FACULTY
OF SCIENCES

QUANTIFYING AND MITIGATING THE COLD-START PROBLEM IN GPU SIMULATION

Jonas Sys

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*.

Abstract

TOELATING TOT HERGEBRUIK

CONTENTS

1 INTRODUCTION

Every year, more and more AI and machine learning papers are published[1]. Many machine learning algorithms rely on GPU computations to train their models, creating a growing need for more and faster GPUs. Improvements to GPUs can be done, among other means, by improving upon their internal micro-architecture. These changes, however, always need to be verified to ensure that they are, in fact, improvements.

When verifying, simulation usually provides results at a lower cost, albeit with lower accuracy. To this end, simulators like AccelSim[2] are used. However, it has also been shown that simulation has a huge overhead; applications that would take only about an hour on real hardware could easily take upwards of a century[3]. Some research has gone to speeding up this simulation. For example Principle Kernel Analysis[3], Sieve[4], and Photon[5] use sampling to select a few kernels, using those as representatives for the entire workload.

The speed-up gained from these techniques comes with a cost to accuracy, as intermediate kernels might have modified the cache state. This might lead to superfluous cache misses or incorrect cache hits, as loads and/or evictions might not have been performed. This problem is known as the cold-start problem, and has been researched with respect to the simulation of CPUs[6].

I aim to show that this problem persists when simulating GPUs, as well as show that it can be mitigated, albeit at the cost of a slightly higher simulation time.

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*[2], *Principal Kernel Analysis*[3], and *Sieve*[4]. 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 [2], 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[7]. 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 122 500 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[3] 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[4], 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. Firstly, it only requires 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

Secondly, Sieve uses strata-based clustering to reduce variability and select a better representative kernel. 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[8] and Cactus[9] 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:

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

3 THE COLD-START PROBLEM

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[10]. 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[11] and Gunrock[12] (on road traversal), from the Cactus[9] suite. Below is a full list of all analyzed workloads:

- Gromacs[13] and LAMMPS[14] (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)[15], reinforcement learning (RFL), spatial transformer (SPT)[16] and language translation (LGT) from PyTorch, and
- The following MLCommons benchmarks (from their MLPerf® Inference v2.0 collection):
  - The ResNet50 model[17],
  - Both MobileNet and ResNet34 variants of the SSD model,
  - The Bidirectional Encoder Representations from Transformers (BERT)[18], and
  - The 3D U-Net model[19]
- 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.

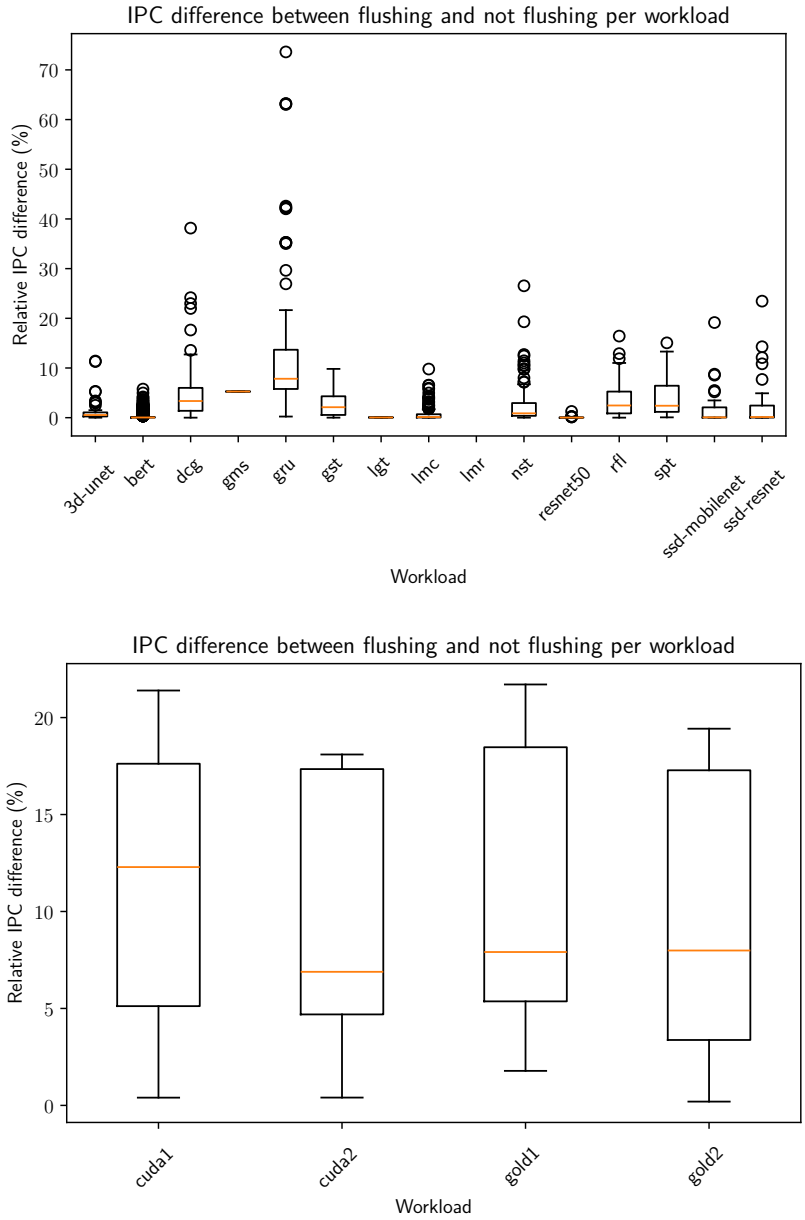


Figure 4.1: Relative IPC difference

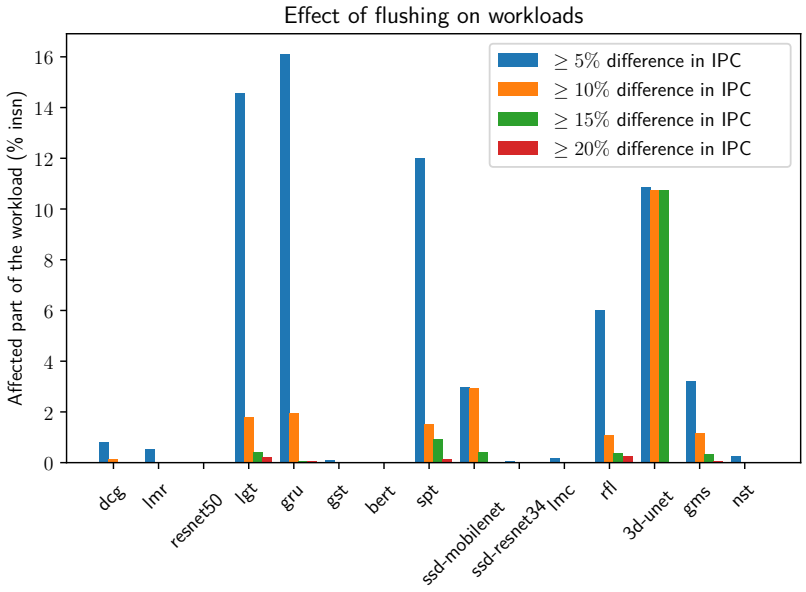
4.2 Weighting kernels

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[4] 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:

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

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

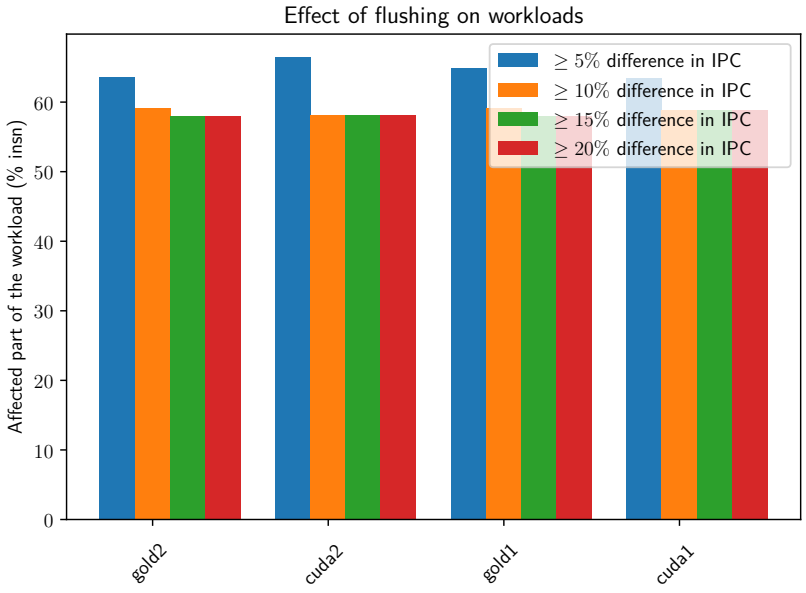
In order to get a better view of the impact of the cold start problem, we have also computed IPC difference when each kernel is weighted by its instruction count.



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

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



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

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.

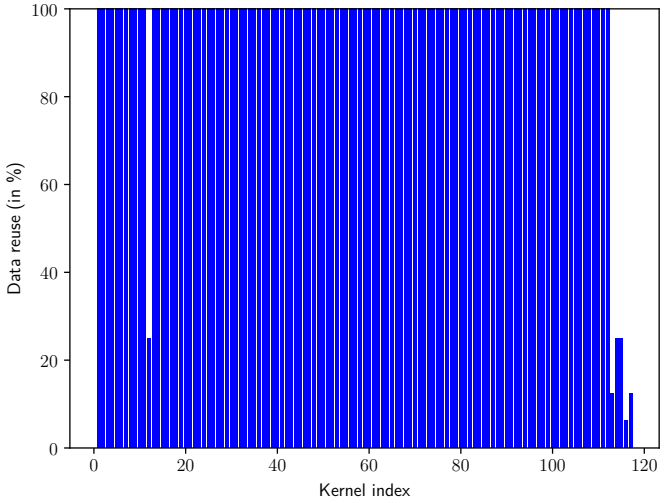
In figure 4.4, you can see the data reuse ratio for the DCT workload. We have analyzed both the forward and backward data reuse for each kernel:

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

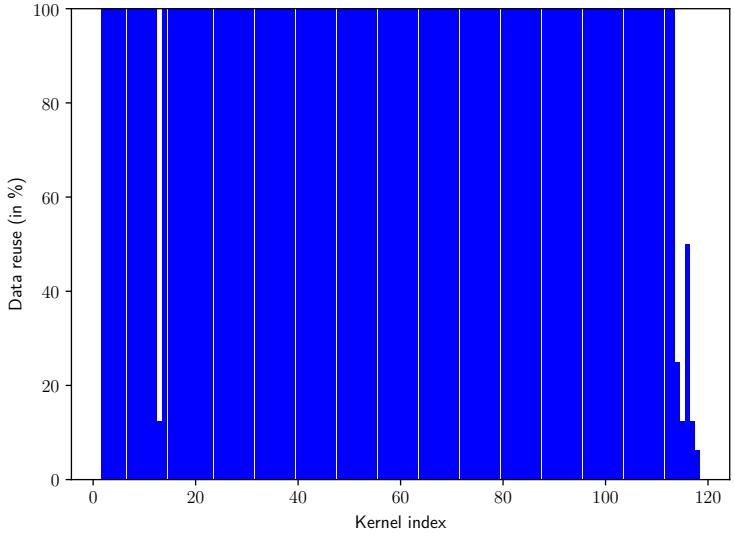
$$\frac{|\text{footprint } k_{i-1} \cap \text{footprint } k_i|}{|\text{footprint } k_{i-1}|} \tag{4.1}$$

- **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.

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



(a) Forward data reuse



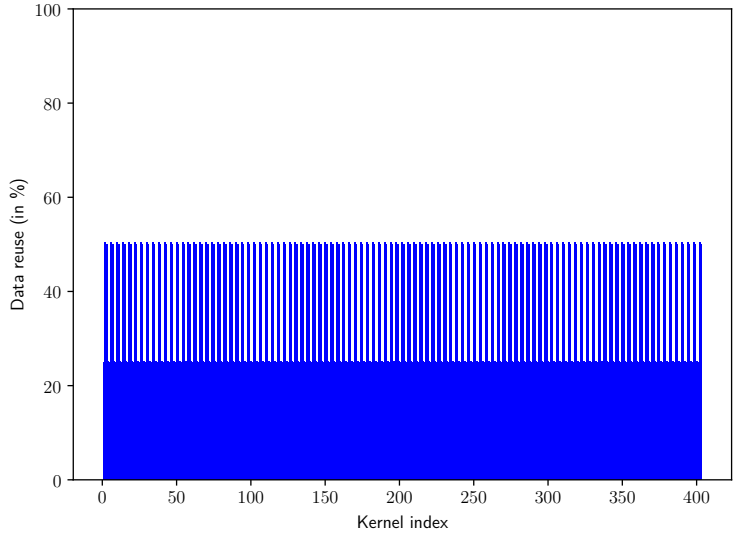
(b) Backward data reuse

Figure 4.4: Inter-kernel data reuse in DCT

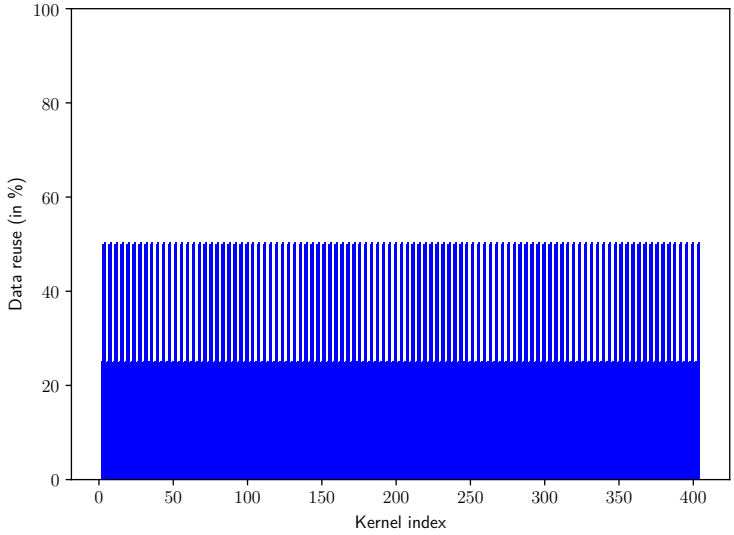
When looking at the DCT workload, most kernels have a very high degree of data reuse, both backward and forward. As we analyzed before, this workload suffers heavily from the cold-start problem (in hardware).

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 4.5: 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've used in the previous chapter.

5.1 Simulation setup

The simulator we will be using is AccelSim[2]. As discussed in section 2.1, AccelSim uses the GPGPU-Sim[7] 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 [10]). 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[20] 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

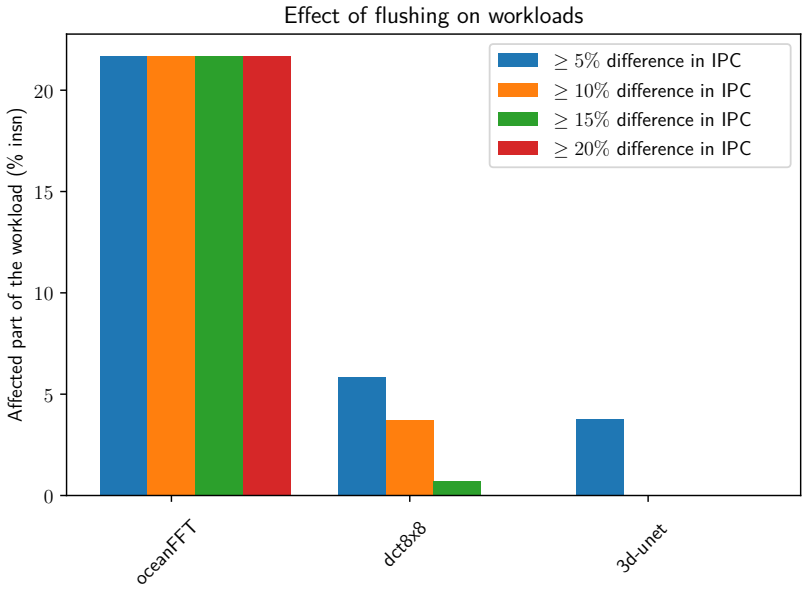
In this section, we will show that the impact of this problem persists in the AccelSim simulator. However, one of the first things we have noticed is that the simulator is affected much differently from the real hardware.

Due to the inherent overhead in simulating GPUs, we had to limit the number of kernels we could simulate, 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).

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,



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

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.

Kernel Name	IPC Difference	Weight	Forward Data Reuse (%)	Backward Data Reuse (%)
`generateSpectrumKernel`	0.28%	44.43%	12.50%	n/a
`regular_fft`	*31.40%*	*21.69%*	*25.00%*	*25.00%*
`vector_fft`	0.45%	17.81%	0.00%	25.00%
`updateHeightmapKernel`	0.01%	6.00%	50.00%	0.00 %
`calculateSlopeKernel`	0.00%	10.08%	n/a	18.18%

Figure 5.3: OceanFFT kernels

5.4 Simulator conclusion

Once again, we see that the cold-start problem persists. However, it is much less severe in the simulator.

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.

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[2] tool comes with an NVBIT[20] 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.

As with the original traces, we need to post-process each memory trace before feeding it to the simulator. By using the same format as the original tool, we ensure that the existing post-processor can also handle these new files.

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

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:

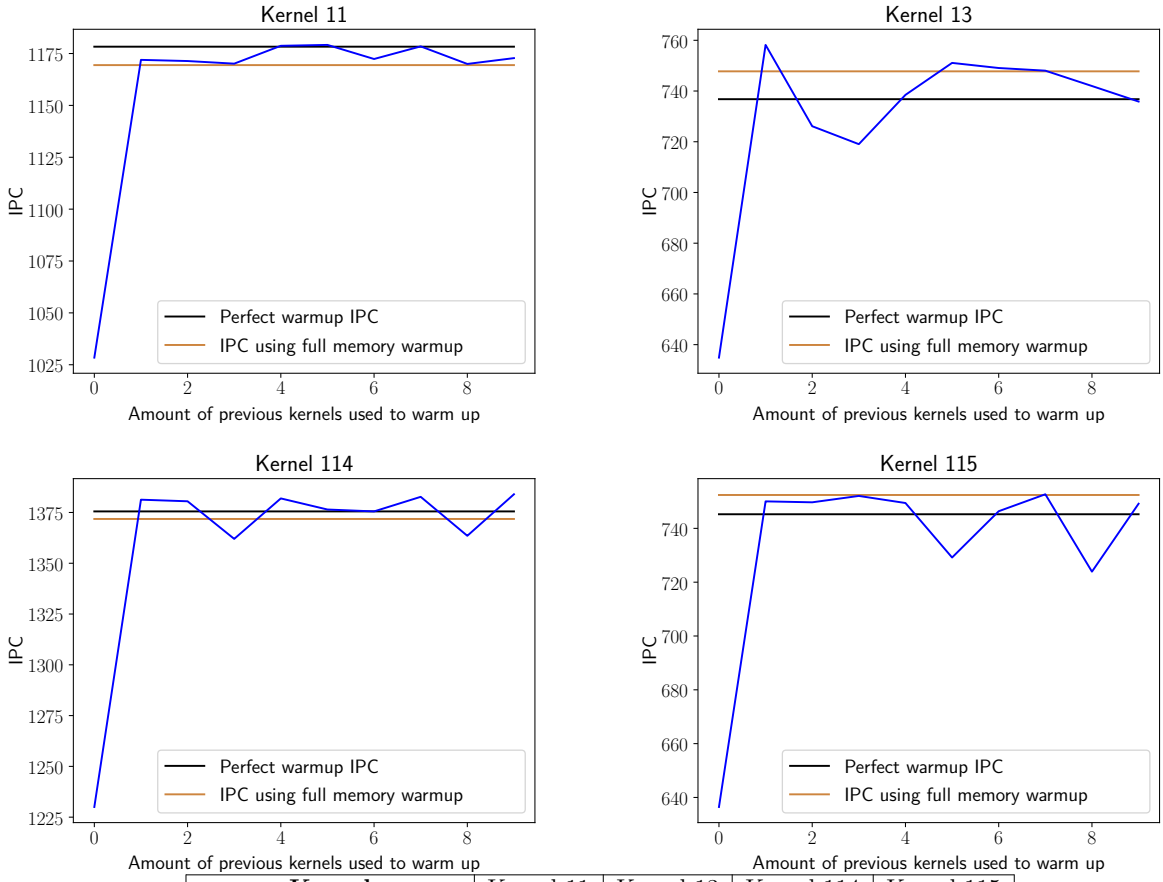
- **Perfect warmup:** the kernel is simulated in order, with all preceding kernels simulated in full.
- **Full memory warmup:** the kernel is simulated in order, with all memory instructions preceding it simulated (for kernel i, this means all memory instructions from kernels 1 until $i-1$ are simulated).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

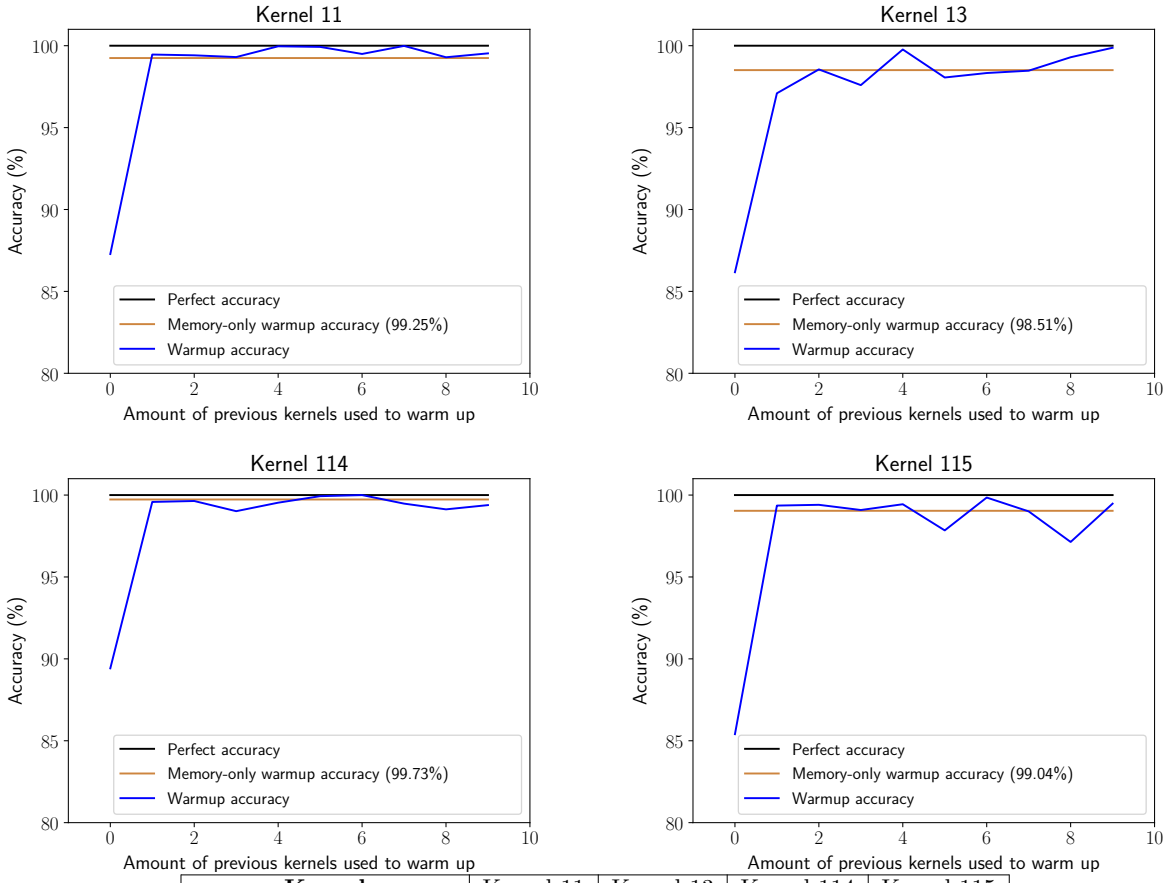
An additional striking detail is that more kernels does not always imply a higher accuracy. This might be due to some non-determinism in the simulator.

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
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%
Full memory warmup	99.25%	98.51%	99.73%	99.04%

Figure 6.3: DCT Mitigation results (accuracy)

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[21] and NVBit[20] can provide detailed cache statistics, which might be used to compute this factor. Some possible factors 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*.

index	memory fraction	memory footprint	accuracy	L2 miss rate	forward reuse	backward reuse
1	20.25%	32814	99.19%	100.00%	0.00%	100.00%
2	20.25%	32814	99.62%	75.00%	100.00%	100.00%
3	20.25%	32814	99.78%	66.67%	100.00%	100.00%
10	20.25%	32814	99.63%	55.00%	100.00%	100.00%
11	*8.04%*	*32768*	*88.17%*	*50.05%*	*100.00%*	*100.00%*
12	21.92%	32814	99.80%	50.04%	100.00%	12.50%
13	*33.16%*	*65632*	*89.15%*	*53.38%*	*25.00%*	*100.00%*
14	33.16%	65632	98.54%	50.07%	100.00%	100.00%
15	33.16%	65632	94.06%	47.43%	100.00%	100.00%
113	33.16%	65632	99.45%	26.98%	100.00%	25.00%
114	*6.45%*	*32768*	*90.62%*	*26.96%*	*12.50%*	*12.50%*
115	*33.16%*	*65632*	*86.92%*	*26.94%*	*25.00%*	*50.00%*
116	21.08%	32944	95.28%	26.94%	25.00%	12.50%
117	5.93%	16384	97.66%	26.81%	6.25%	6.25%
118	20.35%	32944	92.77%	26.81%	12.50%	0.00%

Figure 6.4: DCT mitigation statistics

index	memory fraction	memory footprint	accuracy	L2 miss rate	forward reuse	backward reuse
1	2.45%	33564671	99.72%	85.87%	0.00%	25.00%
2	*16.03%*	*16779666*	*54.23%*	*34.78%*	*12.50%*	*25.00%*
3	15.45%	16780062	99.55%	44.33%	25.00%	0.00%
4	13.64%	16777216	99.99%	49.88%	0.00%	18.18%
5	16.21%	46137344	100.00%	51.06%	50.00%	0.00%

Figure 6.5: OceanFFT mitigation statistics

We have gathered some of these factors in figure 6.4 for a subset of the DCT workload. The kernel invocations for which a mitigation is required (kernels 11, 13, 114, and 115) are emphasized. In figure 6.5, we have done the same for the OceanFFT workload (with its single high-impact kernel marked).

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.

6.4 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.

Finally, we have looked into possible correction factors. Some possible factors that can be used to create a comprehensive function to compute a correction factor have been outlined. These factors range from L2 miss rate and DRAM delay to higher-level statistics about inter-kernel data reusability. However, no conclusive results were found. We also noted that it might be possible to over-tune a solution like this. Computing a highly accurate correction factor like this might work perfectly on one platform but reduce accuracy on another, missing the point of GPU simulation entirely.

7 CONCLUSION

BIBLIOGRAPHY

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] C. Liu, Y. Sun, and T. E. Carlson, "Photon: A fine-grained sampled simulation methodology for gpu workloads," in *56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, ACM, Oct. 2023.

[6] N. Nikoleris, A. Sandberg, E. Hagersten, and T. E. Carlson, "Coolsim: Statistical techniques to replace cache warming with efficient, virtualized profiling," in *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, IEEE, July 2016.

[7] 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.

[8] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou, "Mlperf inference benchmark," 2019.

[9] 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.

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

[11] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," 2015.

[12] 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.

[13] 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.

[14] 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.

[15] L. A. Gatys, A. S. Ecker, and M. Bethge, "A neural algorithm of artistic style," 2015.

[16] M. Jaderberg, K. Simonyan, A. Zisserman, and K. Kavukcuoglu, "Spatial transformer networks," 2015.

[17] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.

[18] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2018.

[19] O. Çediçek, A. Abdulkadir, S. S. Lienkamp, T. Brox, and O. Ronneberger, "3d u-net: Learning dense volumetric segmentation from sparse annotation," 2016.

[20] 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.

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