetch groups (FG). The scheduler will prioritize warps in the same FG until all warps are stalled, then it will continue to the next FG. Priority between FGs is round robin. Two Level is proposed to solve the problem that all the warps may reach long operation at the same time, leaving the computation resources idle.

![Schedulers in GPU](image)

Figure 3.5: Schedulers in GPU

For the backend scheduler, trying to pick the next warp to issue, it might find the warp being stalled due to the following reasons:

**Data Stall:** The warp is executing a memory access instruction, but the data is not in the cache. So the request needs to be sent to the global memory which usually takes hundreds of cycles to complete a transaction.

**Instruction Stall:** The warp cannot be issued since its next instruction has not been fetched yet.

**Hardware Stall:** The warp cannot be issued because the required hardware resources to execute this instruction is not available (as it’s currently being used by others). This is particularly true to the hardware unit that takes long cycles to complete such as special function unit.

**Synchronization Stall:** A warp is stalled waiting for other warps to synchronize. CUDA/OpenCL provides intrinsics for synchronizing threads.

### 3.3 GPGPU-Sim

GPGPU-Sim [5] is a performance simulator for contemporary GPU. It’s one of the widely used GPU simulators in academia. It simulates the compute pipeline, but not the graphics pipeline. It supports running both CUDA application and OpenCL applications. Simulation is configured through a configuration file. It has two simulation mode: functional simulation & performance simulation. Following we will first give an overview of the architecture supported by GPGPU-Sim. Then there’s an explanation about how CUDA applications are executed on the simulator. Finally, it’s a brief description of the simulation statistics output.
CHAPTER 3. BACKGROUND

3.3.1 GPGPU-Sim Architecture Overview

Figure 3.6 shows the overall GPU architecture simulated by GPGPU-Sim. It consists of many SIMT Cores connected through the network to the memory subsystem. SIMT core is an instance of Streaming Multiprocessor (SM) in NVIDIA GPU or Compute Unit (CU) in AMD GPU. Within each SIMT core, all the hardware units are modeled including pipeline, register file, caches, memory ports, etc. The memory subsystem is composed of many memory partitions which have atomic operations unit, last-level cache bank and off-chip DRAM channel. More details of the simulator details can be found at [32].

3.3.2 Running Application on GPGPU-Sim

In a normal process, when running a CUDA application, the host code will be executed by CPU, device code will call API functions that are handled by CUDA runtime library (as shown in Figure 3.7). The runtime library will then send the request to the underlying GPU to run the device code. To run CUDA applications on GPGPU-Sim, the simulator will first be compiled into a shared library which implements all the API functions supported by CUDA Runtime Library. Meanwhile, the OS environment will be modified accordingly so that the time when the application makes API calls, those calls will instead be intercepted by this shared library. The shared library will then initialize the simulator to execute the application.
CHAPTER 3. BACKGROUND

![Diagram](image)

Figure 3.7: API Call Interception

3.3.3 GPGPU-Sim Output

In performance simulation mode, GPGPU-Sim outputs many statistics that can help understand application’s performance. Table 3.3 shows some summary statistics including cycle time and instruction number for the current kernel as well as for all the kernels. Besides these, GPGPU-Sim also gives detailed performance number in each hardware unit such as cache, DRAM, interconnect, etc. In particular, performance penalty due to branch divergence is also included. All these numbers can help to figure out application’s performance bottleneck.

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gpu_sim_cyle</code></td>
<td>Number of cycles executed for the current kernel</td>
</tr>
<tr>
<td><code>gpu_sim_insn</code></td>
<td>Number of instructions executed for the current kernel</td>
</tr>
<tr>
<td><code>gpu_ipc</code></td>
<td>Instruction per cycle for the current kernel</td>
</tr>
<tr>
<td><code>gpu_tot_sim_cyle</code></td>
<td>Total number of cycles executed for all kernels</td>
</tr>
<tr>
<td><code>gpu_tot_sim_insn</code></td>
<td>Total number of instructions executed for all kernels</td>
</tr>
<tr>
<td><code>gpu_tot_ipc</code></td>
<td>Instruction per cycle for the all kernels</td>
</tr>
<tr>
<td><code>gpu_total_sim_rate</code></td>
<td>Simulation speed: <code>gpu_tot_sim_insn</code> / <code>gpu_wall_time</code></td>
</tr>
</tbody>
</table>
CHAPTER 3. BACKGROUND

3.4 Mixture of Gaussian

Background subtraction can be realized with different algorithms (see an overview in [33]). Among the options, we selected Mixture of Gaussians (MoG) [34]. MoG offers very good quality and efficiency in capturing multi-modals background scenes. Furthermore, MoG offers a limited storage requirement since it updates a unified model of the background scene without the need to store a history of old frames. MoG is widely used for deployments with fixed camera position.

Figure 3.8 illustrates a coarse-grained flow of the MoG. Each pixel’s background value is modeled by three to five Gaussian Components (distribution). The pixel’s Gaussian Components are updated accordingly, based on learning factor, taking into account a new incoming pixel value. If none of the Gaussians sufficiently matches the new pixel value, that pixel is declared as a foreground. Otherwise, it’s a background pixel. Each Gaussian Component consists of three parameters: (1) Standard Deviation (sd), (2) Mean (m) and (3) Weight (w).

Algorithm 2 outlines a serial MoG implementation (see details in [35]). The input is a video frame and Gaussian parameters. The algorithm loops through all pixels in the frame (lines 2–29). For each pixel, the algorithm first classifies the pixel’s Gaussian components into match or non-match components (line 5). A component matches if the component’s mean is within a certain range $\Gamma_1$ of the current pixel value. Gaussian parameters are updated based on match classification. In case that no match exists (line 12), the algorithm creates a new Gaussian component, called virtual component, and replaces the virtual component with the Gaussian component with the smallest weight value (line 14). Then, the components are ranked based on their weight over standard deviation ratio (line 16) and sorted by rank (line 19). Starting from the highest rank component, the algorithm declares a pixel to be background based on rank and closeness in matching with the current value (line 23). When finding the first match, the comparison stops and the algorithm continues with the next pixel.
Algorithm 2 MoG Pseudo-code

1: function MoG (in Frame, inout Gaussian, out Foreground)
2:     for $i = 0$ to numPixel do
3:         for $k = 0$ to numGau do
4:             $diff[k] = abs(m_k - pixel)$
5:             if $diff[k] < \Gamma_1$ then \(\triangleright\) Match
6:                 update $w_k$, $m_k$ and $sd_k$
7:                 $match = 1$
8:             else \(\triangleright\) non-Match
9:                 update $w$
10:             end if
11:         end for
12:         if !$match$ then
13:             Create virtual component
14:             Find the smallest $w$
15:         end if
16:         for all gaussian do
17:             Calc Rank for gaussian
18:         end for
19:         for all gaussian do
20:             Sort Component by Rank
21:     end for
22:     Foreground = 1
23:     for $k =$ HighestRank to 0 do
24:         if $\omega_k \geq \Gamma_2$ \&\& $diff[k]/sd_k < \Gamma_1$ then
25:             Foreground = 0
26:         break
27:     end if
28:     end for
29: end function
CHAPTER 3. BACKGROUND

A considerable computation and communication demands are required to realize MoG for full-HD (1080p 60Hz), which exceeds the capabilities of current CPUs. To realize real-time processing, acceleration is necessary. MoG loop iterations over pixels are independent. Thus, each pixel could be operated on in parallel. Hence, MoG is embarrassingly parallel. A GPU with the massively parallel compute units is a suitable target platform.
Chapter 4

High Performance Background Subtraction

This chapter first describes the base implementation and then our step-wise optimizations for MoG background subtraction on GPU platform. For each optimization, the efficiency is analyzed correlated with underlying hardware characteristics. The optimizations are grouped to general GPU, algorithm-specific, and shared memory optimizations. To motivate and assess benefits of individual optimizations, the evaluation results are interleaved throughout this section.

4.1 Base Implementation

The base implementation is almost a direct translation of the sequential implementation into CUDA (version 4.2[36]) targeted for execution on NVIDIA Tesla C2075 with Fermi architecture (see Table[5.1] for characteristics). For execution, the CPU transfers the input frame into GPU global memory, launches the MoG kernel, and transfers the foreground back. Gaussian parameters are initialized once by the CPU and then stored in GPU global memory. Gaussian parameters are updated throughout execution directly in GPU global memory thus do not affect the system bus (PCIs).

Each thread in the GPU operates on an individual pixel independently yielding 2 million threads. We select 128 threads per block which form a thread group. In the base implementation, each thread utilizes 30 registers (as per NVIDIA Visual Profiler, v4.2). To achieve the highest quality, we retain the double-precision floating point data type for Gaussian parameters from the sequential implementation. We initially consider 3 Gaussian components per pixel. In this paper, we measure
the execution time for processing 450 Full-HD frames (1080 × 1920) and report the achieved speedup of GPU execution over execution on the CPU.

A single-thread double-precision CPU implementation with maximum optimization (-O3 in GCC) requires 227.3s for the 450 frames, about 30× slower than real-time (assuming 60Hz). Customizing the code for SIMD operations reduces the MoG execution time by a factor of 0.28× (to 163s). The SIMD benefit is minimal due to many conditional branches in the MoG algorithm. Even a multi-threaded CPU implementation (8 threads, OpenMP) is with 99.8s far beyond real-time operation. For comparison in this paper, we chose the single core CPU implementation (-O3 optimization) as the reference point. The reported speedup can be scaled (divide by 2.3) to yield a relative comparison against the 8-core CPU implementation.

Our base implementation on GPU requires 17.5s, achieving a moderate speedup of 13× over the single core CPU implementation. The following sections introduce and evaluate our GPU optimizations to increase performance.

4.2 General Optimizations

Some general optimizations are suitable for almost any algorithm. This section applies to general GPU optimizations, namely memory coalescing and overlapped execution. For brevity of referencing, we refer to the optimizations by single letter as listed in Table 4.1.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Base Implementation</strong></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Memory Coalescing</strong></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Overlapped Execution</strong></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

**Memory Coalescing:** GPUs are designed with a simplified memory hierarchy and limited on-chip cache to increase power efficiency in comparison to general-purpose CPUs. In result, fewer accesses are masked by caches. Nonetheless, the local memory is exposed to the programmer, giving mapping flexibility. The Tesla GPU uses GDDR5 DRAM which is optimized for bandwidth (144GByte/s peak). Still, if hundreds of cores simultaneously initiate memory accesses, the memory can become the bottleneck. The MoG algorithm itself is bandwidth-hungry, requiring transferring
284 MByte (475 MByte) per full HD frame solely for updating the three (five) Gaussian parameters in double-precision.

GPU memory systems can combine (coalesce) memory accesses across threads in an SM to improve the memory access efficiency. Memory access efficiency is the ratio of data used (e.g. read from the core) over volume of data transferred from/to memory (into cache). Partially using the data in a cache line (e.g. through stride memory access) before the cache line is evicted again reduces the memory access efficiency. To increase efficiency, the parallel executing threads should ideally yield a contiguous memory access. Data structures have to be designed accordingly to yield coalesced memory accesses in order to increase memory access efficiency.

Figure 4.1 illustrates the data layout for three Gaussian components of different pixels that lead to non-coalesced and coalesced memory access assuming 3 simultaneous threads. Each thread first reads mean \( m \), then weight \( w \) followed by standard deviation \( sd \). In Figure 4.1a, Gaussian components are stored in an array, each containing structures with the individual parameters. Hence in memory \( m, w, sd \) are interleaved. Since all threads simultaneously first read \( m \), this leads to a stride access to memory. Due to cache size limitations, the cache line holding the data will evicted while all threads in a group read their \( m \). Although \( w \) and \( sd \) are also fetched from memory, the core could not use them in time. This reduces memory access efficiency and artificially increases memory bandwidth demands threefold.

Since the access sequence within each thread cannot be changed, coalesced access has to be achieved by optimizing the data layout. Figure 4.1b shows the optimized layout where the parameters of neighboring pixel’s Gaussian components are adjacent. Basically, each parameter \( (m, w, sd) \) is stored in its own array. Now, when all threads collectively read \( m \) the first Gaussian component, it results in a contiguous memory access – a coalesced access.
CHAPTER 4. HIGH PERFORMANCE BACKGROUND SUBTRACTION

Figure 4.3a illustrates the benefits of coalesced access in terms of memory access efficiency and memory store transaction. Memory access efficiency improves fourfold from only 17% in A to 78% with coalesced accesses in B. Similarly the number of memory store transaction drops from 13.3 million to 2 million. Memory coalescing increases the speedup threefold from $13 \times$ to $41 \times$ significantly improving performance.

**Overlapping Data Transfer and Execution:** Discrete GPUs do not have direct access to CPU’s main memory. Hence, the CPU initiates data transfers to / from GPU global memory via DMAs. These transfers introduce latency, which in case of limited computation but significant data movement can overshadow the benefits of GPU acceleration. Here the communication delays can even increase total run time over CPU execution [37]. To hide the transfer delay, data transfer and kernel execution can be overlapped.

Figure 4.2: Concurrency of Data Transfer & Kernel Execution.

Figure 4.2 illustrates the MoG execution phases both without and with overlapped transfers. In the base implementation, transfers and kernel execution occur sequentially: transfer frame to GPU, execute kernel, and transfer foreground frame back to main memory. Our profiling results show that almost one third of the total execution time is devoted to data transmission. The GPU is idle during this time, which reduces overall efficiency.

To hide transmission delays, data transfers and kernel execution have to be overlapped. Figure 4.2 plots the overlapped execution. While the kernel processes frame $i$, the DMA transfers the foreground image of the last frame ($i-1$) to main memory and the input data for the next frame.
(i+1) to GPU memory. The overlapped operation effectively hides the data transfer latency. To allow for concurrent data transfer and processing, we apply double-buffering to separate data that is being transferred and data being accessed by kernel. Overlapping transfers and execution increases the speedup to $57 \times$. Profiling shows that the execution time is now just bounded by the kernel execution time.

To reduce kernel execution time it is important to analyze the kernel’s resource utilization. Along these lines, Figure 4.3b shows how many registers are used per thread and the associated SM occupancy (ratio of active threads per SM over maximum possible number of threads). As a side effect of our memory coalescing optimization (B), each thread occupies 6 additional registers for temporary storage. Since all threads in an SM share the same register file, increasing the number of registers per thread can limit the number of concurrent threads. To further improve the performance, SM occupancy as well as thread execution efficiency need to be improved. This requires algorithm-specific optimizations described in the next section.

### 4.3 Algorithm-Specific Optimization

The general optimizations have already facilitated a $57 \times$ performance improvement. To further improve performance a thorough analysis of MoG algorithm is necessary. In result, we developed three algorithm-specific optimizations targeting the number of branches, predicated execution, and register usage. The optimizations are listed in Table 4.2. The next sections describe and analyze these optimizations.
CHAPTER 4. HIGH PERFORMANCE BACKGROUND SUBTRACTION

Table 4.2: Algorithm-Specific Optimizations

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch Reduction</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Predicated Execution</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Register Reduction</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

Eliminating Divergent Branches: Threads within a warp share the same PC counter and program code. This means that parallel threads within a warp execute the same path in lockstep mode. Thread-specific branches may cause branch divergence in a warp, and parallel threads execute different conditional paths. GPUs handle divergent branches through serial execution of all possible conditional paths among parallel threads. The effect of conditional execution is achieved by predicated execution. To guide the hardware, the CUDA compiler inserts a "Set SYNchronization (SSY)" instruction indicating the start of a section with branch divergence. GPU hardware sets an active mask for each thread according to its local branch condition. Only a thread set as active (path taken) commits the final results. The results of threads without the active mask set are discarded.

Branch divergence can lead to an increased thread execution time as all possible paths are executed sequentially. MoG (Algorithm 2) has a considerable number of branches that are thread (pixel) specific: checking the status of Gaussian components (line 5), creating virtual component (line 12), ranking and sorting of Gaussian components (line 16 19), and foreground detection (line 24).

In order to reduce the number of divergent branches, we propose MoG algorithm tuning to match it better to GPU execution characteristics. Ranking and sorting helps in CPU-based execution to eliminate some computation in later steps (i.e. at foreground detection). In CPU, since a Gaussian component with a higher rank is more likely to match the current pixel, the loop can be terminated early. Conversely, this typical CPU-bound optimization is inefficient for GPUs as it introduces divergence. To optimize for GPU execution, we replace ranking and sorting with an unconditional checking of all Gaussian components. Algorithm 3 and Algorithm 4 highlight our proposed algorithm tuning, comparing sort and no-sort versions. In the no-sort version, all Gaussian components are compared unconditionally (the order of finding a match does not matter here), which reduces the number of divergent branches.

Figure 4.4 visualizes the effect of our algorithm-specific optimizations to architecture characteristics. As shown in Figure 4.4a, the no sort optimization (D) reduces the number of executed
Algorithm 3 MoG excerpt (sort)
1: for all gaussian do
2:  Calc Rank for gaussian and Sort
3: end for
4: Foreground = 1
5: for k = HighestRank to 0 do
6:  if \( \omega_k \geq \Gamma_2 \&\& \text{diff}[k]/sd_k < \Gamma_1 \) then
7:     Foreground = 0
8:  break
9: end if
10: end for

Algorithm 4 MoG excerpt (no sort)
1: for \( k = 0 \) to numGau do
2:  if \( \omega_k \geq \Gamma_2 \&\& \text{diff}[k]/sd_k < \Gamma_1 \) then
3:      Foreground = 0
4:  break
5: end if
6: end for

branches from 6.7 million to 6.2 million per frame, and improves branch efficiency by 1.5%. Branch efficiency is the ratio of non-divergent branches over total number of executed branches. At the same time D uses 4 fewer registers (Figure 4.4c) which positively affects SM occupancy. As a side effect of eliminating divergent branches the memory access efficiency slightly improves (Figure 4.4b). By eliminating divergent branches speedup increases from 57x (in C) to 85x (in D).

**Source-level Predicated Execution:** The GPU compiler often expands (serializes) execution paths of branches and predicates the write back. Serializing branches effectively increases the number of wasted instructions. A further reduction in number of branches and more importantly divergent branches is necessary to further improve performance. Some branches can be avoided by source-level predication in cases the scope is beyond the reach of compiler optimizations.

The concept of source-level predicated execution can be applied for updating the parameters according to their component’s match status. Algorithm 5 and 6 compare non-predicated and source-level predicated code. In non-predicated code, the parameters \( (w[k], m[k] \text{ and } sd[k]) \) of a match component are updated (lines 3-6). For a non-match component only \( w[k] \) is updated (line 8). In
predicated code, the match status is directly used when updating all parameters. This unifies the execution path for updating parameters. As an example, when updating \( w[k] \), \((1 - \alpha)\) is multiplied by \( \text{match} \) effectively adding \((1 - \alpha)\) only if \( \text{match} = 1 \) (line 2). Similarly, \( m[k] \) and \( sd[k] \) are updated by \( f(tmp) \) and \( g(tmp) \) if \( \text{match} = 1 \) (line 4, 5), otherwise they keep their previous values.

Source-level predicated execution lets branch efficiency soar to 99.5% (Figure 4.4a). At the same time memory access efficiency (Figure 4.4b) peaks close to 100% efficiency dropping the number of transactions to GPU memory 1.70 million. All data fetched from GPU memory is used for computations. The combined branch reduction and source-level predicated execution improves the speedup to 86x. Pushing branch and memory efficiency to their peak comes at a cost of one additional used register (Figure 4.4c) and slightly lower SM occupancy.

**Register Usage Reduction:** To maximize performance, as many threads as possible should execute concurrently, resulting in higher SM occupancy. One key parameter is the number of used register per thread determining the maximum number of threads on an SM. While more registers per thread may improve the latency of individual thread execution, it potentially reduces
CHAPTER 4. HIGH PERFORMANCE BACKGROUND SUBTRACTION

Algorithm 5 non-Predicated Execution
1: \textbf{for} \textit{k} = 0 \textbf{to} numGau \textbf{do}
2: \hspace{1em} \textbf{if} match \textbf{then}
3: \hspace{2em} w[k] = \alpha \ast w[k] + (1 - \alpha)
4: \hspace{2em} \textit{tmp} = (1 - \alpha)/w[k]
5: \hspace{2em} m[k] = f(\textit{tmp})
6: \hspace{2em} sd[k] = g(\textit{tmp})
7: \hspace{1em} \textbf{else}
8: \hspace{2em} w[k] = \alpha
9: \hspace{1em} \textbf{end if}
10: \textbf{end for}

Algorithm 6 Predicated Execution
1: \textbf{for} \textit{k} = 0 \textbf{to} numGau \textbf{do}
2: \hspace{1em} w[k] = \alpha \ast w[k] + \text{match} \ast (1 - \alpha)
3: \hspace{2em} \textit{tmp} = (1 - \alpha)/w[k]
4: \hspace{2em} m[k] = (1 - \text{match}) \ast m[k] + \text{match} \ast f(\textit{tmp})
5: \hspace{2em} sd[k] = (1 - \text{match}) \ast sd[k] + \text{match} \ast g(\text{tmp})
6: \textbf{end for}

the number of active threads per SM. This poses a trade-off between single thread performance and collective performance.

Figure 4.4c shows the register usage and the corresponding SM occupancy across optimization steps. Optimization C (final general optimization) uses 36 registers per thread, resulting in a low SM occupancy of 52%. Divergent branch elimination used fewer local variables dropping to 32 registers per thread, improving to 61% occupancy. The case E (combined branch reduction and source-level predicated execution) slightly increases to 33 registers since it needs additional variables for predicated execution. In result occupancy drops to 56%. Case E focused on minimizing single thread execution time and achieved to improve overall performance. Nonetheless, there is significant room for improvement to reduce registers per thread and increase the number of concurrently running threads.

Reducing register usage per thread in cost of higher computation is worthy to allow SMs to work at their maximum occupancy. In other words arithmetic calculations are cheaper than occupying register. We employ this insight to improve SM occupancy. In the original MoG algorithm
variable \( \text{diff} \) holds the difference between current pixel value and \( \text{mean} \). \( \text{Diff} \) is used in two conditional statements (line 5 and line 24). To reduce register usage, the variable \( \text{diff} \) is replaced with the actual computation. This reduces to 31 registers, SM occupancy increases to 65% and speedup to 97\( \times \).

Figure 4.5a plots the speedup over all optimization steps. The combination of all algorithm-specific and GPU general optimizations (case F) results in a 97\( \times \) speedup over serial CPU execution. The algorithm-specific optimizations add 40\( \times \) speedup on top of general GPU optimizations.

Figure 4.5b summarizes the effect of all optimization levels (from A to F) on branch efficiency, memory access efficiency and SM occupancy. By combining both general and algorithm-specific optimizations, MoG touches close to 100% in memory access and branch efficiency. More importantly, SM occupancy improves from from 52% at the end of general GPU optimization (C) to 65% at the end of algorithm-specific optimizations (F). With this our MoG optimizations have achieved a very high overall GPU utilization.

### 4.4 Shared Memory

Each SM includes a fast on-chip memory (48 KB for NVIDIA Tesla C2075) for frequently accessed data that is shared across cores within the SM. The local memory is managed explicitly by the program, controlling data movement, access and lifetime. To offset the overhead of fetching data from global memory, only frequently accessed data should be promoted to local memory.
CHAPTER 4. HIGH PERFORMANCE BACKGROUND SUBTRACTION

![Figure 4.6: Frame Group Optimization for Share Memory](image)

The most suitable MoG data for shared memory are the Gaussian parameters. However, the required capacity for all Gaussian parameters (149 MB for full-HD resolution with 3 Gaussians and double precision) surpasses by far the shared memory capacity. In addition, the Gaussian parameters are not shared across threads as individual pixels are processed independently. MoG works on a frame basis, meaning that all Gaussian components of all pixels are updated before starting a new frame. In result, swapping Gaussian components in and out of the global memory adds additional overhead without improving performance. A mechanism is needed that Gaussian parameters are reused multiple times to warrant the effort of copying to/from shared memory.

![Figure 4.7: Performance Over Group Size](image)

To overcome the size limitation, we split frames into smaller pieces which we call tiles. Tiles are dimensioned so that the Gaussian parameters of a tile’s pixels fit into the shared memory resulting in a 640pixel tile size. We modify the MoG to operate on tile basis. To realize the benefits
CHAPTER 4. HIGH PERFORMANCE BACKGROUND SUBTRACTION

of multiple accesses, we order frames into groups. Our tiled MoG operates first on the same tile across each frame of the frame group before shifting to the next tile. Figure 4.6 highlights the basic idea of tiled MoG. The kernel first fetches the Gaussian parameters of the first tile to shared memory, processes the tile across all frames in the frame group updating the parameters in shared memory, and stores them back to global memory. The process repeats for all remaining tiles, before shifting to the next frame group.

We have analyzed the efficiency of our tiled implementation in dependency of the frame group size. Figure 4.7a illustrates that maximum speedup is achieved with a frame group of size 8, yielding 101x speedup over serial CPU execution. Further increasing the group size does not yield better performance. Analyzing SM utilization and memory access efficiency (Figure 4.7b) reveals the reasons for this effect. Switching to a tiled implementation even with frame size of 1 lowered the SM occupancy to 40% (compare with 62% in optimization F). With increasing group size occupancy linearly decreases further down to 38% for a group size of 32. Lower SM occupancy is due contention for shared memory capacity as it is shared across all threads within an SM. With each thread requiring more shared memory, fewer can run simultaneously. Furthermore, processing tiles across multiple frame reduces memory access efficiency. Figure 4.7a illustrates that memory access efficiency reduces significantly from more than 90% (group size of 1), down to less than 60% (group size of 32). In summary, a tiled implementation of MoG does not significantly improve performance (increasing from 97x to 101x). In addition, it leads to an increased latency until a frame is completely processed as frame group size increases.

4.5 Summary

In this chapter, several general and algorithm-specific optimizations techniques are presented and analyzed for MoG background subtraction algorithm. General optimizations are memory coalescing and overlapping between computation and communication. Algorithm-specific optimizations includes branch reduction, register usage optimization and windowed optimization using shared memory. Results show that significant speedup can be achieved by combining these techniques. However, the computation resources are still not fully utilized due conflicts in shared resources which is partly due to hardware unaware of the properties of running applications. Therefore, there’s a need for architecture optimization that can better manage shared sources that can reduce conflicts in order to further improve performance.
Chapter 5

Inter-Warp Divergence Aware Execution on GPU

To demonstrate this thesis’s contribution in architecture level, this chapter will first motivate the benefits of inter-warp divergence aware execution. Then this thesis continues with Section 5.2 presenting WPS metric Section 5.3 analyzes the architecture impacts onto WPS, and Section 5.4 shows our proposed WPSaS mechanism. The exploration and results are based on three divergence-sensitive benchmarks: MoG background subtraction in [38], histogram in Parboil [39], and DXTC in CUDA SDK [40]. All experiments are carried out on GPGPU-Sim [5] (v3.2.2) configured to NVIDIA Fermi architecture (Table 5.1 lists the simulator parameters). All applications are run at full capacity, i.e. launching the maximum possible number of blocks in each SM.

Table 5.1: GPGPU-Sim Configuration

<table>
<thead>
<tr>
<th>Number of Block / SM</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Warp / SM</td>
<td>48</td>
</tr>
<tr>
<td>Number of Scheduler / SM</td>
<td>2</td>
</tr>
<tr>
<td>Scheduling Policy</td>
<td>greedy-then-oldest</td>
</tr>
<tr>
<td>Data Cache / Shared Memory</td>
<td>48 / 16 KB</td>
</tr>
<tr>
<td>Instruction Cache</td>
<td>2KB, 128B block, 4-way, LRU</td>
</tr>
</tbody>
</table>
CHAPTER 5. INTER-WARP DIVERGENCE AWARE EXECUTION ON GPU

5.1 Motivation

GPUs offer a high throughput for applications with massively parallel threads. High throughput is achieved by hiding memory access latency across parallel threads. While some threads are waiting for their data or instructions, others can execute. To hide memory access latency, GPUs have dedicated hardware schedulers performing scheduling decisions at warp-level granularity. Warp (also called wavefront in OpenCL) is a group of threads sharing same Program Counter (PC) and executing in lock-step mode. At every cycle, the scheduler selects a ready warp (with data, instruction and compute resources) to run.

Concurrent warps may run unevenly, which we call inter-warp divergence. Figure 5.1(a) demonstrates active PC of concurrent warps in MoG application. We observe that while all warps flow almost similar sequence of instructions, they get diverge over the sequence of executions. Other GPU applications also share similar behaviors, especially, this is more pronounced in a kernel with larger size and frequently conditional branches. Inter-warp divergence is helpful as it opens more opportunities to overlap memory access and execution from different warps which can hide memory latency hiding. However, excessive inter-warp divergence may hinder GPUs to achieve their peak throughput. The major side-effects of inter-warp divergence are: (1) inter-warp conflicts in I$ and D$ when diverged warps compete for the same cache blocks [7, 8, 9] and (2) temporal resource under-utilization. e.g. when completed warps have to wait for uncompleted warps [41].

Figure 5.1 illustrates the correlation between inter-warp divergence and conflicts for an example of Mixture-of-Gaussians (MoG) background subtraction [38]. Figure 5.1(a) traces PC values over time for 28 concurrent warps on a single Streaming Multiprocessor (SM), and Figure 5.1(b) presents the corresponding I$ and D$ misses per 1000 cycles. Periods with high inter-warp divergence result in high misses in I$ and D$ (such as cycle 20000, 27000 and 36000). High inter-warp divergence means that most warps are executing distinct sections of the kernels. So warps will request different I$ blocks and D$ blocks, creating more conflicts. This is particularly the case for GPUs since the warp scheduling happens in cycle-level granularity. In other words, block that was just brought into the cache by one warp can be evicted very soon by other current warps. Since this block is still useful, it has to be fetched again in the future. When such eviction and re-fetch happens a lot, it’s called cache thrashing. Meanwhile, we see that even though all warps execute a similar PC sequence, they diverge as the execution progresses. Furthermore, while most of the warps have finished, few behind warps are still running (from cycle 40000 to cycle 46000). Such behavior can potentially end up in much temporal resources under-utilization as the resource allocation granularity.
is in block-level. In other words, resources assigned to the finished warps cannot be released unless all warps within the same block have finished.

![PC Trace of 28 Concurrent Warps](image)

![Cache Misses](image)

Figure 5.1: PC Trace and Cache Misses for MoG

Previous approaches aim to reduce inter-warp conflicts in D$ (e.g. [8, 9]) and temporal under-utilization (e.g. [41, 42]). However, they have less attention on quantifying the inter-warp divergence as well as reducing the inter-warp conflicts in I$. In general, the effect of resource conflicts is more pronounced in I$ due to conflicting fetch requests caused by diverged PCs.

To demonstrate the performance potential of improving I$, Figure 5.2a first shows the binary sizes of all benchmarks from Parboil [39], Rodinia [43], CUDA-SDK [40] as well as MoG, in ascending order. The red line in the figure filters out benchmarks with a binary size larger than 2KB. In total, 17 benchmarks are larger than 2KB. This biggest one, dwt2d, can reach to 16KB. Next Figure 5.2b presents the achievable speedup with an infinite I$ for all benchmarks (compared to 2KB I$). Many benchmarks (except a few) that are larger than 2 KB have significant performance improvement. Speedup varies from 6% to 48%. On average, the speedup is 20%. To explain why some larger benchmarks don’t achieve speedup, Figure 5.2c shows the contribution of instruction stall to the overall execution for all the benchmarks running with limited instruction cache size (configured as 2KB). Here instruction stall means that the hardware cannot issue new instructions if the next assembly instruction has not been fetched into the cache yet. We can see that for benchmarks
that are smaller than 2KB, instruction stall is almost negligible because those benchmarks can totally fit into the cache. So there won’t be any misses in the instruction cache except for the first time that the instruction if fetched. For benchmarks that are larger than 2KB, most of them see a significant percentage of instruction stall. Exceptions are lavaMD, heartwall, bisectKernel1 and bisectKernel2. These exceptions are exactly the benchmarks that don’t see speedup while running on infinite instruction cache.

From the description above, we see that many benchmarks have significant performance benefits by eliminating inter-warp conflicts. All these benchmarks have relatively large kernel size with frequent data-dependent branches. To reduce conflicts in I$, a straightforward approach is to increase its size. However, increasing cache size will also add to power consumption since the cache is very power hungry. What’s more, with GPU targeting more and more general purpose application (which tend to have large kernel size), even the increased cache may not fit the entire application. This motivates the need for approaches that manage inter-warp divergence, avoiding I$ conflicts for divergence-sensitive benchmarks without increasing I$ size.

This thesis introduces Warp Progression Similarity (WPS) to quantify the inter-warp divergence based on the temporal similarity in execution progression of concurrent warps. Based on the WPS metric, this thesis introduces the WPS-aware Scheduler (WPSaS) to manage/control inter-warp divergence. We anticipate a conceptual trade-off between inter-warp divergence and IPC (shown in Figure 5.3) for the divergence-sensitive benchmarks with high inter-warp conflicts in I$. A very low inter-warp divergence results in a lock-step execution across the concurrent warps. It limits the opportunity to hide the memory access latency. On the other hand, very high inter-warp divergence brings too many conflicts. While both extremes hinder GPUs to achieve their peak IPC, we anticipate that maximum IPC is achieved with a medium inter-warp divergence.

5.2 Warp Progression Similarity (WPS)

This section introduces and quantifies Warp Progression Similarity (WPS) as a metric for measuring uneven execution progress of concurrent warps. Warps that execute on a single SM share the same kernel binary code and run almost the same PC pattern. However, their execution progress may vary over time depending on scheduling decisions as well as the availability of data, instruction and computation resources. Warp Progression Similarity (WPS) measures temporal similarity in execution progress of concurrent warps. WPS has a reverse correlation with inter-warp divergence. A higher WPS indicates a lower inter-warp divergence.
CHAPTER 5. INTER-WARP DIVERGENCE AWARE EXECUTION ON GPU

Figure 5.2: Standard Benchmark Evaluation
CHAPTER 5. INTER-WARP DIVERGENCE AWARE EXECUTION ON GPU

![Figure 5.3: Conceptual IPC / Inter-Warp Divergence Trade-off](image)

Calculating WPS is an offline approach using PC traces of all warps. Figure 5.4 illustrates the overall flow of the approach. Firstly, the program kernel will be executed in GPGPU-Sim simulator to generate execution trace. Then the trace is fed into the WPS calculator to compute WPS. In more detail, the approach consists of two steps: (a) instruction profiling to capture execution traces per warp and (b) temporal similarity calculation using the captured traces. For profiling (a), we choose an I$ block granularity to capture execution trace as it matches with the granularity of instruction fetch in underlying architecture. Our profiler sequences the kernel binary in smaller blocks of 128 bytes each (typical I$ block size). During simulation on GPGPU-Sim, the profiler records the runtime trace (block number and access duration) of accessed blocks. Figure 5.5a highlights one example for three concurrent warps. Note that the progress of warps through the same I$ block may vary, and even the sequence of accesses may be different (due to conditional execution).

Temporal similarity calculation (b) uses the recorded traces above and computes the temporal similarity of warps as a value between 0 and 1. It is based on the cycle distance of accessing I$ blocks. To correlate distances to temporal similarity values, we use a Gaussian distribution as a decay function; shorter distances have high similarity closer to 1 and longer distances have similarities closer to 0 (presented in Figure 5.5b). The selection of Gaussian distribution is based on our I$ block lifetime analysis over the benchmarks (with a 400 cycles standard deviation). Our analysis shows that the lifetime of I$ blocks almost follows a random behavior, which can be approximately modeled by a Gaussian distribution. This suggests a non-linear decay function to compute the similarities of execution progress. In principle, a linear function also can be used to calculate similarities. However, we think that a Gaussian distribution provides a better approximation as it provides a better model to captures the warps execution on the underlying architecture. The small cycle difference to access an I$ block (cycle difference less than the standard deviation) results in a higher similarity in execution progress. On the other hand, big cycle difference (greater than the standard deviation) has a higher degradation impact on progression similarity.
Algorithm [7] outlines the temporal similarity calculation. For every cycle \( t \), function \( Calc-WPS \) returns a WPS value (Similarity) reflecting the temporal similarity between all warps at that cycle. At first, the algorithm computes the difference of the currently used block to when it will be used again by the target warp (line:7). Calculated distances are normalized through the decay function (line:8) and then added up to the cumulative normalized distance value (\( sum \)) across all concurrent warps of that cycle. Finally, the algorithm calculates final WPS, which is an average of normalized distances across all warps (line:14). Note that \( numWarp \) refers to the number of warps launched. Early finishing warps still contribute to WPS, using the last accessed I$ block for distance calculation.

To evaluate WPS, we applied it to MoG (highlighted in Figure 5.6a). The WPS value is averaged for every 1000 cycles to reduce noise. Comparing the WPS changes over time in Figure 5.6a with PC values in Figure 5.1a, we observe a tight correlation between WPS and inter-warp divergence. For example, cycles with higher variation in PCs and thus higher inter-warp divergence, also show lower WPS. Figure 5.6b also presents WPS for DXTC benchmark. For both benchmarks, WPS
varies significantly during execution. The variation is more pronounced toward the end, hinting an unbalanced execution. For both benchmarks, WPS value is approaching zero in the end, which is because WPS quantification is based on the total number of active warps so even though some warps finish earlier, we still contribute them in WPS calculation for remaining cycles till all warps finish. Furthermore, WPS quantification does not consider 0 for the warps that have already finished. Instead, it uses the decay function and cycle distances to calculate the progression similarity of currently executing warps with those who have finished.

5.3 WPS Contributors

Overall, both algorithm and architecture contribute to the WPS. Algorithms with frequently data-dependent branches tend to diverge across parallel warps. For each branch, some warps may execute only a single path as all threads are in that path; some other warps may execute both paths since branch divergence happens between threads within this warp. Warps that complete both paths...
CHAPTER 5. INTER-WARP DIVERGENCE AWARE EXECUTION ON GPU

Algorithm 7 Thread-group Temporal Similarity Calculation

1: function CALC-WPS(in t, out WPS)
2: float similarity_array[numWARP]
3: for i = 0 to numWARP - 1 do
4: sum = 0
5: for j = 0 to numWARP - 1 do
6: if i ≠ j then
7: distance = calc_distance(i, j, t)
8: decay = calc_decay(distance)
9: sum = sum + decay
10: end if
11: end for
12: similarity_array[i] = sum / (numWarp - 1)
13: end for
14: WPS = AVERAGE( similarity_array[ ] )
15: end function

usually run longer time. At the same time, the underlying architecture can cause further reduction in WPS. Looking at a single SM (highlighted in Figure 5.7), primary WPS contributors are I$, D$ and warp schedulers in the back-end for scheduling warps for execution. I$ and D$ can cause uneven access latency to memory hierarchy since for some warps their data might be in the cache while data for other warps are not in the cache. For schedulers, its policy has a direct impact on the WPS as it decides the execution priority of each warp. What’s more, to achieve maximum utilization of the compute resources, GPUs often employ multiple warp schedulers (e.g. two in Fermi and four in Kepler and Maxwell architectures), which further affects WPS.

For our WPS analysis, we focus on four major contributors: algorithm-intrinsic, the uneven latency of instruction/data access, type and number of warp schedulers. Figure 5.8 cumulatively illustrates the effect of the four major contributors for MoG with 28 concurrent warps running on a single SM.

1Memory system and shared execution units can also affect algorithm-intrinsic WPS. However, we consider minimal effect on WPS. Please note that, we need to consider a set of boundaries in the GPU architecture to maintain the GPU execution semantic. With unlimited shared execution units, there would be no need for warp scheduling. Furthermore, warp would lose its concept. Therefore, we preserved the number of execution units, even when calculating algorithm-intrinsic WPS. For the memory system, we have already consider infinite size for IS and DS. With this the effect of memory access latency for instruction and data will be small.
Algorithm-intrinsic: To remove the effects of architecture elements, the simulator is configured with infinite instruction cache and data cache. There’s only one scheduler per SM, with strict round-robin scheduling policy (SRR). Figure 5.8a shows the algorithm-intrinsic WPS for MoG algorithm. During the entire execution, WPS value is always very high. It varies from 0.8 to 1. Those small fluctuations mainly come from data-dependent branches. Overall, the algorithm has little impact on WPS. For other algorithms, such as graph traversal, the impact could be bigger since it’s more control oriented. As input data decides the behavior of data-dependent branches, it also has direct impact on WPS.

Bounded Cache: In this setup, both data cache and instruction are bounded (data cache is 48KB, instruction cache is 2KB), other configurations are the same as Algorithm-intrinsic. Figure 5.8b shows that compared to Algorithm-intrinsic, the bounded cache adds more fluctuation to WPS during the entire execution. Around cycle 23000 and 51000 the WPS can be as low as 0.6. At the end of execution, the WPS drops significantly.

Scheduling Policy: In this setup, all configurations are the same as Bounded Cache, except that four different policies are applied and compared. These four policies are SRR, Loose Round Robin (LRR), Greedy then Oldest (GTO) and Two Level (see Section 3.2.6 for more detail). Figure 5.8c shows that similar to SRR, LRR also adds some fluctuations to WPS during the execution. But LRR’s running time is shorter than SRR as it will skip warps that are stalled, which avoids unnecessary waiting cycles. Compared to SRR and LRR, GTO has a huge impact on WPS. WPS is very high at the beginning but drops significantly afterward. This is can be explained by the greedy policy that GTO gives priority to the same warp as long as it doesn’t stall. This gives some warps the opportunity to go far ahead than other warps. This makes the difference of execution progress between warps larger and larger. As a result, WPS is becoming smaller and smaller. Even though
Figure 5.8: MoG WPS Contributors
CHAPTER 5. INTER-WARP DIVERGENCE AWARE EXECUTION ON GPU

GTO has the lowest WPS, it still achieves the best performance among all the polices. This is because GTO opens more opportunities for memory access latency hiding. Similar to GTO, Two Level also reduces the WPS significantly. Two Level policy will divide warps into groups. Priority will be given to the same group unless all warps in that group are stalled. Then the priority continues to the next group in line. This is very similar to the greedy policy to GTO but applies it to a group. This gives some groups the opportunities to go far ahead, which significantly reduces the WPS. But at the same time, an obvious pattern can be observed that WPS will go up and down in a repeatedly way. The reason is that Two Level schedule groups in a round robin fashion.

Number of Scheduler: As GTO achieves the best performance, this setup chooses GTO but varies the number of schedulers per SM. Figure 5.8d shows that by adding another scheduler, there are more WPS variations as the schedulers work independently from each other. The variation is most noticeable in cycle 22000, 30000 and 38000. Also there’s higher WPS with two schedulers because the number of warps managed by each scheduler is less. Hence, there’s less divergence in each scheduler.

In a nutshell, the algorithm-intrinsic WPS is high with only small variations due to data-dependent branches in the algorithm. Even with bounded caches, WPS drops only a slightly due to uneven latency for accessing the data and instructions. The scheduler impact varies. Fair policies such as SRR and LRR have minimum effect on WPS. In contrast, greedy policies such as Two-Level, and GTO result in a much lower WPS. Nonetheless, GTO achieves the best performance as it opens more opportunity for hiding memory access latency. Adding a second GTO scheduler reduces divergence, yet it increases WPS variations as the schedulers work independently.

Among all WPS contributors, the warp scheduler is most influential. Aggressive warp schedulers, such as GTO, result in higher inter-warp divergence to hide memory access latency as it can potentially smooth out the access request to the data cache. However, the downside is that for a particular class of applications (divergence-sensitive applications), an excessive divergence can be destructive by causing inter-warp conflict in the instruction cache and temporal under-utilization hindering GPUs to achieve their peak throughput. We have observed a significant inter-warp divergence in other benchmarks including Histogram and DXTC. Novel solutions are required to control/manage the inter-warp divergence to hide memory access latency while reducing inter-warp conflicts. In other words, approaches are needed to leverage aggressive scheduling policies such as GTO and at the same time manage the inter-warp conflicts appeared in I$.
CHAPTER 5. INTER-WARP DIVERGENCE AWARE EXECUTION ON GPU

5.4 WPS-aware Scheduler (WPSaS)

This section introduces our proposed WPS-aware Scheduler (WPSaS) with the aim to demonstrate potential benefits when taking WPS into account for GPU operation. Firstly, the implementation of WPSaS will be explained, with a focus on the algorithm involved. Next, results will be presented to show the relationship between WPS and performance. Lastly, results will be provided to shed light on from what aspects WPSaS improves application performance.

5.4.1 WPSaS Implementation

WPSaS is an extended I-fetch scheduler in the front-end of an SM (illustrated in Figure 5.7). WPSaS will not directly modify the back-end schedulers (warp schedulers), but it just enhances the round-robin policy in the front-end. As shown in Figure 5.9, to facilitate WPSaS, every block in instruction cache will have an associated age counter which records how many cycles have passed since the last access. The counter will be reset every time the corresponding block is accessed. The value of the age counter indicates the locality potential of the current block.

Being an I-fetch scheduler, WPSaS can make global decisions across all concurrent warps independent from the number of back-end schedulers. Meanwhile, The decisions by the front-end (WLLaS scheduler) can implicitly affect the decisions by back-end scheduler as the warp schedulers will not schedule any warps with no available instruction. WPS calculation is an offline approach which needs to have the entire execution trace. But WPSaS, being an online approach, doesn’t have such trace. Therefore, WPSaS uses a coarse approximation of WPS to guide the scheduler. In essence, WPSaS avoids conflicting instruction fetch requests caused by warps which are too far ahead. Avoiding these requests effectively stalls execution of ahead warps by the back-end scheduler(s). This results in a more even progress across all warps.

Algorithm 8 illustrates the algorithm used by WPSaS which is an extension over a standard round-robin I-fetch scheduler. It has two main functions: (1) WPSaS and (2) DoesThrashICache. (1) WPSaS implements round-robin scheduling with an additional check DoesThrashICache. (2) DoesThrashICache receives the PC value and current warp index (cur-idx) and returns the status of the warp. DoesThrashICache returns TRUE for an ahead warp and FALSE otherwise. An ahead warp is a warp with a fetch request that evicts a cache block currently being used by other warps. If DoesThrashICache() returns TRUE (indicating an ahead warp), WPSaS() skips the fetch request.

To detect an ahead warp, DoesThrashICache first calls ICache_Probe(PC) (line 12) to probe the status of I$ for the new fetch request. If the requested PC is in the cache, ICache_Probe(PC)
returns NULL. In the case of cache miss, ICache_Probe(PC) returns the index of the block that would be evicted (victim block). Warp with a fetch request that results in a victim block is also called a trigger warp. A trigger warp can potentially be an ahead warp. The decision of being an ahead warp depends on the access history of the victim block.

To capture the access history, DoesThrashICache calls Inter_Warp_Age function (line 14). Inter_Warp_Age returns the cycle count of last accessing the victim block by any other warp (victim-blk-age). A large victim-blk-age is an indicator that the block is less active. Conversely, a short victim-blk-age indicates that the block has been recently used by other warps (an active block). Thr defines the maximum age for being an active block. Hence, an ahead warp is a trigger warp that evicts an active block (ie. victim-blk-age < Thr).

In a nutshell, DoesThrashICache identifies a non-ahead warp under two conditions: (a) there is no associated victim block to the PC fetch request (ICache_Probe(PC) returns NULL); (b) there is an associated victim block but it is not active (victim-blk-age > Thr); The open research question is to tune Thr to achieve maximum IPC. Low values of Thr neutralize the effect of WPSaS (Thr = 0 is equivalent to WPS-unaware scheduling). In other extreme, high Thr values force warps to execute in lock-step. In addition, a warp might request a victim block for a very long time, but is always denied access. If this is caused by synchronization between warps, deadlock would happen. Therefore, there’s a timeout value to set the maximum cycles that a warp will be stalled. After the timeout cycles, the warp will be able to evict the victim block.
Algorithm 8 The Procedure of WPSaS over I-fetch Scheduler

1: function WPSaS()
2:   while (1) do
3:       if warp[cur-idx].IBUF \neq \text{FULL} \&\& \neg \text{DoesThrashICache(PC)} then
4:           F_Request(warp[cur-idx].PC)
5:       break
6:   end if
7:   cur-idx ++
8: end while
9: end function

10: function DOESTHRASHICACHE(PC)
11:   victim-blk = ICache_Probe(PC)
12:   if victim-blk \neq \text{NULL} then
13:       victim-blk-age = Inter_Warp_Age(victim-blk)
14:       if victim-blk-age < Thr then
15:           return TRUE
16:       end if
17:   end if
18:   return FALSE
19: end function

5.4.2 Result: Tradeoff between WPS & Performance

To evaluate WPSaS, we enhance the round-robin I-fetch scheduler of GPGPU-Sim to WPSaS and configure GTO as the back-end policy. To show how WPS impacts performance, we sweep Thr from 0 to 4000 and measure the resulting WPS and IPC. Figure 5.10 shows the trend of IPC and WPS over different threshold values. Both benchmarks have a similar trend that the WPS value is pretty low at the beginning but slowly increases with bigger threshold value. After some threshold, WPS increases significantly. Then it slowly saturates at some point. For MoG, WPS value increases a lot during 1000 to 1500 threshold values. For DXTC, WPS value increases a lot during 1200 to 2000 threshold value. The highest WPS values for MoG and DXTC are 0.68 and 0.6 respectively. The corresponding IPC value also increases at the beginning, then reaches to an optimal
point. Later, IPC starts to drop when WPS gets too high. For MoG the optimal threshold is 1280, while for DXTC is 1800. During the increasing period, IPC doesn’t increment linearly. Instead, there are small fluctuations.

To see a more clear relationship between threshold and IPC, Figure 5.11 plots the measured IPC over WPS for MoG and DXTC. Figure 5.11 resembles the conceptual trade-off in Figure 5.3. For both applications, WPS generally increases by increasing $Thr$ value. However, the IPC reaches its maximum by WPS around 0.6 which corresponds to $Thr = 1040$ in MoG and $Thr = 1800$ in DXTC. IPC drops steeply with WPS more than 0.6.

![Ips vs Trehshold](image)

(a) MoG

![Ips vs Trehshold](image)

(b) DXTC

Figure 5.10: WPS / IPC Trade-off

For both applications, an optimal performance point can be reached by increasing threshold value. However, the threshold value to reach optimal performance is different across applications.
CHAPTER 5. INTER-WARP DIVERGENCE AWARE EXECUTION ON GPU

One of the major deciding factors is the number of active warps. With a higher number of warps, smaller variations in threshold value caused more changes in WPS and IPC. There are some other factors as well. This will be part of our future research work.

5.4.3 Result: How WPSaS Improves Performance

To illustrate the effects of WPSaS on inter-warp divergence and resource conflicts, Figure 5.12b plots the PC values for MoG application for the best found $Thr (Thr = 1040)$. Comparing WPSaS (Figure 5.12b) against WPS-unaware scheduler (Figure 5.12a), the execution progress is more even and coordinated in WPSaS. Even though there’s still divergence, it’s controlled to a suitable degree. At the same time, there is enough inter-warp divergence to hide the memory access latency. Figure 5.12c reveals the corresponding I$ and D$ misses. Both I$ and D$ misses are evenly distributed throughout the execution. Especially for I$ miss, those spikes (shown in Figure 5.12d) don’t exist anymore. The reduction in I$ misses is more pronounced as it is directly affected by WPSaS. For D$ miss, the trend is almost the same as WPS-unaware scheduler between cycle 0 to 34000. But after that, WPSaS only sees one period when the miss will go high and that period doesn’t stay long. With both reductions in I$ and D$, the execution time is also reduced in WPSaS execution from 46000 cycles to 41000 cycles.

To provide more insight, Figure 5.13 plots for the best found $Thr$, the IPC, temporal under-utilization, and execution stalls for all three benchmarks (MoG, Histogram, and DXTC).

Temporal underutilization happens when not all warps belong to the same thread-block finish at the same time. Since the resource allocation granularity in GPU is block-level, resources
Figure 5.12: Comparisons of PC Trace and Cache Misses for MoG with & without WPSaS
CHAPTER 5. INTER-WARP DIVERGENCE AWARE EXECUTION ON GPU

occupied by finished warps cannot be released or reused until all the warps within the same thread-
block have finished. [41] proposed a method to measure the temporal under-utilization in an SM. Its
basically a relative ratio of underutilized time over entire execution time by contributing the active
number of threads as well. For the purpose of measuring temporal under-utilization, we use the same
metric proposed by [41]. Figure 5.13b also compares the result against WPS-unaware scheduler.
Temporal under-utilization drops across all benchmarks. Histo’s underutilization is reduced by 8%.
The more pronounced reductions belong to DXTC (from 20% to 0.8%) and MoG (from 14% to 2%).
WPSaS helps MoG and DXTC to reach almost 100% utilization.

Figure 5.13a shows a pronounced increase in IPC, 10.6% on average (14.1% MoG, 9.5%
Histogram, 8.4% DXTC). IPC mainly increases due to less temporal under-utilization and fewer
conflicts in I$ and D$.

Figure 5.13: IPC, Temporal Under-utilization and Stall Comparisons
CHAPTER 5. INTER-WARP DIVERGENCE AWARE EXECUTION ON GPU

Figure 5.13c compares the number of stalls in the back-end. There are four kinds of stalls: instruction, data, compute resources and scheduling. Scheduling is a new type of stall introduced by WPSaS as it might stall the pipeline even if there are ready warps to execute. Other stalls are caused by lack of instruction, data or compute resources. WPS-aware scheduler is able to significantly reduce the back-end stalls, in particular stalls due to lack of instructions (45 K to 34 K for MoG, 68 K to 56 K for DXTC, and 39 K to 34 K for Histogram). IS stalls reduce significantly as our WPSaS denies the fetch requests of ahead warps which would evict useful IS blocks. This prevents cache thrashing from happening. What’s more, WPS-aware scheduler also reduces stalls due to lack of data (14 K to 10 K for MoG, 25 K to 14 K for DXTC, 12 K to 9 K for Histogram) even though it’s not directly designed as such. WPS-aware scheduler forces the execution of different warps to be more even and coordinated, which could potentially end up in better data access pattern. On the downside, WPSaS introduces additional scheduling stalls (sched.) when denying fetch requests. MoG and Histogram have a minimal sched. stalls. In contrast, DXTC has a larger number of sched. stalls which hint to a large number of inter-warp conflicts in IS.

Overall, our proposed WPSaS demonstrates the benefits of inter-warp divergence aware execution in GPUs. For the benchmarks under evaluation, WPSaS results in 10.6% speedup on average with 21% to less than 8% reduction in temporal under-utilization, and 23% reduction in back-end stalls. This thesis focuses on defining the WPS metric and highlighting the performance improvement opportunities when taking WPS into account for scheduling. Nonetheless, we anticipate an efficient implementation at the architecture level, as all information required for WPSaS (Algorithm 8) is already available in the front-end stage.

The primary aim of WPSaS is to demonstrate the potential of WPS-aware execution on GPUs. Therefore this thesis only focuses on three IS sensitive benchmarks. Nonetheless, we have executed all 10 applications listed in Figure 5.2b that has performance improvement potential, and WPSaS shows some performance gain for each application (no slowdown). As part of our future work, we are working on a more detailed analysis of WPS impact on a wider set of applications as well as architecture realizations of WPSaS. Additionally, the concept of WPS could also be applied to GPU performance models, and used to guide compiler and hardware optimizations that tradeoff WPS against other goals.

The architecture realization of WPS quantification (Algorithm 7) as well as WPSaS (Algorithm 8) are parts of the future work. WPS quantification is an oracle algorithm that needs the future information of warps execution, so the ongoing work is to figure out a way to predict WPS on the fly using only history information. For WPSaS, we observe minimal hardware overhead to
make them realizable in the underlying architecture. Furthermore, the proposed WPSaS scheduler is a preliminary approach to demonstrate the benefits of WPS-aware executions on GPUs. We are looking for more efficient and simpler approaches to manage inter-warp divergence with respect to our newly introduced WPS.

5.5 Discussion

As illustrated in Figure 5.2b, some applications can achieve significant performance speedup by having an instruction cache with infinite size, hinting that there’s a big potential through optimizing instruction cache usage. Later, this thesis analyzes inter-warp divergence and proposes WPSaS to manage this divergence to reduce conflicts in shared resources, especially instruction cache. To compare infinite cache size with the proposed approach, Figure 5.14 shows the speedup of both approaches against the default setting with 2KB instruction (round robin front-end scheduling and GTO back-end scheduling) on three applications. Both approaches are better than the default setting. For dxtc, WPSaS comes very close to infinite cache (reaching 97% of the performance). Surprisingly, for both MoG and histo, WPSaS beats the infinite cache.

![Figure 5.14: IPC Improvement Comparisons](image)

To understand why WPSaS performs better, Figure 5.15 lists the number of different stalls (instruction, data, hardware and scheduling stalls) for three settings. By using infinite cache, instruction stall for all three applications reduces significantly (especially MoG and dxtc), which is
to be expected since the entire binary can fit into the cache. So, there won’t be any conflict misses in instruction cache. Regarding data stall, infinite instruction has almost no impact for dxtc and histo. However, data stall for MoG is increased compared to the default setting. As explained in Section 5.3, the bounded cache is one of the WPS contributors. With infinite cache, ahead warps can go further easier since GTO is a greedy policy. This makes the inter-warp divergence more severe, hence creating more conflicts in the data cache. As a result, we see more data stall. For hardware stall, dxtc and histo don’t see any big difference, but MoG sees a big increase. Overall, using infinite instruction can significantly reduce instruction stall, even though causing data or hardware stall to increase a little. The impact of the reduction of instruction stall is much bigger, so we see a big performance improvement.

WPSaS reduces the instruction stall significantly for both MoG and dxtc. In particular for dxtc, the reduction is almost the same as using infinite instruction cache. However, the instruction stall reduction for histo is tiny, which is because for histo among all the stall reasons, instruction accounts only for 8.72%. So there’s not too much room for reduction. WPSaS also reduces data stall for all three applications, even though WPSaS is instruction cache oriented. It helps reducing data cache conflicts by creating more even access pattern. This hints to a future work to take data cache access info into account. Interestingly, WPSaS reduces the HW stall for all three applications as well. Though WPSaS helps reduce instruction, data, and hardware stalls, it also introduce some overheads: scheduling stall. For histo, the scheduling stall is almost negligible. For MoG, the scheduling stall is also not too much. But scheduling stalls for dxtc is a lot. This explains why WPSaS surpass infinite cache for both MoG and histo, but not dxtc.

5.6 Future Work

Following is a list of future work items for WPSaS.

**Input Data Sensitivity** Input data affects the behaviors of both data-dependent branches and cache data access, which decides inter-warp divergence. Therefore, for the same application, WPS and optimal threshold for WPSaS can be different when input data are different. To generalize WPSaS, the sensitivity of WPSaS against the input data must be studied.

**Problem Validation in Real Machine** This work originates from an observation from real GPU that instruction stall percentage is too high for certain optimized applications. However, all the
CHAPTER 5. INTER-WARP DIVERGENCE AWARE EXECUTION ON GPU

Figure 5.15: Stall Comparisons for MoG, dxtc and histo
experiments performed in this work is done in simulator. It would be more trustworthy to see similar results also in real machine.

**Possibility of decreasing WPS** The assumption of WPSaS is that there’s too much inter-warp divergence in applications that cause conflicts in shared resources. Therefore, WPSaS reduces divergence to reduce conflicts. However, for certain applications, it may be better to have more divergence. The current implementation of WPS doesn’t support reducing WPS yet.

### 5.7 Summary

In summary, this chapter proposes a new perspective on the efficiency analysis of GPUs by considering the role of inter-warp divergence on GPU execution. As a metric to quantify inter-warp divergence, this chapter proposes Warp Progression Similarity (WPS) to measure temporal execution similarity of concurrent warps. Based on the WPS metric, this chapter introduces WPSaS, a WPS-aware Scheduler, giving an outlook on performance improvement opportunities. WPSaS achieves 10.6% improvement in throughput for three selected benchmarks. We anticipate similar performance improvements with benchmarks which either suffer from inter-warp conflicts in I$, or with high temporal under-utilization. Nonetheless, many design opportunities but also challenges are opened by divergence-aware design principles, such as finding the balance between inter-warp divergence and inter-warp conflicts in I$ and D$.
Chapter 6

Conclusion

This thesis discussed and evaluated a set of optimizations for GPU-based implementation of a widely used background subtraction algorithm: Mixture of Gaussians (MoG). While previous approaches were limited to general GPU optimizations (such as including memory coalescing, data transfer and kernel overlap), the proposed optimizations expand to include algorithm-specific optimizations that require more thorough algorithm understanding: divergent branch elimination and register usage optimization.

Furthermore, we propose frame segmentation to take advantage of SM shared memory to keep Gaussian parameters on-chip. By employing all proposed optimizations, we achieved up to 101x speedup in comparison to serial CPU executions. Reaching this speedup required thorough understanding the algorithm and matching to the target architecture (algorithm/architecture co-design). We studied the effect of proposed optimizations over the foreground quality using MS-SSIM and ensured that our optimizations practically have no impact on quality (MS-SSIM reduces by less than 4%).

As a future work, we plan to realize MoG on an embedded GPUs due to the growing interest in adding vision capabilities in mobile devices. With the significantly lower compute power of embedded GPUs, achieving real-time performance will require to trade-off quality for speed.

To fully utilized the power of GPU, only program optimizations are not enough; architecture enhancement is also much desirable. Therefore, this thesis continues to propose a new perspective on the efficiency analysis of GPUs by considering the role of inter-warp divergence on GPU execution. As a metric to quantify inter-warp divergence, this thesis proposes Warp Progression Similarity (WPS) to measure temporal execution similarity of concurrent warps. Based on the WPS metric, this thesis introduces WPSaS, a WPS-aware Scheduler, giving an outlook on performance improvement.
opportunities. WPSaS achieves 10.6% improvement in throughput for three selected benchmarks. We anticipate similar performance improvements with benchmarks which either suffer from inter-warp conflicts in I$, or with high temporal under-utilization. Nonetheless, many design opportunities but also challenges are opened by divergence-aware design principles, such as finding the balance between inter-warp divergence and inter-warp conflicts in I$ and D$. 
Bibliography


BIBLIOGRAPHY


[18] V. Pham, P. Vo, V. T. Hung, and L. H. Bac, “GPU Implementation of Extended Gaussian Mixture Model for Background Subtraction,” in *Computing and Communication Technologies, Research,
BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


