FACULTY OF SCIENCES

QUANTIFYING AND MITIGATING THE COLD-START PROBLEM IN GPU SIMULATION

Jonas Sys
Student number: 01911330

Supervisor: Prof. dr. ir. Lieven Eeckhout
Counsellors: Dr. Mahmood Naderan-Tahan, Seyyed Hossein SeyyedAghaei Rezaei
Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science

Academic year: 2023 – 2024

GHENT UNIVERSITY

SUMMARY

Many modern applications and workloads depend on GPUs, ranging from gaming and graphics to machine learning and graph analysis. To support this ever-growing need for high-performance parallel processing, constant innovation in GPUs and their micro-architectures is a must. These innovations require methods to verify that changes made are beneficial for real workloads.

One way to verify these changes is through simulation, using e.g. AccelSim. These simulations are a cheaper and more practical way to test new architectures, as compared to in-silicon verification. The computational overhead of the simulation implies that only a subset of all kernels can be simulated within reasonable time. To remedy this, techniques like PKS and Sieve are used.

These techniques focus on selecting a subset of kernels in a workload, simulating only these, and then generalizing the results to the entire workload. This process takes a few steps, starting with profiling the workload. After profiling, statistical techniques are used in combination with clustering algorithms to determine which kernels to simulate.

This selection is then run through a simulator, giving us a certain set of performance metrics. These metrics are then generalized to the entire workload, giving us an estimate of the performance of the workload on the new architecture.

However, these techniques start from a flawed assumption. When simulating an entire workload, preceding kernels might have impacted the state of the caches on the simulated GPU. This cache state might lead to a different performance, depending on the degree of data reuse between kernels. This problem is commonly named the *cold-start problem*, referring to the cold state of the caches at the start of the execution.

In this thesis, we aim to show that the cold-start problem exists in both hardware and simulation. After that, we've come up with a few possible mitigations, raising accuracy to a level where the cold-start problem is no longer significant. We focus on both accuracy and feasibility, as the techniques should be applicable to real-world workloads.

SAMENVATTING

Deze dagen zijn er veel applicaties en *workloads* die grafische kaarten (*GPU's*) gebruiken, van games en andere grafische toepassingen tot machine learning en graaf-analyse. Om deze groeiende vraag naar performante parallelle programma's mogelijk te blijven maken, is er een constante nood aan innovatie in GPU's en hun micro-architectuur. Deze innovaties moeten echter altijd getest worden om er zeker van te zijn dat de veranderingen ook daadwerkelijk een verbetering zijn.

Een veel gebruikte manier om deze veranderingen te controleren, is door het gebruik van simulaties, bijvoorbeeld door gebruik te maken van AccelSim. Deze simulaties zijn een goedkopere en meer praktische manier om nieuwe architecturen te testen, vergeleken met het bouwen van een nieuwe chip. Het nadeel van deze simulaties is dat ze veel rekenkracht vereisen, waardoor het niet altijd mogelijk is om alle kernels in een *workload* te simuleren. Het simuleren van een volledige *workload* zou vaak te lang duren, waardoor er technieken zoals PKS en Sieve gebruikt worden.

Deze technieken bepalen een deelverzameling van de *kernels*, die dan gesimuleerd worden. De resultaten van de simulatie worden dan veralgemeend naar de volledige *workload*. Dit proces bestaat uit een aantal stappen, beginnend met het profileren van de *workload*. Nadat een *workload* geprofileerd is, worden statistische technieken gebruikt in combinatie met *clustering* algoritmes om te bepalen welke kernels gesimuleerd worden.

Deze geselecteerde kernels worden dan gesimuleerd, waarna we een aantal prestatie-metingen krijgen. Deze metingen worden dan veralgemeend naar de volledige *workload*, waardoor we een schatting krijgen van de prestaties van de *workload* op de nieuwe architectuur.

Deze technieken vertrekken echter van een gedeeltelijk foutieve aanname. Wanneer een volledige *workload* gesimuleerd wordt, kunnen de kernels die voorafgaan aan de gesimuleerde kernel de staat van de caches op de gesimuleerde GPU beïnvloed hebben. De staat van deze caches kan de prestaties van de gesimuleerde kernel beïnvloeden, afhankelijk van de mate waarin de data hergebruikt wordt tussen de kernels. Dit probleem wordt vaak het *cold-start* probleem genoemd, verwijzend naar de "koude" staat van de caches aan het begin van de uitvoering.

In deze thesis proberen we aan te tonen dat het *cold-start* probleem zowel in hardware als in simulaties bestaat. Daarnaast hebben we een aantal mogelijke oplossingen bedacht, die de nauwkeurigheid van de simulaties verhogen tot een niveau waarop het *cold-start* probleem niet langer significant is. We focussen op zowel nauwkeurigheid als haalbaarheid, aangezien de technieken ook toepasbaar moeten zijn op echte *workloads*.

Quantifying and Mitigating the Cold-Start Problem in GPU Simulation

Jonas Sys[1], Mahmood Naderan-Tahan[2], Seyyed Hossein SeyyedAghaei Rezaei[2], Lieven Eeckhout[2]

Abstract

[1]Ghent University
[2]Department of Electronics and Information Systems, Faculty of Engineering and Architecture, Ghent University

May 17, 2024

TOELATING TOT HERGEBRUIK

CONTENTS

1 INTRODUCTION

1.1 GPUs

Graphics Processing Units (or GPUs for short), are specialized electronic circuits (or chips), originally designed to accelerate computer graphics. In the modern day, these chips are most known in their standalone form: as a separate card that can be plugged into a computer's motherboard. However, they are also often integrated into CPUs, as is the case with Intel's integrated GPUs, or AMD's APUs.

The history of GPUs goes back to 1968, when the *Evans and Sutherland Computer Corporation* was founded [1]. This company built special-purpose 3D graphics hardware, which was mostly used in flight simulators. The hardware they made took several racks, and was extremely expensive.

The advent of 3D video gaming consoles, around 1994, brought about the era of mass-market 3D gaming. This led to the founding of a lot of companies, all focused on bringing 2.5D and 3D graphics hardware to PCs. One of these companies was NVIDIA, which was founded in 1993.

They released the GeForce 256 in 1999, which is commonly regarded as the first "real" GPU. Up to this point, the vertex processing (transforming 3D data into 2D coordinates for the screen), was done by the CPU, which was not built for this task. The actual rendering (computing lighting and shading) was already done by the GPU. The GeForce 256 allowed offloading the vertex processing to the GPU as well, allowing higher geometric complexity. At this point, computations were still done in fixed-function pipelines, which were hardwired to perform specific tasks.

This would change with the introduction of the GeForce 3 (also known as the NV20) GPU in 2001. It introduced programmable vertex shaders, where the vertex processing stage could be programmed by the user. In the following year, GeForce FX introduced programmable fragment shaders, allowing the user to take full control of the rendering pipeline.

At this point, GPUs were already very powerful, allowing for a lot of parallel computations to be done. The GeForce 6, for example, had a peak performance of 108 GFLOPS (billion floating-point operations per second). This dwarfed the performance of CPUs, which were only capable of a few GFLOPS at the time. The flexibility of the pipeline allowed for a lot of different applications to be run on the GPU, making it a very attractive platform for scientific computing. Despite this, programming for GPUs was still extremely challenging, as both in- and output were very restrictive. Problems had to be translated to a set of vertices, which were then rendered, and the results had to be read back from the framebuffer.

2006 saw the introduction of the *streaming multiprocessor* (SM), a piece of hardware that was used to run both vertex and fragment shaders. Additionally, these SMs could also run *compute shaders*, which allowed users to run code on the GPU, independent of the graphics pipeline.

This grew into the CUDA (Compute Unified Device Architecture) platform [2, 3], which was introduced in 2007. CUDA allows users to write code in C, which is then compiled to run on the GPU. The full CUDA software stack consists of three main elements:

- The CUDA hardware driver, which is responsible for managing the GPU and scheduling work.
- The CUDA API and its associated runtime. The API is an extension to the C programming language.
- Mathematical libraries, optimized to be run on GPUs.

We will focus mostly on the compute capabilities of GPUs (specifically newer generations of NVIDIA GPUs), as they are the most relevant to our research.

Every year, more and more AI and machine learning papers are published [4]. Many of these machine learning algorithms rely on GPU computations to train their models, creating a growing need for more and faster GPUs. Innovations to GPU hardware often require changing the micro-architecture. These changes require verification to ensure that they are, in fact, improvements. However, the creation of a new, physical GPU is very costly and time-consuming. To avoid this overhead and allow for a faster iteration cycle, simulation is used.

1.2 GPU Simulation

The main idea behind simulation is to model the behavior of a system, and then run this model on a computer. This allows for the system to be tested in a controlled environment, without the need for the actual hardware. This is especially useful for GPUs, as the hardware is very expensive, and the development cycle is very long.

A good (architectural) simulator should be flexible, to allow the end-user to make drastic changes to the hardware architecture, without excessive effort. That being said, an accurate model is very hard to build. Two factors hinder us in this regard:

- The desire for increased abstraction; many models are kept at a high level of abstraction, allowing for easier understanding. However, the hidden details can make the model less accurate.
- Most hardware implementations are proprietary, i.e. the actual details are not publicly known.

This means that many simulators used in academia suffer a hit on their accuracy.

Most GPU simulators are cycle-level, meaning that they simulate the hardware at the clock cycle level. Some contemporary examples are Barra [5], GPUOcelot [6], Multi2Sim [7], gem5 [8], GPGPU-Sim [9], and its derivative AccelSim [10].

1.2.1 GPU Code

The typical GPU uses a two-step compilation process:

1. High-level code is compiled to an intermediate representation, which is defined by the vendor. Often, this intermediate representation is referred to as the *virtual Instruction Set Architecture* (vISA). For NVIDIA GPUs, this vISA is called PTX (Parallel Thread Execution). The PTX format is thoroughly documented, and remains relatively stable across GPU generations, which makes it a good target for simulators.
2. The vISA code is compiled to the actual hardware instructions, referred to as the *machine Instruction Set Architecture* (mISA). This just-in-time compilation step is done at runtime. For NVIDIA GPUs, this mISA is called SASS (Source and Assembly). Compared to the PTX format, SASS is very poorly documented, and undergoes changes with every new GPU generation.

As Jain et al. [11] point out, the choice of which ISA to simulate can have an impact on the accuracy of the simulation results. They compared the accuracy of simulating at the PTX level to simulating at the SASS level when using the GPGPU-Sim simulator. From the workloads they analyzed, most tend to have a better accuracy on the mISA level.

When simulating SASS code, GPGPU-Sim translates the instructions to a new format, called PTXPlus. This PTXPlus format is a 1:1 representation of the GT200 SASS code, but translated to resemble PTX in syntax.

1.3 Improving simulation speed

However, simulation is not without its drawbacks. The main issue is that simulation comes with a very large overhead compared to running algorithms on the actual hardware. This overhead is

so large that applications that would take only about an hour on real hardware could easily take upwards of a century [12]. Additionally, the accuracy of the simulation is often lower than that of the real hardware.

Some research has gone into improving the speed of GPU simulation. As most GPU workloads consist of a large number of kernels, sampling a subset of these kernels can be used to speed up the simulation. Techniques like Principal Kernel Analysis [12] and Sieve [13] have been developed to identify which kernels to simulate, and how to generalize the results.

This gives us the general outline of the simulation process:

1. Profile the workload on real hardware, to both gather performance numbers and create instruction-level traces. For this, we will use NVIDIA's Nsight Compute [14] and NVBit [15].

2. Use these performance numbers, together with a sampling method, to identify which kernels to simulate.

3. Simulate the traces of these selected kernels on the simulator, gathering performance numbers. Our simulator of choice is Accel-Sim [10].

4. Use these performance numbers to predict the performance of the entire workload.

However, this methodology of simulation can cause another drop in the accuracy of the simulation. The performance of a kernel can be influenced by the kernels that run before and after it. In this case, we will focus on the cache state of the GPU.

1.4 The cold-start problem

The cold-start problem is a problem that arises due to sampling, both in CPU and GPU.[1] Each kernel modifies the state of the caches by executing memory instructions (e.g. loads and stores). When we sample the workload to select only a subset of the kernels, there is a large chance that we will choose only intermittent kernels. This means that the state of the caches between kernel invocations will not reflect the actual hardware state.

Suppose a workload consists of four consecutive kernels; A, B, C, and D. If kernel A were to be an initializing kernel, readying data for the other kernels, then it might also warm up the caches, filling them with data. This means that kernel B, which might reuse a lot of that data, might not suffer the latency of its memory instructions, as it would only have to fetch the data from the cache and not DRAM. However, if we were to only sample kernel B, each cache hit might turn into a miss, drastically reducing the IPC of the kernel.

This problem is known as the cold-start problem, and it can have a significant impact on the accuracy of the simulation. It also gives us our research question: *how big is the impact of the cold-start problem (both in hardware and simulation), and can we mitigate it?*

1.4.1 Quantifying

In Chapter 4, we will quantify the cold-start problem in hardware. We will do this by running a set of benchmarks on real hardware, and then simulating a subset of these (the most promising ones). The benchmarks range from machine learning inference to graph traversal and molecular simulation. Each of these benchmarks is run twice; once normally, and once with a forced cold start. We force the cold starts by flushing the caches between every pair of kernels.

By doing this, we show that the cold-start problem exists in GPU hardware, and that it can have a significant impact on the performance of the workload. We also look at the impact of the cold-start problem when taking into account each kernel's weight (i.e. what portion of the workload it represents). Additionally, found that the cold-start problem is tied to the degree of inter-kernel data reuse, which we formalize in Section 4.3. From this data, we select the following workloads:

[1] As we focus on GPU simulation, we will refer to units of execution as kernels. In CPU simulation, these might be functions or basic blocks.

- **DCT**, from the CUDA samples;
- **OceanFFT**, from the CUDA SDK; and
- **3D-UNet inference**, a machine learning benchmark from MLPerf [16].

These are selected based both on how much they were impacted by the cold-start problem, and by their feasible runtime in the simulator (for 3D-UNet we limited the execution to the first 130 kernels).

After profiling using the NVBit tool that comes with AccelSim, we will simulate these workloads, analyzing the results in Chapter 5. Each workload will be run twice again, so that we can compare the performance impact of the cold-start problem on these workloads. The main conclusion will be that the cold-start problem does exist in the simulator as well, albeit with a very different impact. Overall, the simulator will suffer less from the cold-start problem than the hardware, but the impact will still be noticeable.

1.4.2 Mitigating

After quantifying the cold-start problem, we will look at ways to mitigate it. Chapter 6 will focus on three avenues we've looked into:

Cache warming: in order to force the cache state to be accurate, we can simulate each preceding kernel. However, this is not a feasible solution, as it will increase the simulation time by a lot. This is what we tried to avoid by using sampling.

Selected simulation: only the kernels directly before a selected kernel will have a significant impact on the cache state. Additionally, only the memory instructions do actually modify the cache state. From these two observations, we will simulate only the memory instructions of a low number of preceding kernels. The main conclusion here will be that simulating the memory instructions of a single preceding kernel will give us a very high accuracy (often up to 99%) at a rather low computational cost.

Correction factors: using certain factors we can compute a factor to correct the performance numbers of a cold-started kernel. The main idea will be to find a subtractive factor for the number of cycles. This approach will prove to also yield good results, but will be slightly less accurate than simulating memory instructions for a single additional kernel.

1.5 Conclusions

In this thesis, we will quantify the cold-start problem in GPU hardware and simulation. We will show that the cold-start problem exists in both hardware and simulation, and that it can have a significant impact on the performance of a workload. We will also show that this problem is tied to the degree of inter-kernel data reuse. Additionally, we will show that the cold-start problem is less severe in simulation than in hardware. Finally, we will look into ways to mitigate the cold-start problem, exploring three different avenues. The first avenue, simulating all preceding kernels, will be discarded as it is not feasible, especially for larger workloads. The second avenue, simulating only the memory instructions of a single preceding kernel, will be shown to give a very high accuracy at a low computational cost. The third avenue, using correction factors, will be shown to be viable as well, having a slightly lower accuracy to the second avenue, but at a lower computational cost.

2 LITERATURE REVIEW

The topic of GPU simulation has been studied extensively, albeit not as extensively as CPU simulation. Before continuing with the cold-start problem, we will shortly discuss the current state of GPU simulation. The three main papers this thesis is based on are *AccelSim* [10], *Principal Kernel Analysis* [12], and *Sieve* [13]. The first of these provides the simulator we use, while the latter two provide the kernel sampling techniques that are commonly used to speed up simulation.

2.1 GPU Simulation

As outlined in the AccelSim paper [10], most of the ISA and architecture changes in GPU innovation are usually closed off by the industry. This makes it harder for research to keep up with industry changes.

The proposed simulator AccelSim, which is based on the older GPGPU-Sim simulator [9]. It consists of four main components:

1. The flexible frontend;
2. A very flexible and detailed performance model;
3. A correlation generation tool, which can be used to expand the simulator to newer GPU architectures; and
4. A configuration tuner based on micro-benchmarks, which uses the correlation generation tool to tune the simulator.

2.1.1 AccelSim Frontend

An important improvement which AccelSim brings, is the very flexible frontend. The existing GPGPU-Sim is largely limited to virtual ISA (vISA) execution-driven simulation, using PTX (parallel thread execution) instructions. AccelSim improves upon this by adding support for trace-driven machine ISA (mISA) simulation, which uses the actual SASS (source and assembly) code.

In trace-driven mode, the simulator reads the mISA trace and converts it into the internal ISA-independent representation. This representation has a one-to-one correspondence with SASS instructions, where the active mask and memory addresses are embedded in the trace itself. Conversely, the execution-driven mode requires the computation of the active mask and memory addresses at runtime, which is done by emulation of the PTX code. Finally, another benefit of the mISA is that it includes the actual register allocation, while the vISA assumes an infinite amount of registers.

2.1.2 AccelSim Performance Model

AccelSim's performance model attempts to mimic actual GPU hardware as closely as possible. To this end, it is structured in a number of streaming multiprocessors (SMs), each of which is composed of a number of warp schedulers. Each of these schedulers has an associated register file and is in turn composed of a number of execution units.

The combination of a warp scheduler with its register file and execution units is called a sub-core. These only share an instruction cache and memory subsystem.

As with the frontend, the performance model is also very flexible. It can simulate either a unified or split L1 Data cache (L1D), as the device driver can configure the cache at runtime. In many

modern workloads, an adaptive cache is used. This means that if a kernel does not use shared memory, all on-chip storage is used for the L1D cache. Additionally, this flexible cache model supports multiple cache designs: throughput-oriented, banked, and sectored; which allows for high-accuracy simulation.

Importantly, the simulator also needs to model the CPU-to-GPU memory copy engine. Each DRAM access has to pass through the L2 cache, changing cache state.

Finally, the performance model also includes support for domain-specific process pipelines. Extending the simulator with these pipelines (e.g. Tensor Cores) requires the addition of some config files. However, if the user wishes to also support the PTX simulation of these pipelines, they will need to add emulation code in the GPGPU-Sim implementation.

2.1.3 Extension and Verification

The final two components, the correlation generation tool and the configuration tuner, are not only used to extend the simulator to newer architectures, but also to verify the simulator's accuracy.

Firstly, the tuner uses micro-benchmarks to tune the simulator; each of these micro-benchmarks can be used to discover non-public configuration parameters, including:

- It can pinpoint changes in memory latency and bandwidth (both for L1 cache, L2 cache and shared memory);
- It can detect the cache write policy and its configuration (associativity, line size, etc.).

After running these micro-benchmarks, the tuner reads the results and provides a configuration file which can be fed to the performance model. Some parameters cannot be determined by the benchmarks themselves, like warp scheduling policy, memory scheduling, and some L2 cache parameters (interleaving and hash function). To determine these, the tuner will attempt to simulate each possible combination on a set of memory bandwidth micro-benchmarks. The configuration with the highest average hardware correlation is then picked to be the correct one.

The other component, the correlation generation tool, can be used to generate targeted information on inaccuracies. This information is important because the tuner might not be able to detect and/or capture drastic architectural changes. These changes often require manual intervention.

2.1.4 AccelSim Conclusion

AccelSim is a very flexible and detailed simulator, which can be used to simulate both vISA and mISA code. It is also able to simulate a wide range of cache designs, and can be extended to support domain-specific pipelines. The correlation generation tool and configuration tuner are used to verify the simulator's accuracy, and to extend it to newer architectures.

The current version can simulate up to a speed of 12 500 warp instructions per second, which still improves upon the previous GPGPU-Sim version. Half of this speedup comes from the trace-driven mode, which avoids overhead of functional execution. The user can also set kernel-based checkpoints to avoid simulation of non-interesting regions.

The other half of the speedup comes from a simulation optimization strategy called *simulation-gating*, which provides a tradeoff between event-driven and cycle-driven simulation. During simulation, it is quite often the case to have thousands of in-flight memory requests from hundreds of active threads. This means that each cycle, there is always something to simulate. However, ticking every component every cycle can be expensive, especially if there are quite a few empty components (e.g. cores, execution units, caches, and DRAM channels). To avoid this, the simulator only ticks the active components.

2.2 Kernel Sampling

Simulation has a lot of benefits; like the inherently configurable design, the flexibility, and the ability to reconfigure hardware to analyze model changes, but it also has the downside of its inherent enormous overhead. The authors of the PKA paper [12] show that workloads which would take mere seconds on real hardware, could take upwards of a century on a simulator. This drawback means that each simulation platform must limit the number of instructions.

Many approaches attempt to restrict the workload and/or platform to speed up simulation. To this end, workloads could be scaled down; which would limit the applicability to extremely short runtimes. Another option would be to simulate only the first few billion instructions of a larger, scaled workload. However, this option limits the horizon of the simulation, often only simulating the warmup-phase of the program. A third option would be to scale the GPU itself down, but this would force the workload to adapt to the scaled-down hardware. Finally, neither of these options have really been validated against scaled workloads.

To come up with a better solution, the authors turned their attention to the solutions employed when simulating CPUs. For CPU workloads, the simulation often focuses on selecting a subset of the basic blocks in each thread. However, this does not translate well to GPU simulation, as the control-flow graphs (CFGs) each GPU thread executes are usually small and trivial compared to CPU CFGs. To really make a difference in simulation time, you would need to curtail the number of threads, rather than the number of basic blocks per thread.

2.2.1 Principal Kernel Analysis

To select a subset of the kernels to simulate (and thus limit the number of threads), the authors propose a new technique called Principal Kernel Analysis (PKA). This technique is based on the following three observations:

- Even though a workload can contain many kernel instances, all of them can be characterized and grouped based on a small number of architecture-independent metrics.
- Heavy-duty detailed profiling of an entire workload can very easily take an extreme amount of time. To combat this, we can do the full profiling on a subset of kernels, and use lightweight profiling on the rest. Finally, statistical techniques can be used to generalize the results from the subset to the entire workload, allowing us to cluster all kernels without spending too much time on profiling.
- During the lifetime of a kernel, its IPC tends to stabilize to a value representative for the entire kernel. This allows us to cut the execution short, and still get a good estimate of the kernel's final performance values.

The PKA technique has two big steps: firstly *Principal Kernel Selection* (PKS), and secondly *Principal Kernel Projection* (PKP). During this first step, a workload is profiled, after which all profiling results (both heavy-duty and lightweight) are used to cluster all kernel invocations. From each cluster, a representative kernel is selected to be simulated. The second step, PKP, then uses the third observation above to stop the simulation once the deviation in IPC is below a certain threshold. This clear approach gives PKA its four main characteristics:

- **Scalable:** the two-level profiling assures that a reasonable amount of time is spent on pre-processing, after which the selection algorithm can choose which kernels to simulate. By simulating a limited number of kernels, the simulation time is drastically reduced.
- **Automatic:** PKA requires only very few inputs: profiling results (which can be directly obtained from the workload itself); a maximum error bound for the PKS step; and finally, a confidence interval for the PKP step.
- **Tunable:** the parameters outlined in the previous point allow the user to tune the tradeoff between simulation time and accuracy.

- **Verification:** PKA has been verified against silicon, which is not true for some other sampling techniques.

2.2.2 Principal Kernel Selection

During the profiling phase, all metrics gathered are micro-architecture-independent. This means that they depend only on the workload, not the GPU being profiled. By making sure that the metrics are independent, we can avoid discrepancies (similar to the differences between x86 instructions and micro-ops when simlating CPUs). The metrics used are:

- Coalesced global loads and stores
- Coalesced local loads
- Thread global loads and stores
- Thread local loads
- Thread shared loads and stores
- Thread global atomics
- Instruction count
- Divergence efficiency
- Thread block count

After gathering results from the heavy-duty profiler (if needed, for only a limited number of kernels), principal component analysis (PCA) is used to reduce the dimensionality of the data. This makes sure that we can avoid the curse of dimensionality, as the principal dimensions will have the highest variability. Once we have the smaller dataset, we use k-means to cluster the data. The k-means algorithm was chosen partially because of explain-ability, and partially because it can be tuned with its factor k.

From each of the obtained clusters, a representative kernel invocation is selected. The authors have tried three different methods of selection: random, first chronologically, and closest to cluster center. The first option, random, caused inconsistent error rates, and is not recommended. The other two options, however, showed negligible differences in error rates. In this case, the first chronologically was picked, as this has certain benefits in practice (for both tracing and profiling).

If the workload is very large, it can be impractical to use heavy-duty profiling on all kernels. In this case, we can use a two-level approach: profiling the first j kernels using the heavy-duty profiler, while from the other $n - j$ kernels only a subset of the metrics is gathered. We cluster the fully profiled kernels using the PCA and k-means approach, while the others are mapped to the clusters using either *Stochastic Gradient Descent*, *Naive Bayes Gaussian*, or *Multi-layer Perceptron*.

2.2.3 Principal Kernel Projection

While PKA solves the problem of the number of kernels, it does not address long-running ones. To this end, the authors propose the Principal Kernel Projection (PKP). PKP is based on the observation that, since each thread in the grid runs the same code, the code of a kernel usually has only a few phases (largely due to their lifetime being shorter than a CPU thread). This means that the IPC of a kernel usually stabilizes, even in very irregular applications (like graph processing).

PKP attempts to detect this stabilization by tracking two statistics about the IPC across the last n cycles: the rolling average, and the deviation. One of the parameters to the PKA application is the confidence interval for stabilization detection. From this parameter, a threshold value s is computed. When the deviation drops below this threshold, the IPC is considered quasi-stable, and the simulation can be stopped. A smaller value for s leads to higher accuracy at the cost of a longer simulation time. According to the authors, a value of $s = 0.25$ should be fine.

2.3 Improving PKA

While PKA is a very powerful technique, leading to drastically lower simulation times while still maintaining a high accuracy for most application, it is not perfect yet. The main issues with PKA, as outlined in the Sieve paper [13], are:

- Since the PKS phase assumes the same execution time for all invocations in a cluster, there is a very large variability in cycle count.

- The heavy-duty profiling phase is very time-consuming. Using only the instruction count (which can be obtained from the lightweight profiler) also leads to a high accuracy.
- PKS also relies on a golden reference obtained from real hardware to select its representative kernels. This implies that the final clustering is not micro-architecture-independent, since the hardware platform ultimately decides the clustering. Sieve only uses the instruction count to cluster the kernels, so the actual clusters and representatives will be micro-architecture independent.

The Sieve technique improves upon both steps of the PKS phase. It attempts to speed up the profiling step by only requiring a few easily-gathered metrics from the application to cluster the kernels. These metrics (kernel name, invocation ID, and number of dynamically executed instructions) can be obtained from the lightweight profiler, which reduces profiling time. This is a much lower one-time cost than PKA requires, opening up many more applications that were prohibitively expensive to profile using PKA .

2.3.1 Strata

To reduce the variability in cycle count, Sieve uses strata-based clustering. This allows for a better representative kernel to be selected. Within each stratum, only instances (or invocations) of the same kernel are present, while also keeping the amount of dynamically executed instructions as similar as possible. The strata are organized in three tiers:

1. **Tier 1:** all instances in tier 1 have the exact same number of dynamic instructions; removing all variability.
2. **Tier 2:** only a small amount of variability is allowed in tier 2, with a configurable maximum.
3. **Tier 3:** the remaining instances are placed in tier 3, where the variability is allowed to be much larger.

In order to determine to which tier a kernel invocation belongs, the Coefficient of Variation $CoV = \frac{\sigma}{\mu}$ (the ratio between standard deviation and mean instruction count) is used. A threshold θ can be set by the user; where a lower θ implies less variability within the strata and higher accuracy at the cost of a longer simulation time. The authors found that $\theta = 0.4$ is a good compromise.

To improve the variability of tier 3, which allows for a lot of variability, Kernel Density Estimation (KDE) is used. This allows to minimize the number of strata while also limiting the variability in instruction count (using the same threshold θ).

Using this methodology, the authors found that Sieve was able to fit most kernel invocations from the MLPerf [16] and Cactus [17] workloads into tiers 1 and 2.

2.3.2 Selection

When selecting a representative kernel, Sieve makes a decision based on the tier of the stratum:

1. **Tier 1:** in tier 1, all kernel invocations are identical, so the first chronologically is picked.
2. **Tiers 2 and 3:** in these tiers, the first chronological kernel with the most dominant cooperative thread array (CTA) is selected. Using the CTA as a metric makes sure that the selected kernel invocation occupies the hardware resources in the most representative way.

Additionally, a weight for each stratum is computed as $w_i = \frac{\text{total instruction count of stratum } i}{\text{total instruction count of workload}}$. This ensures that each stratum is weighted according to its actual weight in the workload.

2.3.3 Performance Prediction

The final step in the Sieve technique is the performance prediction. After simulating the representative kernels and obtaining their performance numbers (e.g. IPC), these can be generalized per stratum. Computing or predicting the final IPC for the application is done by taking the weighted harmonic mean of the IPC values from each stratum:

$$IPC = \frac{1}{\sum_{\text{stratum/cluster } i} \frac{w_i}{IPC_i}}$$

2.4 Cold start in CPU simulation

This thesis will focus on the cold-start problem in GPU simulation. However, extensive research has already been done on the cold-start problem in CPU simulation. We will shortly discuss some results from this research, both regarding quantification and mitigation, as they might be useful in the context of GPU simulation. Finally, we will look at one mitigation strategy in detail.

Both in GPU and CPU simulation, workloads are often too large to simulate in their entirety. Whereas GPU simulations can take up to centuries, simulation of common CPU benchmarks takes usually takes a few days or weeks [18]. This is still too long of a time to be practical, so researchers have come up with sampling strategies to reduce the simulation time, just like with GPU simulation.

These sampled methodologies suffer from the cold start problem as well, as some hardware state might be lost (e.g. caches, TLBs, branch predictors, etc.). Additionally, for many samples, this warmup history is very long, as caches[1] are typically rather large. This leads to proportionally long warmup times.

From this, we can gather two main problems:

1. **Selection:** CPU benchmarks often have a lot of different phases, and all phases need to be represented among the samples. This is not a problem we will look into, as it is covered by the PKA and Sieve techniques.

2. **Cold-start:** the hardware state at the beginning of each sample might be lost. The main cause here is the fact that a lot of instructions are never simulated, so they have not had a chance to modify the micro-architectural state.

Regarding the cold-start problem, other works often identify three phases in the simulation:

Cold simulation: this is pure, functional simulation. Instructions are fast-forwarded and do not modify the micro-architectural state. This type of simulation is very fast, but does not contribute to the final results obtained by simulation.

Warm simulation: a region of instructions before the sample is simulated. These instructions are used to warm up the caches, TLBs, and branch predictors. Because more data needs to be gathered, the simulation time is longer (about twice as long as cold simulation).

Hot simulation: the sample is simulated, gathering all performance metrics. This type of simulation is very slow (almost ten times slower than warm simulation).

In [19], the simulation speeds of each of these phases are compared: cold simulation runs at about 7 MIPS (million instructions per second), warm simulation at about 3 MIPS, and hot simulation at about 0.3 MIPS. To speed up simulation, we need to reduce the amount of instructions run under warm and/or hot simulation. The number of instructions run under hot simulation can be reduced by sampling, which requires simulating instructions under warm simulation.

[1] We will often refer to just caches, as these are most relevant to the research in this thesis, but this also includes TLBs, branch predictors, and more.

2.4.1 Warmup strategies

The authors of [18] have identified a number of well-known strategies to reduce the warmup time:

- **Cold scheme:** this scheme performs no warmup at all, assuming all caches are cold at the beginning of every sample.
- **Checkpoint:** [20] this scheme saves the micro-architectural during profiling, and restores it at the beginning of each sample. However, this requires a lot of disk space, as a lot of micro-architectural variables have to be saved and restored. On the other hand, it does guarantee a perfectly warmed up cache.
- **Stitch:** [21] this scheme attempts to approximate the micro-architectural state at the beginning of a sample by taking into account the state at the end of the last sample, stitching them together.
- **Prime-xx%:** [21] this scheme assumes that the first $xx\%$ instructions of the sample are warmup, and only gathers performance data on the other instructions.
- The authors of [22] propose a combination of the Stitch and Prime-xx% schemes. In this combined strategy, the hardware state at the beginning of the sample is assumed to be the hardware state at the end of the previous sample, plus the changes made by the first $xx\%$ instructions of the sample.
- **Correction factor:** [21, 23] using statistics and math, the number of cold misses due to the cold-start problem is estimated.
- **NSL:** [24] no-state-loss determines which memory addresses are used by the sample. Using this information, it checks memory instruction preceding the sample, keeping track of whether they would overwrite the cache line for any memory address in the sample. This LRU (least-recently-used) cache is then used to warm up the cache. Obviously, this scheme does not work for branch predictors, and only works for caches (and TLBs) using the LRU eviction policy.
- **MSE:** [25] minimal subset estimation uses probability and a set of formulae to estimate the start position of the warming region.

Two other methods, *Memory Reference Reuse Latency* (*MRRL*) [26] and *Boundary Length Reuse Latency* (*BLRL*) [18], will be discussed more extensively below. Both are derivatives of the *MSE* scheme.

From all these methods, only *NSL*, *MSE*, *MRRL*, and *BRLR* guarantee a nearly perfect hardware state at a reasonable cost.

2.4.2 Memory reference reuse latency

The name of the method is a reference to the metric used: the *memory reference reuse latency* [26] refers to the number of instructions between two consecutive references to the same memory location. This metric is used to estimate the required length of the warmup phase, given a factor representing the tradeoff between accuracy and simulation speed.

In general, MRRL works like this, using all instructions between two consecutive samples as the pre-sample regions for the next sample:

1. The entire region, both pre-sample and sample, are divided into N_B buckets, each having L_B contiguous instructions (this means a total of $N_B * L_B$ instructions are considered). These buckets are indexed from 0 to $N_B - 1$, with bucket 0 being at the start of the pre-sample region. Additionally, these buckets are divided into $N_{B,P}$ pre-sample buckets and $N_{B,S}$ sample buckets ($N_B = N_{B,P} + N_{B,S}$). Finally, for each bucket i, a counter c_i is initialized to 0.

2. For each memory reference in the region, its MRRL is calculated, and the relevant counter is incremented. To determine which counter c_i should be updated, the MRRL is (integer) divided by L_B ($i = \lfloor \frac{MRRL}{L_B} \rfloor$).

3. After all memory references have been processed, a histogram is created from the counters. For each counter c_i, a value $p_i = \frac{c_i}{\sum_{j=0}^{N_B-1} c_j}$ is computed. These values correspond to the probability that a memory reference has an MRRL in the range $[i * L_B, (i+1) * L_B)$. Due to the temporal locality of memory references, we expect p_i is higher for smaller i.

4. Using this histogram, we can compute the percentile $K\%$, and its corresponding bucket k. This means that $K\%$ of all memory references have an MRRL of at most $k * L_B$.

5. These k buckets are then selected as the warmup buckets, meaning that warm simulation is started $k * L_B$ instructions before the sample.

While this yields a very good estimate of the warmup region, it does forget about one issue: when the MRRL behavior of the pre-sample and the sample do not match very well, it might be suboptimal. Due to this mismatch in behavior, it could overestimate the warmup region, leading to a longer simulation time. On the other hand, it could also underestimate the warmup region, leading to lower accuracy than expected. To mitigate these issues, the BLRL scheme was proposed.

2.4.3 Boundary length reuse latency

BLRL [18] is a derivative of the MRRL scheme, keeping the same general workflow. Instead of using each memory reference, like MRRL does, it only uses memory references whose MRRL crosses the boundary between the pre-sample and the sample. This means that only the memory references that need warming up are considered, which leads to a smaller warmup region.

Additionally, the BLRL scheme computes only takes into account the pre-sample reuse latency; they subtract the reuse latency that is part of the sample. For example, if instruction i (part of the sample), reuses a memory location that was last used by the pre-sample region, we compute the pre-sample reuse latency as $MRRL - (i - N_{B,P} * L_B)$ (we subtract the number of instructions since the beginning of the sample from the MRRL). Using these pre-sample reuse latencies, a similar histogram is created, and normalized to the number of reuse latencies crossing the boundary.

This histogram is then, like MRRL, used to compute the percentile $K\%$. In the same way, the corresponding bucket count k is selected as the warmup region.

The authors note three key differences between MRRL and BLRL:

1. Instead of considering all memory references, BLRL only considers those which also appear in the sample.

2. Instead of using all memory references in both pre-sample and sample, BLRL only considers those crossing the boundary.

3. Instead of using the actual MRRL value, BLRL uses the pre-sample reuse latency (it only counts latencies up to the border).

From an analysis of multiple CPU benchmarks, the authors found that BLRL is more accurate in almost all cases, and very often features a shorter warmup phase. This means we get a better accuracy at a lower cost, for almost all benchmarks.

3 THE COLD-START PROBLEM

3.1 Caches

The cold-start problem is inherently tied to caches. Caches are a hardware component that is used to improve the performance of a computer system. They attempt to mitigate the latency of accessing data from main memory (DRAM) by storing a subset of the data that is recently accessed. Older research has shown that many applications exhibit temporal locality, meaning that if a piece of data is accessed once, it is likely to be accessed again in the near future. Caches exploit this property by storing recently accessed data in a small, fast memory that is closer to the processor (in this case, the GPU) than the main memory.

When the GPU issues a memory request (often due to a load or store instruction), the cache controller checks if the data is present in the cache. If it is, and the data is not stale, the cache controller returns the data to the GPU, avoiding the expensive DRAM access. If the data is not present in the cache, the cache controller must fetch the data from the main memory. Modern GPUs have multiple levels of cache, each with their own size, latency, and other parameters. Often, the caches are organized in a hierarchy, where closer caches (which have a lower latency), are smaller. We will mainly focus on the last-level cache (LLC), which is the cache closest to the DRAM.

3.1.1 Cold Caches

Initially, the caches of a GPU are cold, meaning that they do not contain any data. This means that the first time a new piece of data is accessed, it will not be present in the cache, and the GPU must fetch it from DRAM. This causes a significant latency penalty, as the latency of accessing DRAM is much higher than the latency of accessing the LLC. From then on, repeat accesses to the same data will be much faster, as the data will be present in the cache.

The benefit of caches is that they can persist data between kernel invocations. This implies that a kernel can "warm up" the caches by accessing all data that the next kernel might need.

In Figure 3.1, we see an example of two kernels, and how caching can improve performance. This small example abstracts away of how the caches decide which cache line is used for which memory address. The first kernel executes some memory instructions, which the second one builds upon. The full "code" executed (we used a simplified representation of the memory instructions) is as follows:

```
1   ; Kernel 1:
2   ld 0x00AA ; performs a load
3   ld 0x11BB
4   st 0x00AA ; performs a store
5   st 0x22CC
6   ld 0x33DD
7
8   ; Kernel 2:
9   ld 0x33DD
10  ld 0x11BB
11  st 0x00AA
```

The figure shows for each instruction the cache state before and after the execution (if the cache state changed). We can clearly see that, for kernel 1, the caches start out cold, but get warmed up by the unique memory accesses (`0x00AA`, `0x11BB`, `0x22CC`, and `0x33DD`). Because the cache couldn't supply us with the data, we call this a cache miss. More specifically, since we access a memory

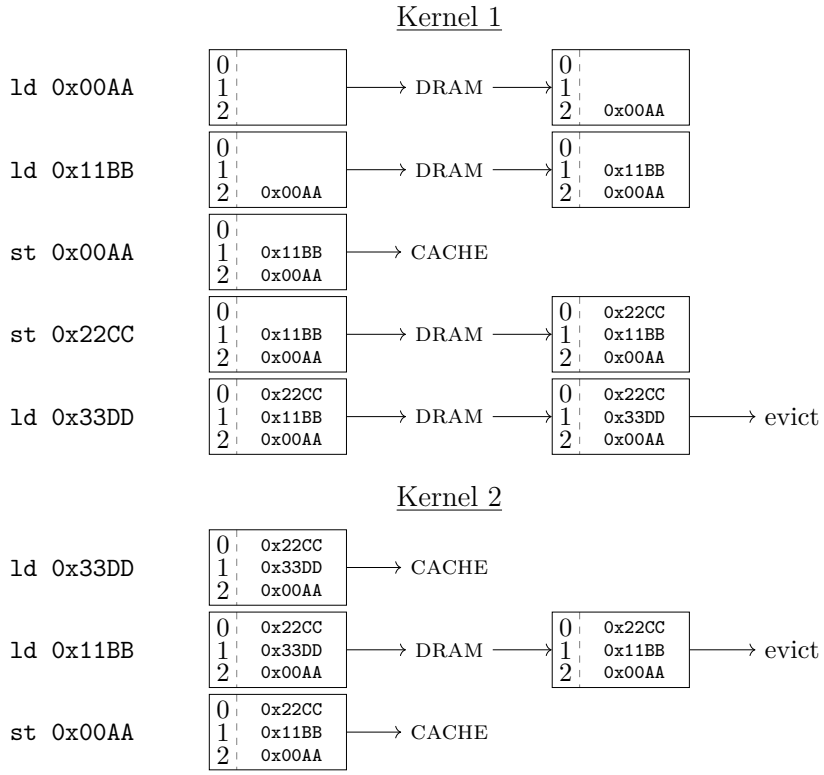


Figure 3.1: Caches example

location that had not been accessed before, it is called a cold miss. Each cache miss causes a DRAM access, costing us precious cycles. However, the repeat access of `0x00AA` on line 4 (`st 0x00AA`) goes only through the cache, speeding up the access.

Because we do not flush the caches between two kernel invocations in this example, kernel 2 can benefit from the cache state that kernel 1 left behind. At this point, we have no cold misses anymore, as all memory locations that are accessed have been accessed before. However, line 10 causes another miss, as the cache line associated with `0x33DD` was overwritten by the access of `0x11BB` on line 9, causing a conflict miss. These conflict misses occur when multiple memory locations are mapped to the same cache line, and one of them is evicted from the cache.

These evictions are a normal part of cache operation. The controller will choose a line to remove from the cache based on its eviction policy. Some common cache eviction policies are[1]:

LRU (Least Recently Used) The line that has not been accessed for the longest time is evicted. This choice is based on the assumption that, if a line has not been accessed for a long time, it is less likely to be accessed in the near future.

LFU (Least Frequently Used) The line that has been accessed the least amount of times is evicted.

FIFO (First In, First Out) The line that has been in the cache the longest is evicted.

RR (Random Replacement) A random line is evicted.

For simplicity, we will assume that all instructions are run serially, i.e. no two instructions are issued at the same time. Additionally, we will assume that a cache hit takes 180 cycles, and a DRAM access takes 250 cycles (which is close to what the simulator will use). From these numbers we can compute how many cycles each kernel will take to execute. For kernel 1, we have 4 misses and 1 hit, resulting in a total of 1180 cycles; while kernel 2 can be executed in 610 cycles.

[1] Cache eviction policies only have an effect when a memory location can be placed in one of multiple cache lines (the cache has an associativity of at least 2). If a memory location can be placed in only one cache line, it has to replace that specific cache line.

3.2 The Problem

As we discussed in Section 2.2, most kernels in a workload won't be simulated. This means that a lot of intermittent kernels will never be run, which might impact the cache states. In effect, we can assume that each kernel will be run with cold caches, as the kernels right before a kernel might not have been run.

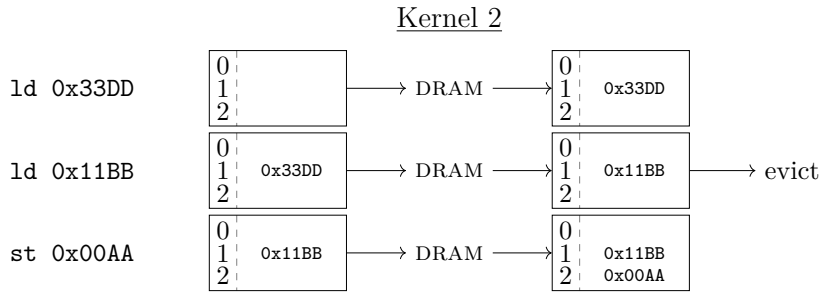


Figure 3.2: Sampled example

In Figure 3.2, we revisit the same example, assuming that the sampling method only selected kernel 2 to be simulated. Below is the code excerpt for kernel 2:

```
8     ; Kernel 2:
9     ld 0x33DD
10    ld 0x11BB
11    st 0x00AA
```

This means that kernel 1 was not run, and kernel 2 can not benefit from the cache state that kernel 1 left behind. This causes all memory accesses to be cold misses, as the caches are cold. Again, we can compute the number of cycles kernel 2 will take to execute. This time, we have 3 misses, resulting in a total of 750 cycles (taking into account our assumptions from before). This is a significant increase in execution time compared to the 610 cycles we had before (about 23%).

4 THE COLD-START PROBLEM IN HARDWARE

4.1 Initial profiling

Firstly, we need to quantify and detect whether the cold-start problem exists in GPGPU hardware. To this end, we have analyzed some workloads and benchmarks using the NVIDIA Nsight Compute tool. This tool allows us to control the caches during the execution of an application. Additionally, we can gather detailed metrics, like IPC, for each kernel in the workload.

Each workload was profiled twice: once like it would run normally, and once with the caches flushed between kernel invocations. Nsight Compute supports this option by using the `--cache-control=all` argument. To keep the runtime in check, we limited the execution to 20 000 kernels per workload. These experiments were run on an NVIDIA GeForce RTX 3080 [27]. This GPU has 68 SM cores, 6 MB of L2 cache.

After profiling, we attempted to match each profiled non-flushed kernel with its flushed counterpart. This proved easier said than done, as the kernel IDs were not consistent between the two profiles.

From this initial analysis, the most interesting workloads seemed to be PyTorch DCGAN [28] and Gunrock [29] (on road traversal), from the Cactus [17] suite. Below is a full list of all analyzed workloads:

- Gromacs [30] and LAMMPS [31] (with both rhodo (LMR) and colloid (LMC) inputs); two molecular simulation workloads,
- Gunrock on both road (GRU) and social networks (GST),
- DCGAN, neural style transform (NST) [32], reinforcement learning (RFL), spatial transformer (SPT) [33] and language translation (LGT) from PyTorch, and
- The following MLCommons benchmarks (from their MLPerf® Inference v2.0 collection):
  - The ResNet50 model [34],
  - Both MobileNet and ResNet34 variants of the SSD model,
  - The Bidirectional Encoder Representations from Transformers (BERT) [35], and
  - The 3D U-Net model [36]
- Four inputs to the 8x8 DCT implementation in the CUDA Samples (each labeled with their respective input).

In Figure 4.1, you can see the relative IPC difference for each kernel in the Cactus and MLPerf workloads. For the majority of workloads, the IPC difference is rather small (at most 10%). However, for some of the more interesting workloads (like DCG, GRU, NST, and DCT), we can find IPC differences of up to 70% for some kernels.

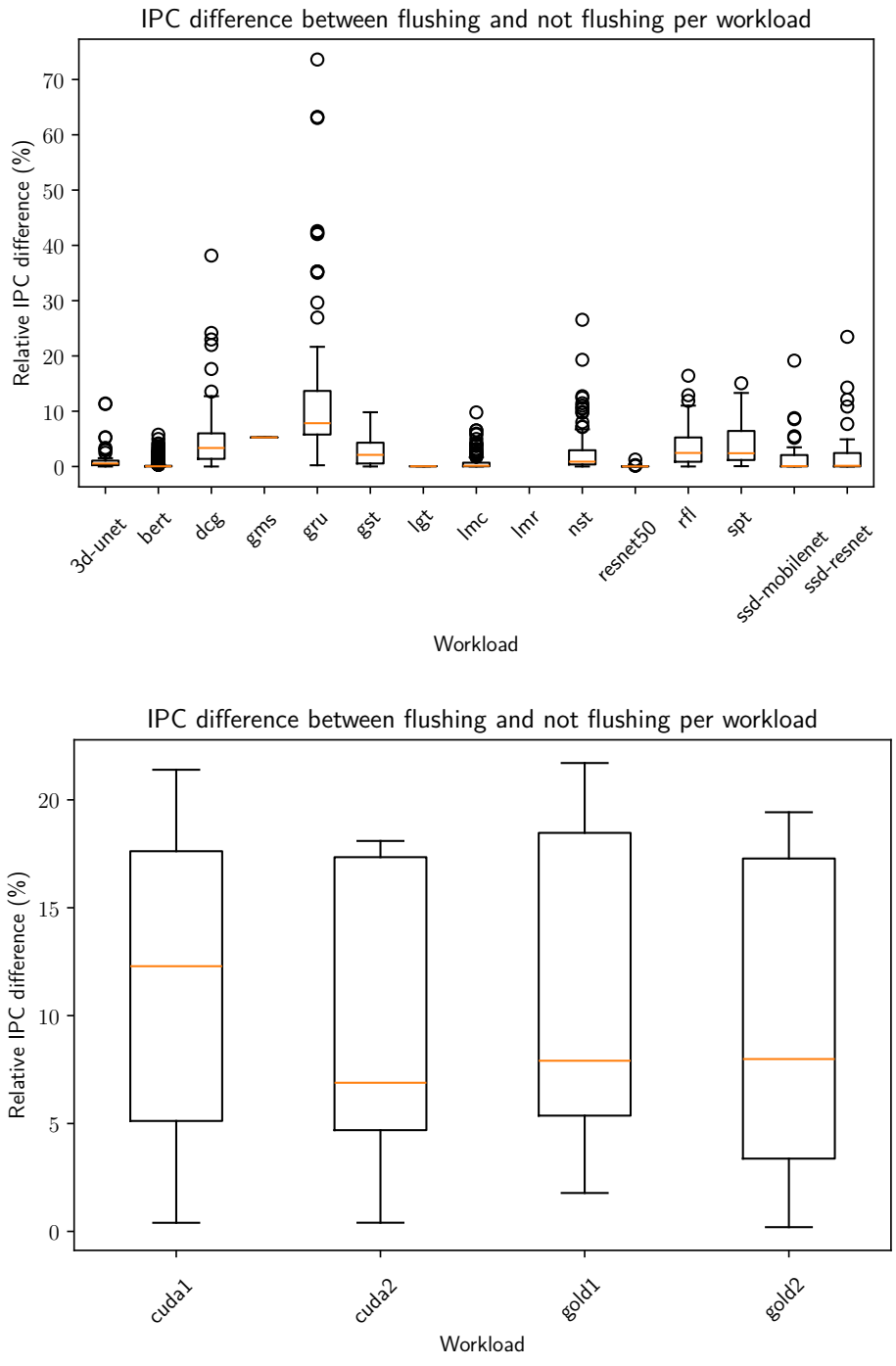


Figure 4.1: Relative IPC difference

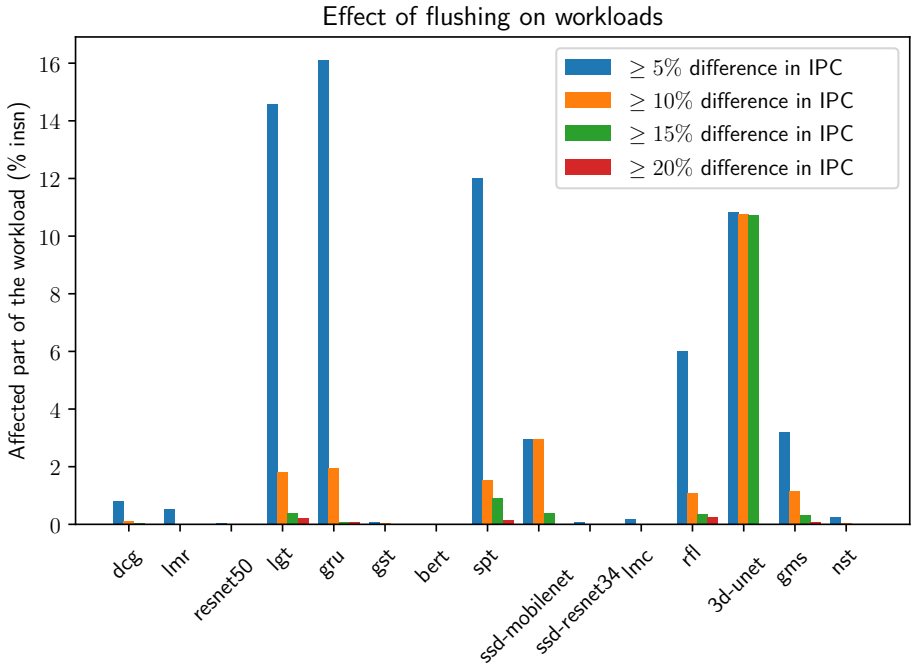
4.2 Weighting kernels

While this shows that the problem does exist in hardware, it might not relate to how modern simulation is carried out. Modern techniques like Sieve [13] use clustering to select a subset of kernels to simulate, generalizing the results to the entire workload. In these sampling techniques, many kernels are clustered together based on characteristics. The process then selects a single kernel from each cluster to simulate, using those results are representative for the entire cluster. To combine the clusters and compute a final result, each cluster's result is weighted by its instruction count, relative to the full workload:

$$w_{kernel} = \frac{\text{kernel instruction count}}{\sum_{\text{kernel } k \in \text{ workload}} \text{kernel}_k \text{ instruction count}}$$

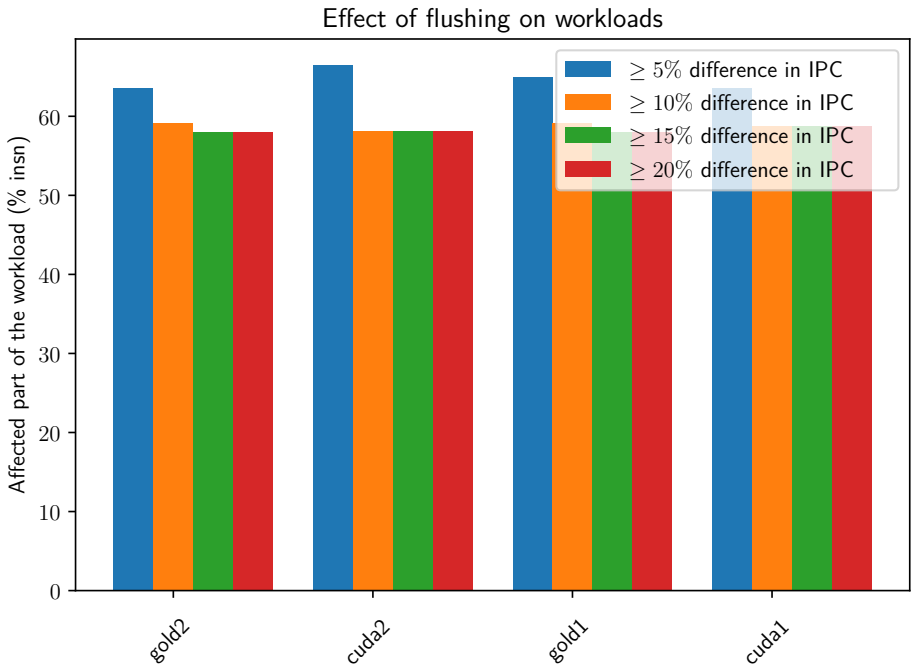
$$w_{cluster} = \sum_{\text{kernel } k \in \text{ cluster}} w_{kernel}$$

In order to get a better view of the impact of the cold start problem, we have also computed IPC difference when each kernel is weighted by its instruction count.



Workload	5% difference	10% difference	15% difference	20% difference
dcg	0.81 %	0.12 %	0.03 %	0.02 %
lmr	0.52 %	0.01 %	0.00 %	0.00 %
resnet50	0.04 %	0.01 %	0.01 %	0.01 %
lgt	14.57 %	1.80 %	0.40 %	0.21 %
gru	16.11 %	1.93 %	0.07 %	0.07 %
gst	0.08 %	0.04 %	0.00 %	0.00 %
bert	0.00 %	0.00 %	0.00 %	0.00 %
spt	12.00 %	1.53 %	0.91 %	0.13 %
ssd-mobilenet	2.96 %	2.94 %	0.40 %	0.00 %
ssd-resnet34	0.07 %	0.00 %	0.00 %	0.00 %
lmc	0.17 %	0.01 %	0.01 %	0.01 %
rfl	6.01 %	1.10 %	0.36 %	0.25 %
3d-unet	10.84 %	10.75 %	10.73 %	0.00 %
gms	3.21 %	1.17 %	0.33 %	0.08 %
nst	0.26 %	0.04 %	0.01 %	0.01 %

Figure 4.2: Weighted IPC differences for MLPerf and Cactus



Workload	5% difference	10% difference	15% difference	20% difference
gold2	63.62 %	59.17 %	58.00 %	58.00 %
cuda2	66.46 %	58.12 %	58.12 %	58.12 %
gold1	64.97 %	59.17 %	58.00 %	58.00 %
cuda1	63.52 %	58.80 %	58.80 %	58.80 %

Figure 4.3: Weighted IPC differences for DCT

In Figure 4.2, you can see the result of this analysis for the Cactus and MLPerf workloads. The results for the DCT workload (with its same four input images) is shown in Figure 4.3. We have set four thresholds for the relative IPC difference (5%, 10%, 15%, and 20%). For each of these thresholds, we have summed up the weights of all kernels where the IPC difference is at least that much. This means that e.g. for the GRU workload, approximately 16% of the entire workload suffers from a difference of at least 5%.

From these figures, we end up with a different set of affected workloads. Most of the MLPerf workloads only suffer slightly from the cold-start problem, with the only notable exception being the 3D U-Net workload. In the Cactus suite, we found language translation (LGT), spatial transformer (SPT), reinforcement learning (RFL), and Gunrock (with road input, GRU) to be the most affected. The real outlier here, however, is the DCT one. It consistently suffers from at least 20% relative IPC difference, no matter the input.

4.3 Data reuse

We assume that the cold-start problem is more prevalent in workloads with high data reuse. To verify this, we have analyzed the degree of inter-kernel data reuse in the DCT workload. For each kernel, we have profiled all memory instructions and extracted the unique memory addresses.

In Figure 4.4, you can see the data reuse ratio for the DCT workload. We have analyzed both the forward and backward data reuse for each kernel:

- **Forward data reuse:** the amount of unique memory addresses in kernel k_i that are reused by kernel k_i (as seen in Figure 4.4a); i.e.

$$\frac{|\text{footprint } k_i \cap \text{footprint } k_{i+1}|}{|\text{footprint } k_i|} \tag{4.1}$$

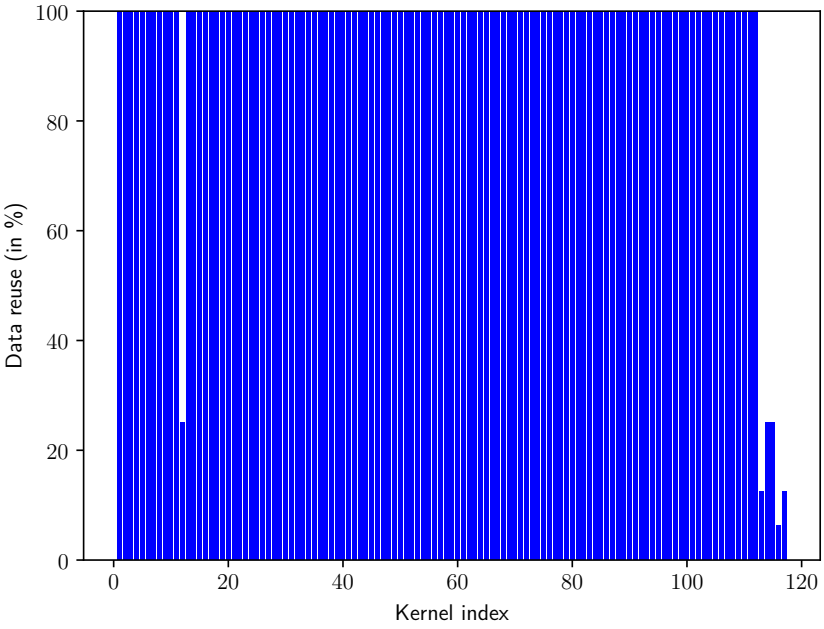
- **Backward data reuse:** the amount of unique memory addresses in kernel k_i that are also present in kernel k_{i-1} (as seen in Figure 4.4b); i.e.

$$\frac{|\text{footprint } k_{i-1} \cap \text{footprint } k_i|}{|\text{footprint } k_i|} \tag{4.2}$$

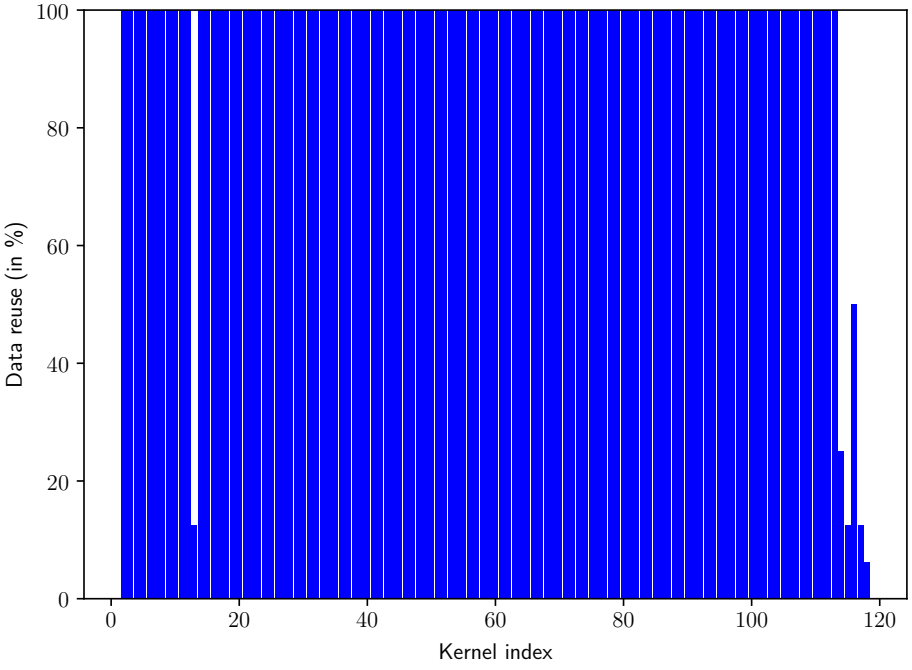
When looking at the DCT workload, most kernels have a very high degree of data reuse, both backward and forward. As we analyzed before, this workload suffers heavily from the cold-start problem (in hardware).

For contrast, we have also analyzed `recursiveGaussian` (from the CUDA SDK); as you can see in Figure 4.5. Roughly half of the kernels hit 50% of data reuse (in either direction), while the other half has a much lower degree of data reuse. This is also reflected in the IPC difference due to the cold-start problem: only about 1.3% of the workload suffers from at least 5% IPC difference.

The notion of data reuse, more specifically forward data reuse, will be used in the final chapter, when we discuss possible mitigations to the cold-start problem.

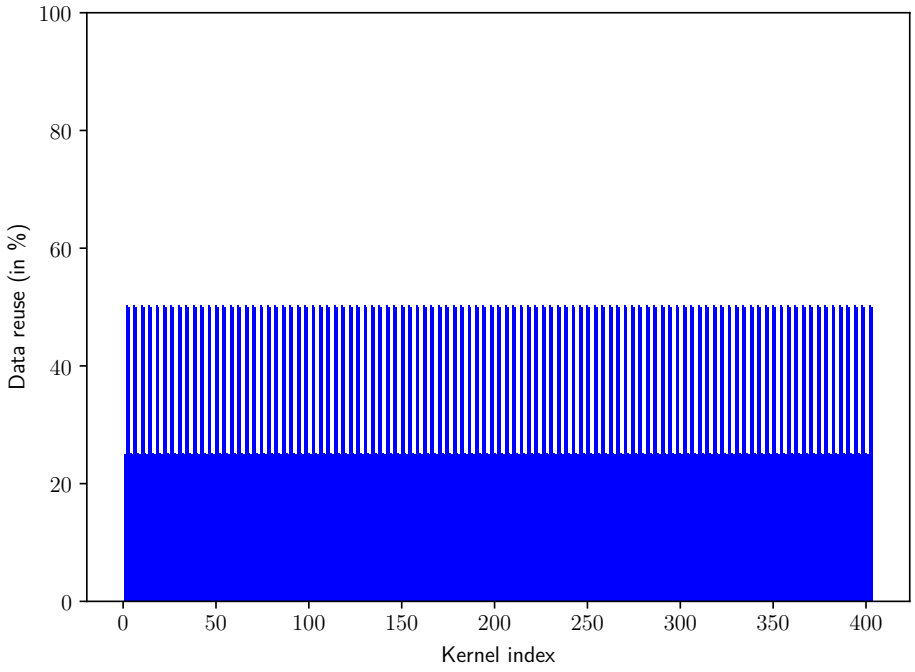


(a) Forward data reuse

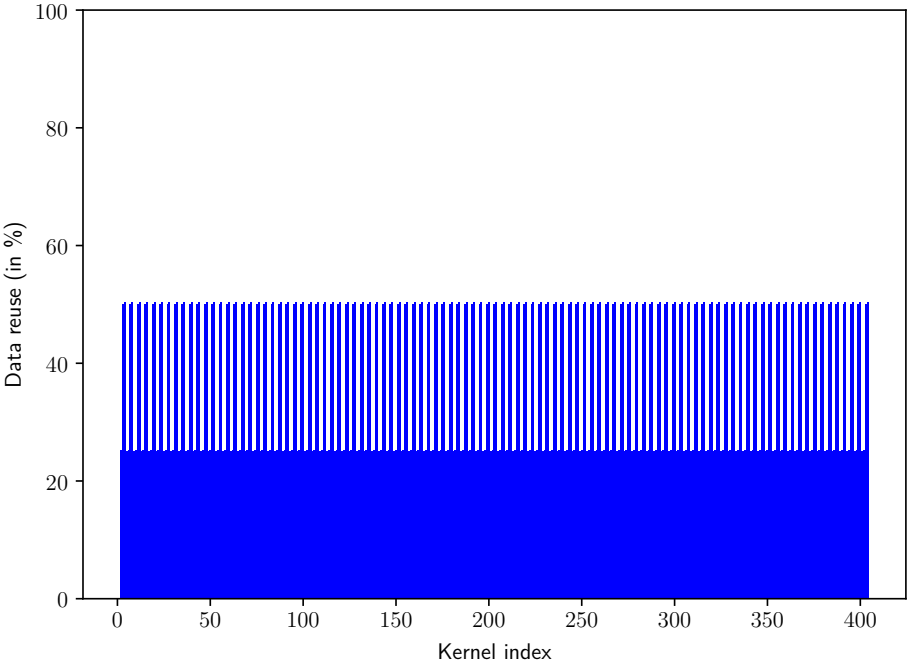


(b) Backward data reuse

Figure 4.4: Inter-kernel data reuse in DCT



(a) Forward data reuse



(b) Backward data reuse

Figure 4.5: Inter-kernel data reuse in recursiveGaussian

4.4 Hardware conclusion

From this, we can conclude that the cold-start problem does exist in hardware. Additionally, we noticed that the problem changes severity when we take each relative kernel's instruction count into account, giving us other workloads to focus on. Finally, we suspected that workloads with high inter-kernel data reuse would suffer more from the cold-start problem. This was confirmed by the DCT workload, as we have shown in Figure 4.4.

In the next chapter, we will be analyzing the impact of the cold-start problem in the AccelSim simulator. We'll mostly use the DCT and 3D U-Net workloads for this, as well as the OceanFFT one.

5 THE COLD-START PROBLEM IN ACCELSIM

We can now safely conclude that GPGPU hardware and workloads suffer from cold caches. In this chapter, we will attempt to validate the existence of the cold-start problem when simulating. To this end, we will use the AccelSim framework, with a similar configuration to the hardware we used in the previous chapter.

5.1 Simulation setup

The simulator we will be using is AccelSim[10]. As discussed in Section 2.1, AccelSim uses the GPGPU-Sim[9] system to perform actual simulation. However, we are able to speed everything up by avoiding full-functional simulation. To achieve this, we use pre-generated traces, which can be replayed in the simulator. This also means that we can just use compiled binaries, not needing any source at all.

In the experiments discussed below, we have used the GPGPU-Sim configuration for the NVIDIA GeForce RTX 3070 hardware (running on SM 86; as described in [27]). Some of its configuration parameters are shown in Figure 5.1.

Configuration parameter	Value
L2 cache size	4 MB
Number of sets in L2 cache	64
L2 cache block size	128 B
L2 cache associativity	16
Number of memory controllers	16
Number of SMs	46
L2 Latency	187 cycles
DRAM Latency	254 cycles

Figure 5.1: Simulator configuration parameters

Just like with the hardware analysis, we will be simulating each workload twice: once with the cache flushed between kernel invocations, and once without. This is a feature that AccelSim natively supports, using the `-flush-l1 -flush-l2` arguments.

5.2 Preparation

In order to simulate each workload, we needed to profile them again. In the previous chapter, we used NVIDIA's Nsight Compute tool, which is a lightweight profiler. However, we need a cycle-level trace for the simulator, which means that we will need something more powerful.

Preparing a workload for simulation has two big steps:

1. *Profiling:* AccelSim requires a cycle-level trace. We will discuss the exact process below, in Section 5.2.1.

2. *Post-processing:* the trace, as generated in the profiling phase, is not yet fit for simulation. AccelSim includes a post-processing tool, which allows us to convert the trace. We will also shortly discuss this in Section 5.2.1.

5.2.1 Profiling phase

One of the components of the AccelSim framework is an NVBit[15] tool.

NVBit is a framework, developed by NVIDIA, to instrument CUDA applications. It allows us to, among others, catch certain events (like kernel invocations), and insert additional instructions and calls in a kernel. Each NVBit tool is compiled into a shared library, which is then injected into the application. To this end, the `LD_PRELOAD` trick is used; which allows us to load a shared library before any other library.

AccelSim's tool is in the `util/tracer_nvbit/tracer_tool` directory. It roughly works like this:

1. The runtime will register that the NVBit tool has declared the `nvbit_at_cuda_event` function. At each CUDA event, this function will be invoked, with details on the event.

2. When the function is invoked, it checks if it is due to the invocation of a kernel that has not been instrumented yet. If this is the case, the tool will instrument the kernel by adding a call to `instrument_inst` before each instruction in the kernel. The arguments it passes to this function call depend on the instruction that is being instrumented.

3. Each time `instrument_inst` is called, it receives some information about the instruction that is to come. This information is written on a channel to another thread, which will write it to a file.

In this way, a directory with a trace file for each kernel is generated. These trace files contain an instruction-level trace for each kernel. Additionally, it generates `kernelslist`, an additional file containing a reference to each file generated.

However, due to interleaving of threads, these traces are not in the correct order yet for the simulator. The post-processing tool, in the `util/tracer_nvbit/tracer_tool/traces-processing` directory, will make sure that the traces are ready to be used. It reads each trace file listed in the given `kernelslist` file, sorting the instructions by thread block. This makes sure that the simulator can quickly access the instructions it needs to simulate a given thread block.

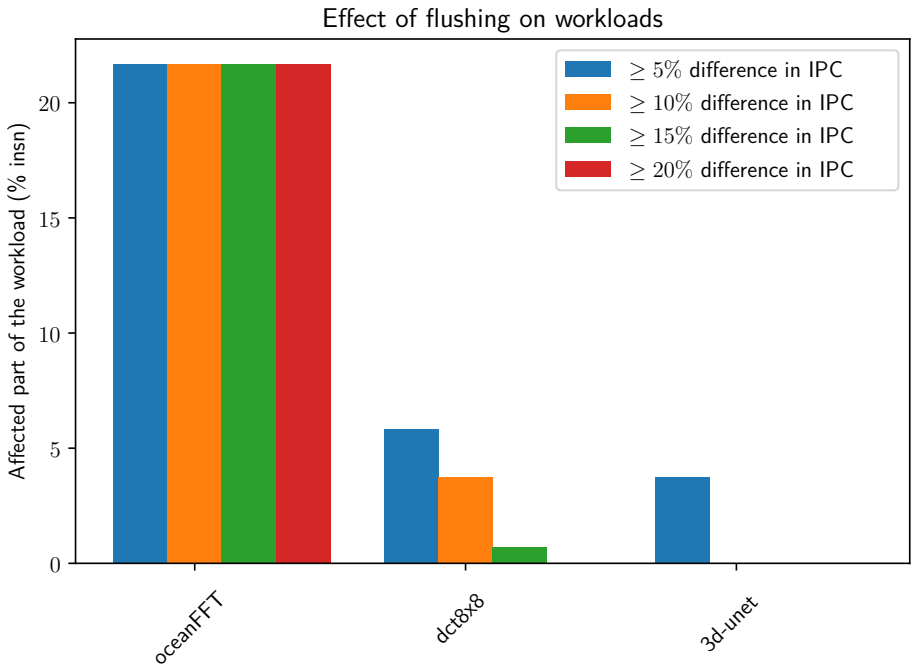
These sorted traces are each written to their own new file, while an additional `kernelslist.g` file is generated, referencing the processed traces.

5.3 Simulation results

In this section, we will show that the impact of this problem persists in the AccelSim simulator. However, one of the first things we have noticed is that the simulator is affected much differently from the real hardware.

Due to the inherent overhead in simulating GPUs, we had to limit the number of kernels we could simulate(in this case, up to 130, which fit the DCT nicely, but served as a cutoff for 3D-UNet), as well as which workloads we could focus on. The DCT and 3D U-Net workloads were selected for this analysis, as they showed promising results in the hardware analysis. Following our discussion about inter-kernel data reuse, we also included the OceanFFT workload (from the CUDA SDK) in this analysis.

The graphs in Figure 5.2 show the results of this analysis. They follow the same structure as Figure 4.2 from the hardware analysis: showing weighted IPC differences for each workload. The first thing we noticed is that the simulator results are very different from the hardware results. Where 3D U-Net had kernels with a relative IPC difference of over 15% in hardware, the differences in the simulator cap out around 5%. For DCT we notice a similar trend: the simulator results are much lower than the hardware results. Where the hardware results showed that more than 50% of the workload suffered from at least 20% in IPC difference, the simulator results give a maximum of 15% (worth less than 1% of the workload).



Workload	5% difference	10% difference	15% difference	20% difference
oceanFFT	21.69 %	21.69 %	21.69 %	21.69 %
dct8x8	5.84 %	3.75 %	0.69 %	0.00 %
3d-unet	3.75 %	0.00 %	0.00 %	0.00 %

Figure 5.2: Weighted IPC differences

5.3.1 OceanFFT

In this figure, the OceanFFT workload really stands out, compared to the other workloads. When looking into the structure of the OceanFFT workload, we noticed that it consists of only five kernels, of which one (kernel #2) is significantly affected by the cold-start problem. The other four kernels (kernel #1, #3, #4, and #5) are not affected by the cold-start problem at all (all of them suffering less than 0.5% relative IPC difference), as you can see in Figure 5.3. We also compared the data reuse factors for each of these kernels, and they are both very irregular, and a lot lower than DCT.

Kernel Name	IPC Difference	Weight	Forward Reuse (%)	Backward Reuse (%)	Memory footprint
`generateSpectrumKernel`	0.28%	44.43%	12.50%	n/a	33 564 671
`regular_fft`	*31.40%*	*21.69%*	*50.00%*	*25.00%*	*16 779 666*
`vector_fft`	0.45%	17.81%	0.00%	25.00%	16 780 062
`updateHeightmapKernel`	0.01%	6.00%	25.00%	0.00 %	16 777 216
`calculateSlopeKernel`	0.00%	10.08%	n/a	18.18%	46 137 344

Figure 5.3: OceanFFT kernels

In addition, we also noticed that OceanFFT suffers from another oddity: its kernels run faster with flushed caches. Especially the second kernel, whose IPC increases with almost 50% if we flush the caches, as you can see in Figure 5.4.

We assume that this might be caused by the fact that the second kernel only reuses a small fraction of memory addresses used by the first one. This means that the caches are not warmed up enough to be beneficial for the second kernel. Additionally, if a lot of cache lines are dirty, and need to be flushed, this might stall the memory pipeline. As you can see in Figure 5.3, the first kernel has a very large memory footprint (33.5 million unique addresses), which could easily flood the caches with data that is not useful for kernel 2. If a large part of this data has to be evicted (and, thus, possibly written back to DRAM), this can pile up in the memory hierarchy, causing the processors to stall.

Kernel Name	Instruction Count	IPC (flushing)	IPC (no flushing)	Forward Reuse (%)
_Z22generateSpectrumKernel	10 938 744 844	2482.26	2487.96	12.50%
_Z11regular_fft	5 339 348 992	640.58	439.84	50.00%
_Z10vector_fft	4 385 144 832	1401.63	1397.86	0.00%
_Z23updateHeightmapKernel_y	147 6395 008	683.88	684.10	25.00%
_Z20calculateSlopeKernel	2 482 536 508	1161.11	1160.73	n/a

Figure 5.4: OceanFFT IPC values

5.4 Simulator conclusion

Once again, we see that the cold-start problem persists. However, it is much less severe in the simulator; the maximum impact of the cold-start problem on 3D-UNet drops from over 20% to around 15%. For DCT, this shows even more, as a large part of the workload (just shy of 60%) suffered from at least 20% of IPC difference in hardware, while in the simulator, not a single kernel suffers that much. Instead, the differences cap out around 15%, with less than 1% of the workload suffering from that.

We do not know what caused this discrepancy between the hardware and the simulator, however, we have a hypothesis. Perhaps there are some hardware details that get lost in the simulator, which is a slightly more idealized version of the hardware. Additionally, a lot of hardware details have to be gathered from micro-benchmarks [10], because vendors keep these details secret. While both the hardware (NVIDIA GeForce RTX 3080) and simulator (NVIDIA GeForce RTX 3070) used the same micro-architecture, there might be some subtle details that impacted the cold-start problem. Finally, our hardware experiments (and the simulation) ran using NVIDIA's Ampere architecture, which is not among the heavily tested architectures in the original Accel-Sim paper.

In the next chapter, we will start looking for a mitigation, limiting the impact of the cold-start problem. We will mostly analyze the DCT workload, due to its short runtime and thus reasonable runtime. Two main factors will exclude the OceanFFT workload from this analysis: firstly, the most interesting kernel is the second one, limiting one of our mitigations. Secondly, its oddity (the fact that it runs faster with flushed caches) makes it a bad candidate for our second mitigation.

6 MITIGATION

So far, we have talked about the cold-start problem in both hardware and simulation. The reason why we have looked into it, is because of how we approach simulating large workloads in GPUs. Instead of simulating each and every kernel, we only simulate a select subset. Usually, these are selected using both profiling and some form of clustering based on characteristics. However, since we only simulate intermittent kernels, cache state might be lost and/or incorrect; which leads to inaccuracies.

Taking this all into account means that we see a few options for mitigation:

- **Simulate preceding kernels:** this is the most naive, but also the most computationally intensive approach. Simulating all instructions from one or more kernels that come right before the one we need ensures that the cache state is as close to the real state as possible, but also increases the simulation time.
- **Simulate certain instructions from preceding kernels:** this option is more refined than the previous, finding a balance between computation time and accuracy. Since we have the full trace of each kernel, we can select only the memory instructions from the preceding kernels, simulating those to warm up the caches artificially. Especially when kernels contain a lot of computations, and fewer memory loads/stores, this could allow for a higher accuracy at a very low cost.
- **Compute a correction factor:** logic dictates that there should be a way to compute a correction factor for the cold-start problem. There are different factors that we could analyze ahead-of-time to compute this factor.

In Sections 6.1 and 6.2, we will focus on the second option, opting to simulate memory instructions in order to artificially warm up the caches. Section 6.3 will focus on the third option, offering some possible factors that could be used to compute a correction factor.

6.1 Gathering trace info

The AccelSim [10] tool comes with an NVBit [15] tool and post-processor which are already used to generate the instruction-level traces that are eventually fed to the simulator itself. By default, this tool generates a number of files:

- `traces/kernelslist`: a list of all kernels that were run, with their respective kernel IDs.
- `traces/stats.csv`: a CSV file containing some global statistics about each kernel.
- `traces/kernel-<number>.trace`: the actual instruction-level trace for each kernel, identified by their numbers.

We have expanded this tool to output an additional file for each kernel: `traces/kernel-mem-<number>.trace`. This file contains all memory instructions issued by the selected kernel, as well as the final `EXIT` instruction (omitting this one would lead to a segmentation fault in the simulator). The original NVBIT tool reports for each instruction it instruments whether it is a memory instruction or not. We use that information to filter out the instructions we need.

Kernel Index	Kernel Name	IPC Difference (%)
11	_Z27CUDAkernelQuantizationFloatPfi	13.42
13	_Z14CUDAkernel2DCTPfS_i	12.17
114	_Z27CUDAkernelQuantizationFloatPfi	10.36
115	_Z15CUDAkernel2IDCTPfS_i	15.05

Figure 6.1: High IPC difference kernels in DCT

As with the original traces, we need to post-process each memory trace before feeding it to the simulator. This post-processing step sorts the instructions by their thread block, allowing the simulator to easily access the next instruction. By using the same format as the original tool, we ensure that the existing post-processor can also handle these new files[1].

6.2 Kernel selection

In order to warm up the simulator's caches, we will be simulating the memory instructions from preceding kernels. However, this is once again a trade-off: the more kernels we use to warm up the caches, the more accurate the simulation will be, but the longer it will take. We have selected four kernels from the DCT8x8 workload with high IPC differences, as shown in Figure 6.1.

For each of these kernels, we used multiple simulations:

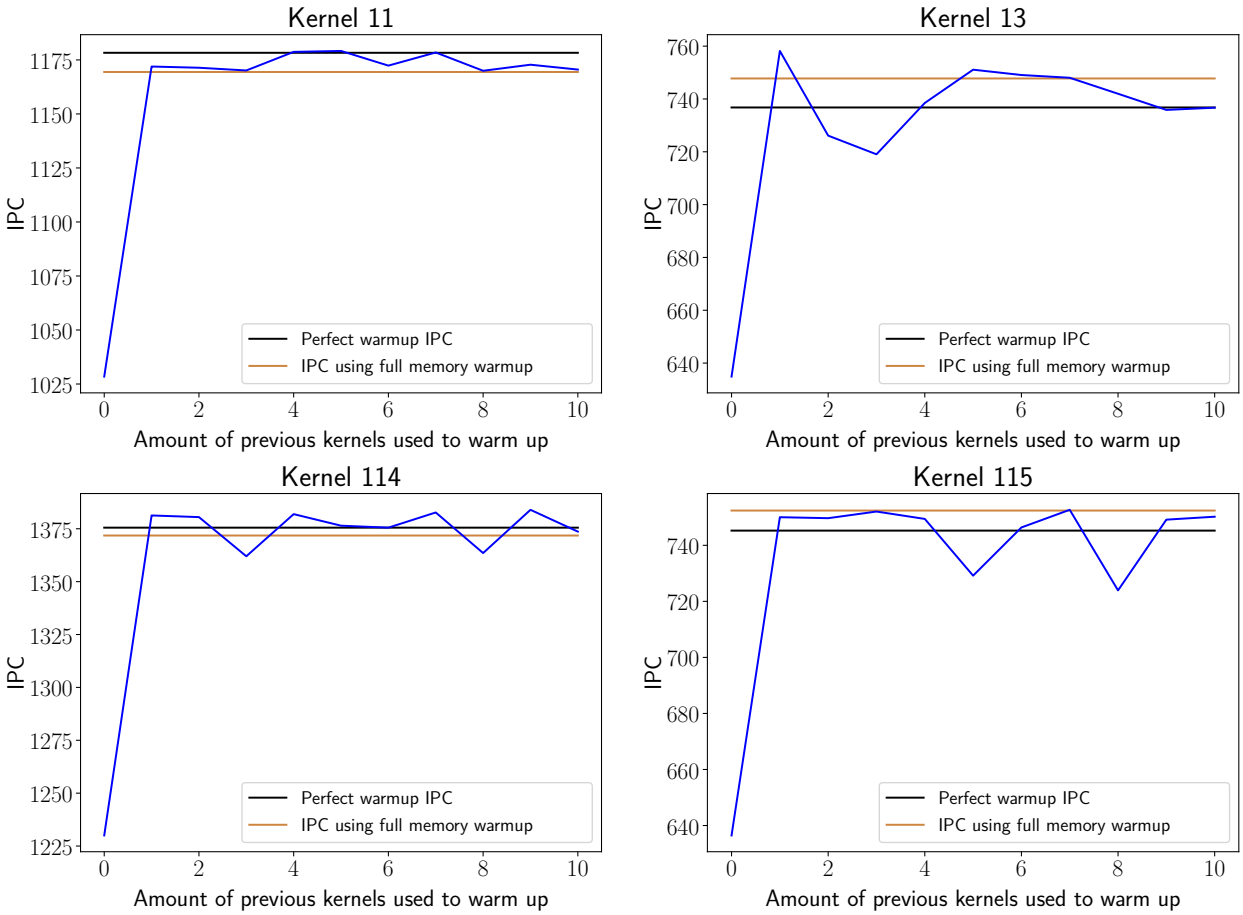
- **Perfect warmup:** the kernel is simulated in order, with all preceding kernels simulated in full.
- **Full memory warmup:** the kernel is simulated in order, with all memory instructions preceding it simulated (for kernel i, this means all memory instructions from kernels 1 until $i-1$ are simulated).
- **Partial memory warmup:** the memory instructions of up to 10 preceding kernels are simulated.

In Figures 6.2 and 6.3, we have plotted the results of these simulations. For each kernel, we have plotted the result of *perfect warmup* as a black line, this was the golden reference we tried to reach. Additionally, the light-brown line represents *full memory warmup*, while the blue line represents *partial memory warmup*. Both raw IPC values and accuracy (in percents) are shown.

From these figures, we can quickly deduce that even a single preceding kernel can lead to drastic accuracy increases: most kernels reach over 99% accuracy with just one kernel warmed up.

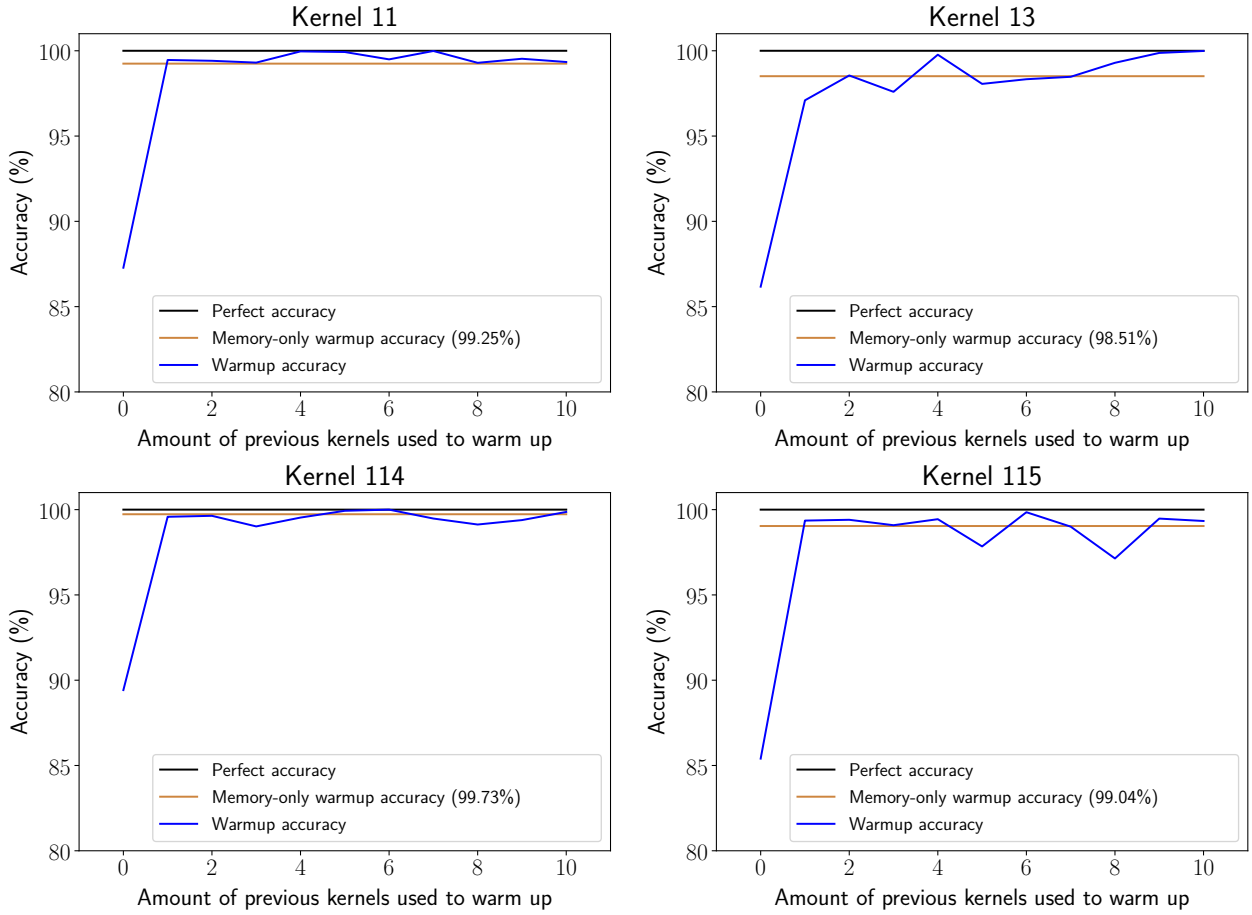
An additional striking detail is that more kernels does not always imply a higher accuracy. This might be due to some non-determinism in the simulator.

[1] *Note:* AccelSim by default uses a compressed format for memory instructions, taking up less disk space in their traces. However, due to time constraints, we have elected to work with the uncompressed format, which takes up more disk space.



Kernel	Kernel 11	Kernel 13	Kernel 114	Kernel 115
0 warmup kernels	1028.44	634.89	1230.01	636.46
1 warmup kernel	1171.94	758.18	1381.29	750.03
2 warmup kernels	1171.38	726.12	1380.51	749.68
3 warmup kernels	1170.12	719.05	1361.99	752.05
4 warmup kernels	1178.72	738.48	1381.92	749.42
5 warmup kernels	1179.15	751.09	1376.45	729.17
6 warmup kernels	1172.36	749.07	1375.52	746.37
7 warmup kernels	1178.44	748.02	1382.70	752.67
8 warmup kernels	1169.98	741.97	1363.52	723.91
9 warmup kernels	1172.79	735.85	1383.96	749.16
10 warmup kernels	1170.54	736.70	1373.66	750.21
Full memory warmup	1169.42	747.76	1371.81	752.40
Perfect warmup	1178.30	736.78	1375.52	745.25

Figure 6.2: DCT Mitigation results (absolute IPC values)



Kernel	Kernel 11	Kernel 13	Kernel 114	Kernel 115
0 warmup kernels	87.28%	86.17%	89.42%	85.40%
1 warmup kernel	99.46%	97.10%	99.58%	99.36%
2 warmup kernels	99.41%	98.55%	99.64%	99.40%
3 warmup kernels	99.31%	97.59%	99.02%	99.09%
4 warmup kernels	99.96%	99.77%	99.54%	99.44%
5 warmup kernels	99.93%	98.06%	99.93%	97.84%
6 warmup kernels	99.50%	98.33%	100.00%	99.85%
7 warmup kernels	99.99%	98.47%	99.48%	99.00%
8 warmup kernels	99.29%	99.30%	99.13%	97.14%
9 warmup kernels	99.53%	99.87%	99.39%	99.48%
10 warmup kernels	99.34%	99.99%	99.86%	99.33%
Full memory warmup	99.25%	98.51%	99.73%	99.04%

Figure 6.3: DCT Mitigation results (accuracy)

6.2.1 Full Memory Warmup

A striking result from the figures above, is that the *full memory warmup* approach does not yield 100% accuracy. The cause of this might be the ordering of instructions; since we remove all non-memory instructions, they might get executed in a different order. This could lead to different cache states, and thus different IPC values.

In the example in Figure 6.4, we have two threads executing. Each of these has a different pattern: one starts with some memory instructions, performing arithmetic afterward, while the other does the opposite. We assume that each instruction takes the exact same time, ignoring latencies as well. This causes all memory instructions to be executed serially, resulting in a certain cache state.

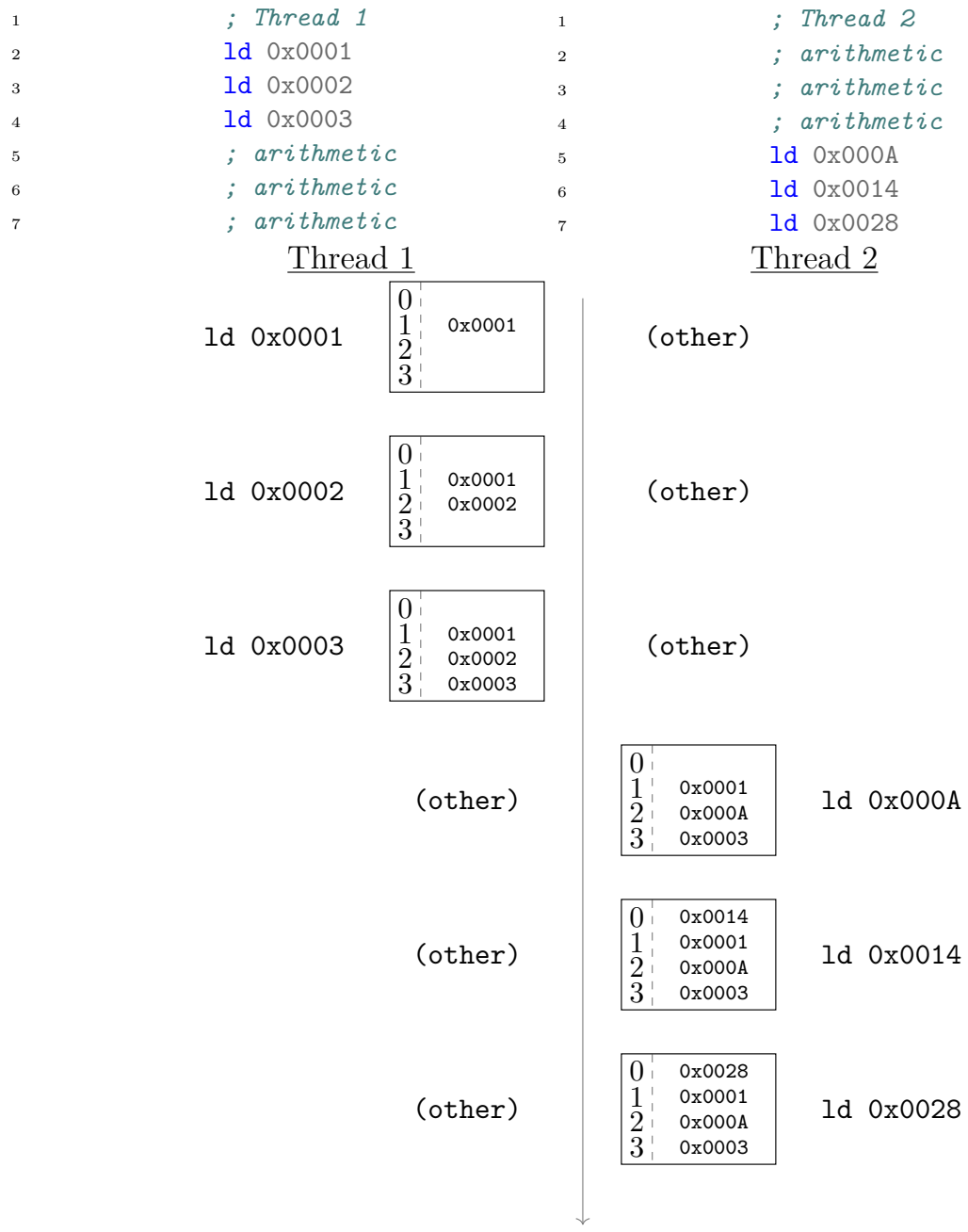


Figure 6.4: Interleaving example

When we execute only the memory instructions in the "trace" above, we get a whole different execution. See Figure 6.5 for the result of that execution.

When we compare the final cache state of each execution (see also Figure 6.6), we can see a clear difference. This can impact the performance of the next kernel: for the first execution (a perfect

```
1  ; Thread 1
2  ld 0x0001
3  ld 0x0002
4  ld 0x0003
```

```
1  ; Thread 2
2  ld 0x000A
3  ld 0x0014
4  ld 0x0028
```

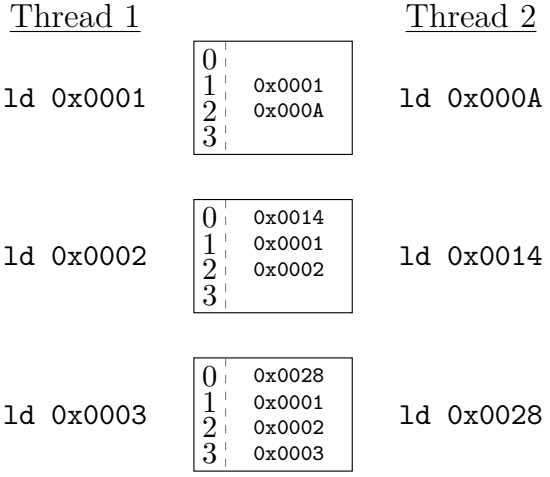


Figure 6.5: Interleaving example - memory only

warmup), accessing `0x000A` would result in a cache hit, while `0x0002` would miss. The opposite happens when we only simulate memory instructions (full memory warmup, like in execution 2). This can explain the slight inaccuracy we see for full memory warmup.

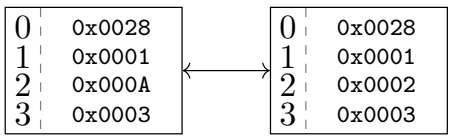


Figure 6.6: Interleaving example - cache state differences

6.3 Correction factor

Since the cold-start problem is inherently tied to the cache state, we should look at factors that can impact this state in order to find a correction factor. Tools like NVIDIA's Nsight Compute [14] and NVBit [15] can provide detailed cache statistics, which might be used to compute this factor. Some possible factors we considered are:

- *Profiler differences:* by profiling a workload twice, once with and once without flushing, one can see the impact of the cold-start problem in silicon. This data might be used to improve simulation results.
- *Profiler cache statistics:* a detailed profiler can provide statistics about hits and misses, which in turn could be used to compute the correction factor.
- *Degree of data reuse:* by using an instruction-level trace, it is possible to extract all memory references. The ratio $\frac{\text{total memory references}}{\text{unique memory references}}$ could inform us about the cache state. We have outlined the main ideas of *forward data reuse* in Section 4.3 (that section also includes the notion of *backward data reuse*, but that will prove not as useful here).
- *The DRAM delay:* the difference in cycles between a cache hit and a DRAM access could be useful to get an idea of the amount of cycles lost due to cold-start.

- *The number of memory controllers:* since each memory controller can issue requests in parallel, the number of controllers might impact the cache state, as well as the cost of a cache miss.
- *L2 miss rate:* this value could be used together with the *number of memory instructions*.

Some of these factors will be used in Section 6.3.1, where we outline a formula that seems to perform rather well. All the data used for that formula can be gathered by simulating a single kernel, and/or profiling the workload.

However, we have to be careful when finding a correction factor. Since some factors that could impact cache state (e.g. the number of cache evictions) are hardware- and platform-dependent, we could end up with a correction factor that only works for a specific platform.

6.3.1 A Possible Formula

Through analysis of the data, we have come up with a formula that allows use to compute a correction factor. This correction factor proved very effective, resulting in an average 95% accuracy for the DCT workload. Especially for problematic kernels (kernels suffering from the cold-start problem), this correction factor succeeded in improving accuracy.

The main goal is to compute a subtractive factor for the number of cycles, i.e. provide an upper bound on how many cycles could have been saved. To this end, we will use the following metrics, all of which can be gathered by simulating a single kernel and/or profiling the workload:

- **Unique DRAM accesses:** we modified Accel-Sim to output each DRAM request. By filtering these on unique values, we know the number of cold misses.
- **Memory controller occupation:** since the memory controllers in the system are able to process requests in parallel, the number of controllers might impact the number of cycles lost due to latency. Accel-Sim outputs their occupancy by default (the lines starting with `dram[]`). Importantly, we also take into account how evenly they are used; see Section 6.3.1.
- **Cache line size:** the DRAM accesses contain byte-level memory addresses. However, multiple consecutive addresses map to the same cache line, which might avoid some cold misses.
- **Reuse factor:** we use the *forward reuse factor* from the *previous* kernel. This value acts as a correction on the number of cold misses: if only a few memory addresses are reused, most cold misses aren't due to the cold-start problem.
- **Latencies:** the number of cycles saved is directly related to the DRAM- and L2-latency of the system.

Combining all of these, we came up with the following formula:

$$IPC = \frac{instructions}{cycles_f - \Delta} \tag{6.1}$$

$$\Delta = \frac{accesses}{controllers * line} \cdot fwd_{i-1} \cdot (DRAM - L2) \tag{6.2}$$

Here, IPC is the final (corrected) IPC value, $instructions$ is the number of dynamically executed instructions, $cycles_f$ is the number of cycles when simulating the kernel with cold caches (in flushed mode), and Δ is the correction factor. We use $accesses$ for the number of unique DRAM accesses, $controllers$ for the memory controller occupancy, $line$ for the size of a cache line, and $reuse$ for the (forward) reuse factor. $DRAM$ and $L2$ are the latencies of the respective memory regions.

This formula was tested on the DCT workload and improved on the sensitive kernels (11, 13, 114, and 115). The other kernels suffered from a slight drop in accuracy, when compared to their raw accuracy (i.e. using no correction factor at all).

Note: the formula above cannot be used for OceanFFT, as it uses a subtractive factor. In OceanFFT, we would need to increase the number of cycles (since starting from a cold cache increases the IPC for some kernels).

In Figure 6.7, we have gathered some results for the DCT workload. For a number of kernels, the accuracy of the formula is compared to the raw accuracy (meaning the flushed accuracy, without any corrections). As you can see, the overall accuracy took a slight hit (dropping from 98% to around 96%), while the sensitive kernels improved (from 89% to 93%). In Section 6.4, we will compare the accuracy of this approach against the accuracy for memory warmup (with a single kernel).

Index	IPC (flushing)	IPC (no flushing)	IPC (corr.)	Δ	Accuracy (raw)	Accuracy (corr.)
1	1912.75	1904.84	1912.75	0	99.58%	99.58%
2	1896.46	1899.59	1944.35	268	99.84%	97.64%
3	1891.79	1894.21	1939.43	268	99.87%	97.61%
4	1882.67	1892.82	1929.85	268	99.46%	98.04%
5	1897.33	1889.02	1945.26	268	99.56%	97.02%
10	1869.58	1889.89	1916.10	268	98.93%	98.61%
11	*1035.63*	*1174.34*	*1066.00*	*268*	*88.19%*	*90.77%*
12	1831.25	1825.83	1879.62	268	99.70%	97.05%
13	*640.14*	*719.45*	*696.00*	*805*	*88.98%*	*96.74%*
14	739.50	731.83	763.08	268	98.95%	95.73%
15	743.09	715.44	766.91	268	96.14%	92.81%
110	744.12	744.12	768.01	268	100.00%	96.79%
111	745.07	743.86	769.02	268	99.84%	96.62%
112	747.15	738.05	771.24	268	98.77%	95.50%
113	755.86	757.64	780.52	268	99.76%	96.98%
114	*1248.03*	*1380.66*	*1283.30*	*268*	*90.39%*	*92.95%*
115	804.32	755.06	811.65	72	93.48%	92.50%
116	689.91	735.53	700.14	135	93.80%	95.19%
117	1011.89	1047.26	1022.41	134	96.62%	97.63%
118	702.35	767.67	712.56	134	91.49%	92.82%
Average accuracy (entire workload)					98.16%	95.92%
Average accuracy (suffering kernels)					89.19%	93.49%

Figure 6.7: DCT mitigation results

The exact values are slightly different from the ones used in Section 6.2, but the general trend is the same. This is due to the inherent non-determinism in the simulator, which can lead to slightly different results for the same kernel.

Memory controllers

Not all workloads use all memory controllers evenly - in the worst case, a single controller does all the work while the others are idle. In that case, it would be incorrect to use the total number of controllers in the formula. To mitigate this, we compute the evenly used controllers, and use that value in the formula.

AccelSim outputs memory controller data under the form of number of DRAM requests made, per controller, per bank. Additionally, this data is output for both reads and writes.

We compute the evenly used controllers from the average occupancy of each controller (occ_{ctrl}) as follows:

$$occ_{ctrl} = \frac{\sum_{i=1}^{banks} reads_{ctrl,i} + \sum_{i=1}^{banks} writes_{ctrl,i}}{2 \cdot banks} \tag{6.3}$$

$$controllers = \frac{\sum_{ctrl} occ_{ctrl}}{\max_{ctrl} occ_{ctrl}} \tag{6.4}$$

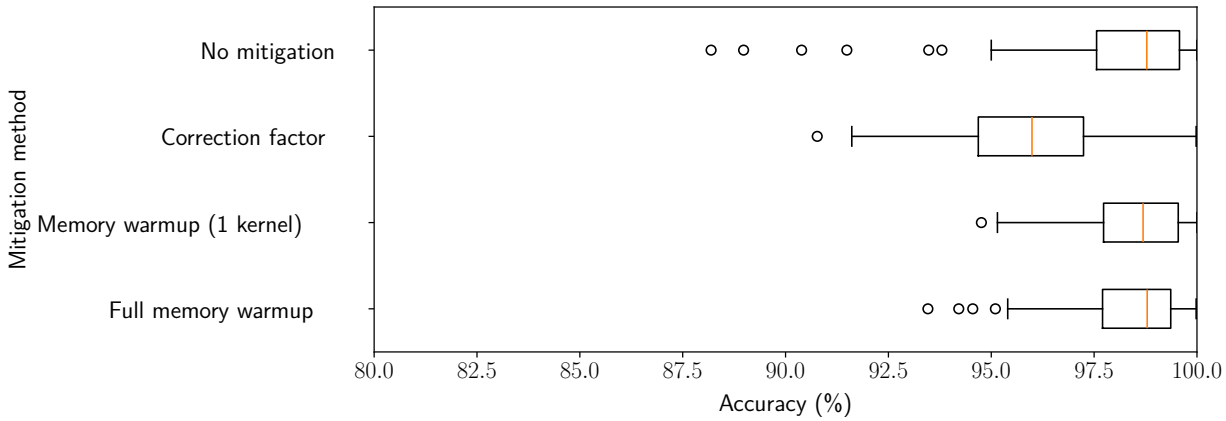
Where:

- occ_{ctrl} is the average occupancy of a controller $ctrl$;
- $banks$ is the number of memory banks;

- $reads_{ctrl,i}$ is the number of read accesses made by controller *ctrl* to bank i;
- $writes_{ctrl,i}$ is the number of write accesses made by controller *ctrl* to bank i; and
- *controllers* is the number of evenly used controllers (the final value used in the formula for Δ).

For DCT, this value was around 8 in our experimental setup, meaning that 8 out of the 16 controllers were used. OceanFFT, however, used almost all controllers evenly.

6.4 Comparison



Method	Minimum	Q1	Median (std. dev.)	Q3	Maximum
No mitigation	88.19%	97.56%	98.78% (2.10)	99.58%	100.00%
Correction factor	90.77%	94.69%	95.99% (1.98)	97.24%	99.98%
Memory warmup (1 kernel)	94.76%	97.73%	98.69% (1.20)	99.55%	100.00%
Full memory warmup	93.46%	97.70%	98.79% (1.33)	99.36%	99.98%

Figure 6.8: Comparison of mitigation methods

In Figure 6.8, we see that both mitigation approaches are able to eliminate the (negative) outliers. However, since there are so few suffering kernels, the median doesn't improve by a lot. In general, we can conclude that either mitigation approach is able to improve the accuracy of the simulator.

Additionally, we also included minimum accuracy, the first quartile, the median, the standard deviation, the third quartile, and the maximum accuracy for each of these methods. Due to the elimination of outliers, the standard deviation is lower for both mitigation methods, compared to the raw accuracy.

Due to temporal locality, we expect the impact of a kernel on the caches to be lower the further back in time it is. This is also reflected in the results: Figure 6.8 clearly shows that full memory warmup has no practical benefits over just using a single kernel to warm up the caches. Worse still, all metrics are slightly worse for full memory warmup, compared to partial memory warmup.

6.5 Conclusion

We have looked into three possible mitigations, and analyzed their strengths and weaknesses. First, there is the naive approach of simulating all preceding kernels in full, guaranteeing a correct warmup, and thus a "perfectly accurate" simulation. However, this has the drawback of being incredibly computationally intensive, negating the work done by sampling kernels in the first place.

Secondly, we have looked into simulating only parts of the previous kernels. In this case, we have selected the memory instructions of preceding kernels and simulated those to artificially warm up the caches. Even for kernels that suffered a lot from the cold-start problem, we have shown that

warming up using a single kernel has a drastic impact on the accuracy already. In many cases, a single kernel was enough to reach over 99% accuracy.

Lastly, we have looked into computing a correction factor. The factor we proposed is based on the number of cycles that could be saved, taking into account parallel requests and DRAM information. This factor has less of a computational overhead (since no additional traces need to be simulated), but resulted in a slightly lower accuracy than the memory warmup approach. However, we need to be careful not to over-tune this factor, as it could limit the simulator to a specific platform (beating the purpose of GPU simulation in the first place).

In general, both approaches are able to mitigate the cold-start problem, and improve the accuracy of the simulator, especially for kernels that do suffer from it. Either method was able to raise the accuracy of each DCT kernel to at least 90%, with the memory warmup reaching up to almost 95%.

7 CONCLUSION

In this thesis, we have analyzed the cold-start problem in GPUs. We have shown that the cold-start problem is a real issue, affecting the performance of GPUs in a significant way. We have also shown that the cold-start problem is not limited to a single workload, but that it can affect a wide range of workloads. In this chapter, we will present our conclusions, discussing both quantification and mitigation.

7.1 Quantification

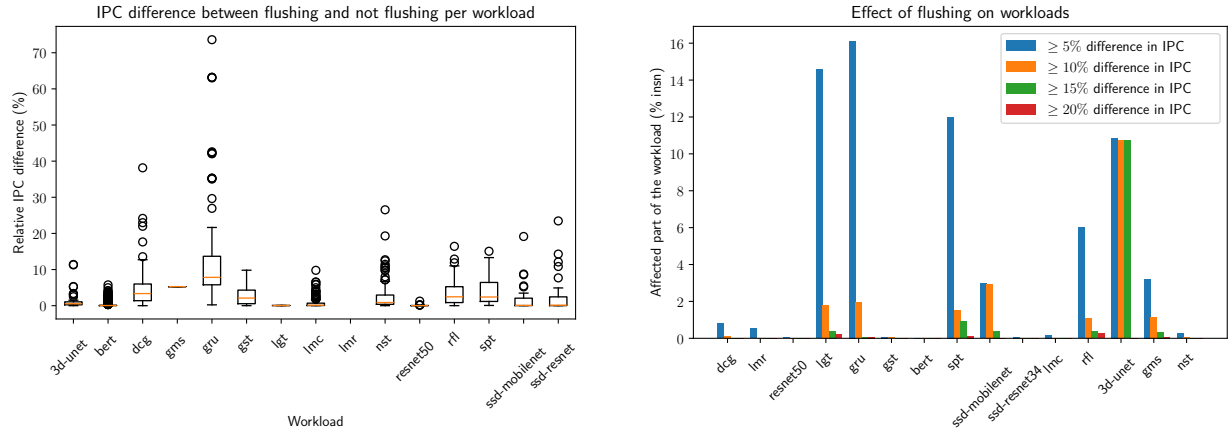
We started by analyzing whether the cold-start problem is an actual issue. To this end, we analyzed multiple workloads on real hardware, and a limited number of them in the AccelSim simulator. From this, we have concluded that the cold-start problem is a real issue, affecting the performance of GPUs in a significant way.

7.1.1 The cold-start problem in hardware

We have analyzed the cold-start problem in real hardware, and have shown that it is a real issue. Multiple workloads have shown a significant impact from the cold-start problem, see Figure 7.1a.

Additionally, we have weighted each kernel based on its instruction count, and have seen that its severity changes based on the kernel's weight. Taking into account is important because sampling methods like *Sieve* uses these weights to select the kernels to simulate, see Figure 7.1b.

For your reference, we have included the main plots from Chapter 4 below, in Figure 7.1.



(a) Unweighted impact

(b) Weighted impact

Figure 7.1: Impact of the cold-start problem in hardware

Finally, we also introduced the notion of *forward* and *backward data reuse*. These were defined as the number of unique memory instructions that are reused. For any given kernel k_i, we defined M_i as the set of memory instructions used by k_i. From this, we can compute both *forward* (fwd_i) and *backward data reuse* (bwd_i) as:

$$fwd_i = \frac{|M_i \cap M_{i+1}|}{|M_i|} \tag{7.1}$$

$$bwd_i = \frac{|M_{i-1} \cap M_i|}{|M_i|} \tag{7.2}$$

7.1.2 The cold-start problem in simulation

From all workloads we have analyzed in the hardware, we have selected a subset to analyze in the AccelSim simulator. The main ones we have analyzed were DCT, 3D U-Net, and OceanFFT.

From this analysis, we have concluded that the cold-start problem is less severe in the simulator. The maximum impact of the cold-start problem on 3D-UNet drops from over 20% to around 15%.

Once again, we have included the main plot from Chapter 5 in Figure 7.2.

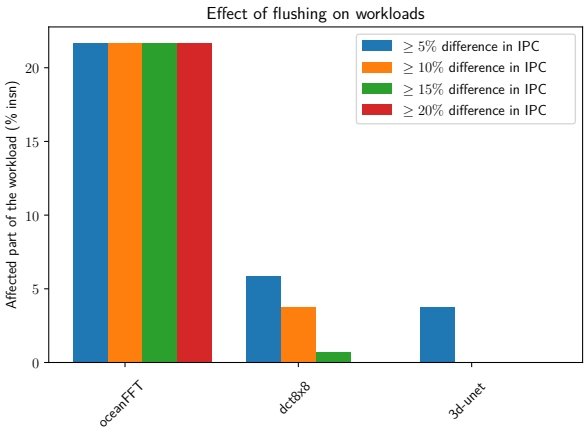


Figure 7.2: Impact of the cold-start problem in simulation

OceanFFT

The OceanFFT workload really stands out, compared to other workloads. Strangely enough, the simulator shows that it runs faster when the caches are flushed, as you can see in Figure 7.3.

Kernel Name	Instruction Count	IPC (flushing)	IPC (no flushing)	Forward Reuse (%)
`_Z22generateSpectrumKernel`	10 938 744 844	2482.26	2487.96	12.50%
`_Z11regular_fft`	5 339 348 992	640.58	439.84	50.00%
`_Z10vector_fft`	4 385 144 832	1401.63	1397.86	0.00%
`_Z23updateHeightmapKernel_y`	147 6395 008	683.88	684.10	25.00%
`_Z20calculateSlopeKernel`	2 482 536 508	1161.11	1160.73	n/a

Figure 7.3: OceanFFT oddity

This will prove problematic for one of our mitigations.

7.2 Mitigation

The last chapter of this thesis, Chapter 6, was dedicated to finding a mitigation for the cold-start problem. We have identified three different avenues, two of which proved to be practical. Finally, we have also compared their effectiveness.

The three avenues discussed were:

- **Simulating preceding kernels in full:** This method involves ensuring a consistent, warm hardware state by simulating preceding kernels in full. However, this is quite infeasible, and goes against the idea of sampling (which is used to speed up simulation).
- **Memory-only simulation of preceding kernels:** We only simulate the memory instructions of preceding kernels. This is a very feasible and accurate method, partly thanks to memory locality.

- **Correction factor:** We have also proposed a correction factor, which is based on the data reuse of the kernels, among others. This removes the overhead of any additional simulation, at the cost of a slightly lower accuracy.

7.2.1 Memory-only simulation

From the DCT workload, we have selected four kernels to analyze. For each of those kernels, we simulated the memory instructions of up to 10 previous kernels. To be able to do this, we have modified AccelSim's builtin NVBit tool.

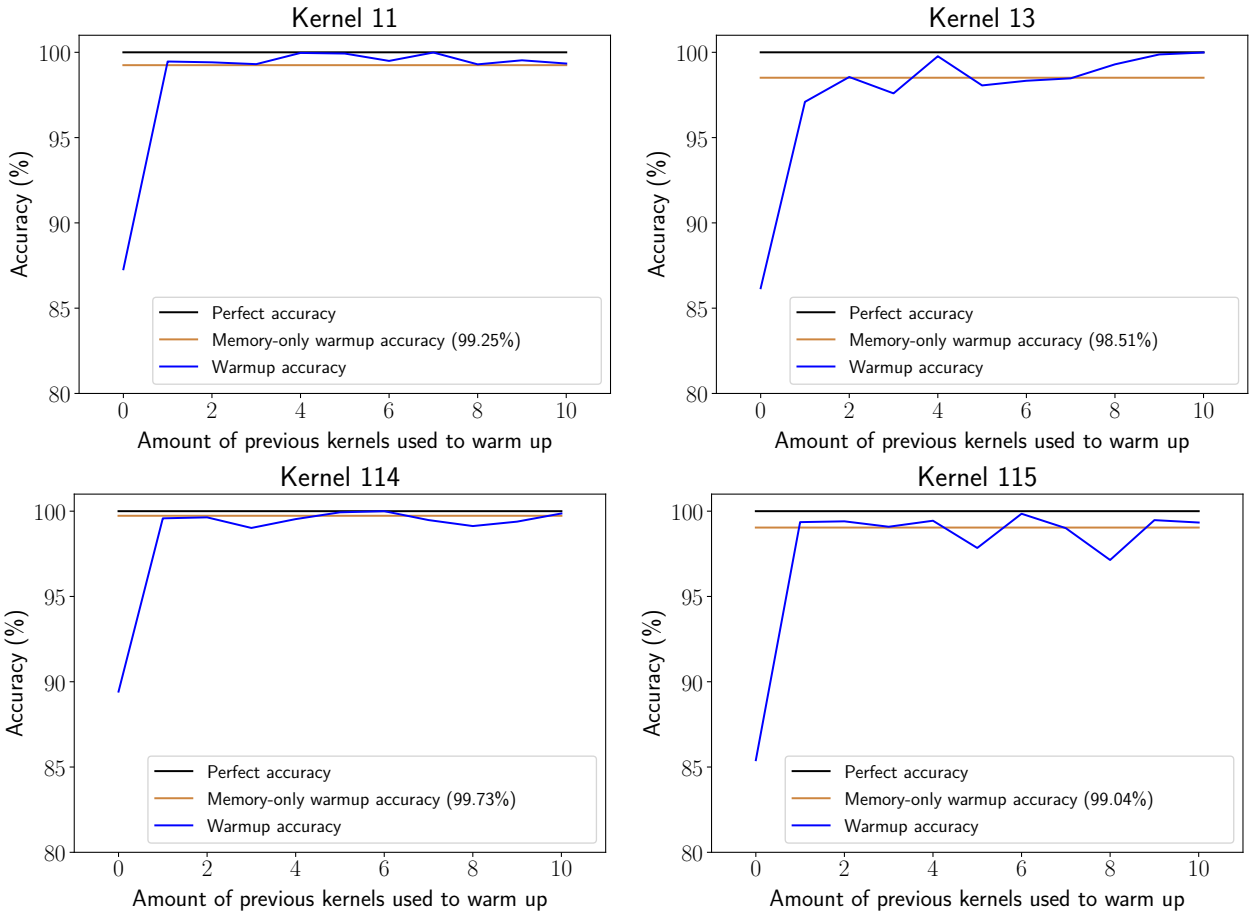


Figure 7.4: Accuracy of memory-only simulation

Our main conclusion from this analysis is that the memory-only simulation is very accurate. Even simulating only a single additional kernel results in incredible accuracy, see Figure 7.4.

7.2.2 Correction factor

Our final avenue was the correction factor. Using a selection of factors (like data reuse, forward data reuse, and backward data reuse), we have proposed a formula to correct the IPC values of kernels.

Our main goal was to compute an upper bound for the number of cycles lost due to the cold-start problem. We approximated this number to be proportional to:

- The number of (unique) DRAM requests made; this gives an idea of the number of cold misses. We scale this value by dividing it by the number of memory controllers used (and the cache line size).
- The forward data reuse factor of the previous kernel; this gives an idea of the fraction of cold misses due to the cold-start problem.

- The difference in latencies between LLC (L2) and DRAM.

With these observations, we came up with the following formula:

$$IPC = \frac{instructions}{cycles_f - \Delta} \tag{7.3}$$

$$\Delta = \frac{accesses}{controllers \cdot line} \cdot fwd_{i-1} \cdot (DRAM - L2) \tag{7.4}$$

Where:

- IPC is the final, corrected IPC;
- $instructions$ is the number of instructions executed by the kernel;
- $cycles_f$ is the number of cycles as reported by the simulator (in cold-start mode);
- Δ is the correction factor;
- $accesses$ is the number of unique DRAM accesses made by the kernel;
- $controllers$ is the number of memory controllers used[1];
- $line$ is the cache line size;
- fwd_{i-1} is the forward data reuse factor of the *previous* kernel; and
- $DRAM$ and $L2$ are the DRAM and L2 cache latencies, respectively.

We found that this formula does manage to eliminate the (lower) outliers in IPC values, at the cost of a slightly lower median accuracy.

7.2.3 Comparison

In the final section, Section 6.4, we have compared the two methods. We have found that the memory-only simulation is the most accurate, but we also noted it is the most computationally expensive. Below is once again the boxplot showing our conclusion, see Figure 7.5.

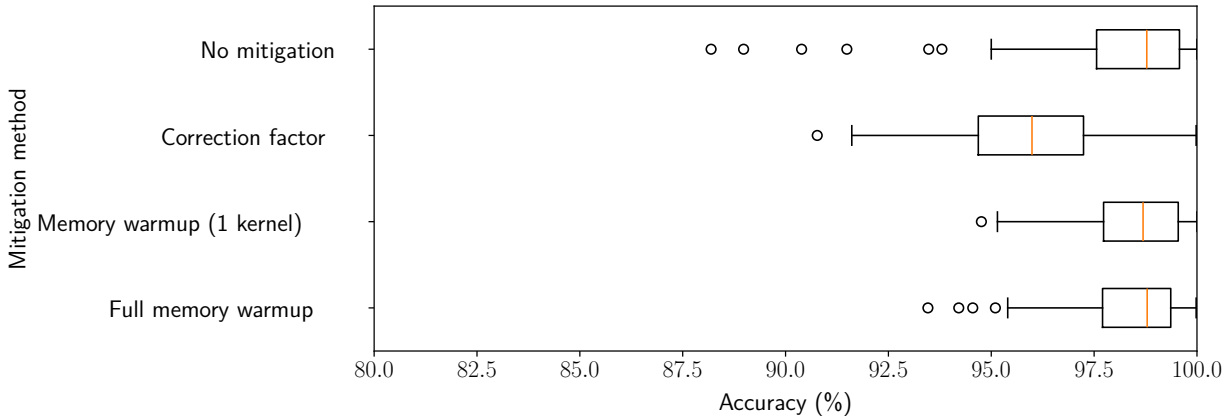


Figure 7.5: Comparison between mitigations

A strange result here is that full memory simulation is less accurate than simulating just a single kernel. Since the numbers are quite close together, we assume that this is largely due to non-determinism in the simulator.

[1] We count memory controllers that are used less than others, as partial memory controllers; see Section 6.3.1.

BIBLIOGRAPHY

[1] W. J. Dally, S. W. Keckler, and D. B. Kirk, "Evolution of the graphics processing unit (gpu)," *IEEE Micro*, vol. 41, p. 42–51, Nov. 2021.

[2] NVIDIA Corporation, "Cuda," 2022.

[3] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for?," *Queue*, vol. 6, p. 40–53, Mar. 2008.

[4] D. Zhang, S. Mishra, E. Brynjolfsson, J. Etchemendy, D. Ganguli, B. Grosz, T. Lyons, J. Manyika, J. C. Niebles, M. Sellitto, Y. Shoham, J. Clark, and R. Perrault, "The ai index 2021 annual report," 2021.

[5] C. Collange, M. Daumas, D. Defour, and D. Parello, "Barra: A parallel functional simulator for gpgpu," in *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, IEEE, Aug. 2010.

[6] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark, "Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems," in *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 353–364, 2010.

[7] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, " Multi2Sim: A Simulation Framework for CPU-GPU Computing ," in *Proc. of the 21st International Conference on Parallel Architectures and Compilation Techniques*, Sep. 2012.

[8] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, p. 1–7, May 2011.

[9] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, IEEE, Apr. 2009.

[10] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-sim: An extensible simulation framework for validated gpu modeling," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, May 2020.

[11] A. Jain, M. Khairy, and T. G. Rogers, "A quantitative evaluation of contemporary gpu simulation methodology," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 2, p. 1–28, June 2018.

[12] C. Avalos Baddouh, M. Khairy, R. N. Green, M. Payer, and T. G. Rogers, "Principal kernel analysis: A tractable methodology to simulate scaled gpu workloads," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, ACM, Oct. 2021.

[13] M. Naderan-Tahan, H. SeyyedAghaei, and L. Eeckhout, "Sieve: Stratified gpu-compute workload sampling," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, Apr. 2023.

[14] NVIDIA Corporation, "NVIDIA nsight compute."

[15] O. Villa, M. Stephenson, D. Nellans, and S. W. Keckler, "Nvbit: A dynamic binary instrumentation framework for nvidia gpus," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, ACM, Oct. 2019.

[16] V. Reḍḍī, C. M. B. Cheng, D. Kanter, P. H. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. A. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. Gardner, I. Hubara, S. S. Idgunji, T. B. Jablin, J. B. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. K. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. C. Wu, L. Xu, K. Yamada, B. Yu, G. Y. Yuan, A. Zhong, P. S. Zhang, and Y. Zhou, "Mlperf inference benchmark," *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 446–459, 2019.

[17] M. Naderan-Tahan and L. Eeckhout, "Cactus: Top-down gpu-compute benchmarking using real-life applications," in *2021 IEEE International Symposium on Workload Characterization (IISWC)*, IEEE, Nov. 2021.

[18] L. Eeckhout, Y. Luo, K. De Bosschere, and L. K. John, "Blrl: Accurate and efficient warmup for sampled processor simulation," *The Computer Journal*, vol. 48, no. 4, pp. 451–459, 2005.

[19] T. Austin, E. Larson, and D. Ernst, "Simplescalar: an infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 59–67, 2002.

[20] G. Lauterbach, "Accelerating architectural simulation by parallel execution of trace samples," in *1994 Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*, vol. 1, pp. 205–210, 1994.

[21] R. Kessler, M. Hill, and D. Wood, "A comparison of trace-sampling techniques for multi-megabyte caches," *IEEE Transactions on Computers*, vol. 43, no. 6, pp. 664–675, 1994.

[22] T. Conte, M. Hirsch, and K. Menezes, "Reducing state loss for effective trace sampling of superscalar processors," in *Proceedings International Conference on Computer Design. VLSI in Computers and Processors*, pp. 468–477, 1996.

[23] D. A. Wood, M. D. Hill, and R. E. Kessler, "A model for estimating trace-sample miss ratios," in *Proceedings of the 1991 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, SIGMETRICS91, ACM, Apr. 1991.

[24] T. Conte, M. Hirsch, and W.-M. Hwu, "Combining trace sampling with single pass methods for efficient cache simulation," *IEEE Transactions on Computers*, vol. 47, no. 6, pp. 714–720, 1998.

[25] J. Haskins and K. Skadron, "Minimal subset evaluation: rapid warm-up for simulated hardware state," in *Proceedings 2001 IEEE International Conference on Computer Design: VLSI in Computers and Processors. ICCD 2001*, pp. 32–39, 2001.

[26] J. Haskins and K. Skadron, "Memory reference reuse latency: Accelerated warmup for sampled microarchitecture simulation," in *2003 IEEE International Symposium on Performance Analysis of Systems and Software. ISPASS 2003.*, pp. 195–203, 2003.

[27] NVIDIA Corporation, "NVIDIA ampere ga102 gpu architecture," whitepaper, 2021.

[28] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," *CoRR*, vol. abs/1511.06434, 2015.

[29] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel, and J. D. Owens, "Gunrock: GPU graph analytics," *ACM Transactions on Parallel Computing*, vol. 4, pp. 3:1–3:49, Aug. 2017.

[30] S. Páll, A. Zhmurov, P. Bauer, M. Abraham, M. Lundborg, A. Gray, B. Hess, and E. Lindahl, "Heterogeneous parallelization and acceleration of molecular dynamics simulations in gromacs," *The Journal of Chemical Physics*, vol. 153, Oct. 2020.

[31] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in 't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott, and S. J. Plimpton, "LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales," *Comp. Phys. Comm.*, vol. 271, p. 108171, 2022.

[32] L. A. Gatys, A. S. Ecker, and M. Bethge, "A neural algorithm of artistic style," *ArXiv*, vol. abs/1508.06576, 2015.

[33] M. Jaderberg, K. Simonyan, A. Zisserman, and K. Kavukcuoglu, "Spatial transformer networks," *ArXiv*, vol. abs/1506.02025, 2015.

[34] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2015.

[35] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *North American Chapter of the Association for Computational Linguistics*, 2019.

[36] Ö. Çiçek, A. Abdulkadir, S. S. Lienkamp, T. Brox, and O. Ronneberger, "3d u-net: Learning dense volumetric segmentation from sparse annotation," in *International Conference on Medical Image Computing and Computer-Assisted Intervention*, 2016.