DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Capturing the Memory Topology of GPUs

Dominik Größler

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Capturing the Memory Topology of GPUs

Erfassung der Speichertopologie von GPUs

Author:	Dominik Größler
Supervisor:	Prof. Dr. Martin Schulz
Advisor:	Stepan Vanecek
Submission Date:	17.10.2022

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 17.10.2022 Dominik Größler

Acknowledgments

I would like to thank Professor Schulz for the outstanding opportunity to write my bachelor's thesis about such an interesting topic as well as my advisor Stepan Vanecek for his ongoing support during the last months and his valuable advice. The weekly meetings helped me a lot to drive progress and to stay focused.

Further, I wish to thank my family and close friends.

Abstract

Optimizing program code is an essential process for High-Performance Computing and in general. Due to a trend in the last years of employing graphics cards as accelerators for systems and due to a universal gain of the importance of GPUs, optimizing GPU code is crucial in order to achieve the best possible performance of a GPU program. The optimization process works best with fundamental background knowledge about the GPU(s) and their structure on which the program will be executed. However, a lot of the relevant information especially regarding the overall memory structures is not or only vaguely documented. For this purpose, this thesis presents the design and implementation of a tool to capture the memory topology of GPUs. This work only considers the GPUs of Nvidia whereas AMD or other manufacturers are not covered.

Zusammenfassung

Das Optimieren von Programmcode ist ein wichtiger Prozess für den Bereich des High-Performance Computing, aber auch grundsätzlich von großer Bedeutung. Gemäß eines Trends der letzten Jahre, in denen Grafikkarten häufig als Beschleuniger für Programme und Systeme eingesetzt werden, und aufgrund eines generellen Bedeutungsgewinns von GPUs ist die Optimierung von GPU Programmcode von höchstem Wert, um die bestmögliche Leistung erreichen zu können. Grundsätzlich funktioniert der Optimierungsprozess am besten, wenn umfangreiches Hintergrundwissen über die GPUs, auf denen das Programm laufen wird, bekannt ist. Jedoch sind viele der nötigen Informationen, besonders bezüglich der Speicherstrukturen und -hierarchien, überhaupt nicht oder nur unzureichend dokumentiert. Diese Arbeit soll zu diesem Zweck das Design und die Implementierung eines Hilfsprogrammes darlegen, mit dem die Speichertopologie von GPUs erfasst werden kann. Dabei beschränkt sich der Umfang der Arbeit ausschließlich auf GPUs von Nvidia, während Grafikkarten anderer Hersteller wie AMD von der Arbeit nicht behandelt werden.

Contents

1. Introduction

In the past decade, the usage of Graphics Processing Units (GPUs) as coprocessors has evolved into a crucially relevant aspect in the domain of High Performance Computing (HPC). In the TOP500 list of the most powerful HPC-machines, seven of the top ten machines utilize GPUs as accelerators for their systems [1]. 34 out of the 50 most well-known HPC-application packages support the usage of GPUs including all top 15 applications [2]. The underlying cause for this trend is the potential for vast and beneficial performance improvements of HPC-applications by the use of GPUs (e.g. by factors of 20x or even more [3][4]) which is possible because of the highly parallelized design of a GPU as well as, consequently, a considerably more accelerated increase of the computation capabilities (GFLOP/sec) and the memory bandwidth (GB/sec) compared to conventional CPUs [5, p.19-20].

However, the GPU computation performance increases significantly faster than the GPU memory system bandwidth. This makes memory access a serious potential bottleneck for the overall GPU performance [6]. One of the most common ways to at least lower this often called "*memory wall*"-bottleneck is typically to employ a sophisticated cache system that aims to keep the number of cost-intensive data transfers as low as possible by adding caches closer to the computation cores in order to have mostly inexpensive data transfers [7]. Thus, in order to fully exploit the capabilities of GPUs plus the cache system and to soften this memory bottleneck at best, developers should be informed about the concrete hardware characteristics - especially the various memory and cache components - in the most detailed way possible. Unfortunately, the manufacturers of GPUs prefer to either not share all of the relevant information or to share only rather unintuitive details about the inner workings of the corresponding GPUs in huge technical documentation sheets instead of a comprehensible overview of the memory topology. Some performance and analysis tools include a small fragment of the relevant information about the memory characteristics of GPUs in its output like LIKWID [8] although a huge part of the overall topology still remains uncovered. These tools mainly contain surface information about the GPU in regards of the memory topology like the main memory size or the L2 Cache size. However, the core information about the low-level caches like the L1 Data Cache, Texture Cache, Read-Only Cache and Constant Cache is still missing.

Therefore, the goal of this thesis is to design an approach for automatically gathering concrete information about every memory and cache component in the GPU, especially the size and latency. The approach employs a specific microbenchmarking technique inspired by previous works in the field of microbenchmarking in order to generate data about the cache-loading behavior of GPUs [9] [10] [11]. Additionally, an extraction of the relevant information from the data generated by the microbenchmarks is necessary to receive concrete values for the cache sizes and other characteristics. This extraction is applied via an algorithm coming from the field of mathematical change point detection which takes a series of values and analyzes it to obtain the point with the most significant change in the series. The focus of this work mainly relies on GPU-devices manufactured by the Nvidia Corporation by primarily using CUDA code for the implementation of the proposed approach. The tool utilizes a combination of various CUDA functions together with the microbenchmarking technique followed by change point detection to create an extended overview of the complete memory topology.

The second chapter mostly explains theoretical background concerning memory caches and GPU-related background, then it further depicts in the third chapter the general microbenchmarking technique as well as the change point detection algorithm and other topics relevant to the overall proposed design. The fourth chapter gives a detailed insight into the actual implementation of the tool. In the fifth chapter, the results for all tested GPUs are presented and the remarkable details of the results are highlighted. The sixth chapter contains an outlook on how the tool can be further extended by bandwidth data, instruction cache information or other matters. Previous research that is partially related to this work or the goal of this work - to gain more insight into the GPU-system in general - is illustrated in chapter seven. The final chapter concludes the thesis and reviews the overall outcome of the executed work.

2. Background

The tool combines techniques of microbenchmarking with change point detection on multiple caches of the GPU. Consequently, this chapter outlines briefly the theoretical background of cache functionality followed by a clear explanation of the hard- and software perspective on Nvidia GPUs.

2.1. Cache

2.1.1. General

A significant part of the memory system of a GPU consists of multiple caches, hence this section describes the basic functionality of a cache and some of its essential characteristics. Fundamentally, a cache functions as an additional buffer between a processing unit and a main memory component. The most important feature of a cache is the considerably faster loading time of data that is stored in the cache compared to data that is only stored in the main memory. Whenever the processing unit calls a load instruction, first the cache is asked to return the requested data. If the data is cached, the data is loaded from the cache (cache hit). If the data is not cached the request is forwarded further to the main memory (or higher-level caches). Then the data will be loaded from the main memory and stored in the cache for future demands (cache miss). One major drawback - but at the same time requirement for caches - is the significantly smaller size in comparison to the main memory. This is necessary because the lower loading time of data originates from the reduced size of the cache, which makes it much easier and faster for the cache controller to find data in the cache and to send it back to the processing unit. In order to increase the number of cache hits and thus, the efficiency, most caches use *principles of locality*, especially two types. *Temporal locality* operates on the observation or assumption that previously required data is often reused at some point of the program and is therefore kept in the cache as long as possible. *Spatial Locality* assumes the prospective usage of data that is close to already referenced data and tries to cache it before the first actual usage [12] [13] [14] [15] [16].

2.1.2. Cache Associativity

The structure of a cache highly depends on its desired associativity but in general a cache with size C is structured into m cache sets, which contain n cache lines. The higher the associativity of a cache gets, the more complex will be the hardware to support it. Maximum associativity of a cache or a *fully associative* cache means, every location of the main memory can be mapped to and the corresponding data can be cached into every cache line of the cache. Consequently, a fully associative cache consists of one cache set with lots of cache lines. A slightly more restricted version is the *k-way set-associative* cache. Here each cache set contains exactly k cache lines. A location of the main memory can only be mapped to one specific cache set. The most restricted but concerning hardware simplest version is the *direct mapped* cache with only one cache line per cache set. Thus, each location of the main memory directly maps to one specific line of the cache [12] [13] [14] [16].

2.1.3. Cache Replacement Policy

Assuming a cache is completely empty at the beginning of a program (or during the startup of the system), once a program starts to request data from the memory, the request first tries to load the required values from the cache. Subsequently, due to an unoccupied cache cache misses are produced, also known as *cold cache misses* or *compulsory cache misses*. Those cannot be prevented. Eventually, as soon as the cache and its cache sets are filled after lots of memory requests the cache starts to have cache misses as a result of an "overflowed" cache (or cache set). These cache misses are called *conflict cache misses*, if a cache miss would have not occurred with a different cache structure (e.g larger cache sets) or *capacity cache misses*, if a cache miss would have occurred despite the internal cache structure (e.g a program loads 600 KiB data and the cache size is 300 KiB). Regardless of the exact type of miss, each cache uses a definitive *cache replacement policy*, which regulates how these cache misses are resolved by removing data from the cache and moving the requested data into the cache. The overall goal of the cache replacement policy is ideally to remove data that is not going to be used anymore in order to keep the number of cache hits as high and the number of cache misses as low as possible and thus, guaranteeing the best efficiency for the cache. Popular replacement strategies among many others are:

- *Round-Robin*: Uses an index pointer to select the cache line to be removed. After the cache replacement, the pointer is set to point to the next cache line or it is reset to the first cache line if the pointer reached the end of the cache set.
- *Least Frequently Used LFU*: Replaces the cache line that was referenced the fewest

number of times.

- *Least Recently Used LRU*: Replaces the cache line that was not referenced for the longest time compared to all other lines in the cache set.

It is worth mentioning that in real examples the applied replacement strategy is often a variation or modification of the above policies (Time aware LRU, LFU-aging, etc.) with the same principle. Since the area of cache replacement policies is highly researched, more information about it can be found at [13] [14] [16].

2.2. Graphics Processing Unit

Principally, the structure and workflow of modern GPUs are very similar regardless of the manufacturer. Nonetheless, this thesis uses terminology that is used for Nvidia GPU-devices, since its focus is exclusively on GPUs of Nvidia. The corresponding terms for the other manufacturers can be found in **2.2.1 Terminology (other manufacturers)**.

2.2.1. Terminology (other manufacturers)

Table 2.1.: Terminology - Computation Resources

Nvidia	**Streaming Multiprocessor**	**CUDA Core**
AMD	Compute Unit	Stream Processors
Intel	Execution Unit or X^e Vector Engine	Hardware Threads

2.2.2. Structure

The structure of a GPU is fundamentally different from a CPU. In order to fulfill its needs of heavily parallelized computations, a GPU consists of considerably more computation cores, which are able to run instructions in parallel. The basic architecture is to be described by reference to the following simplified example of a Nvidia GPU microarchitecture (See Figure **2.1 Exemplary GPU architecture**). The typical Nvidia GPU consists of a huge number of Streaming Multiprocessors (SMs) that are able to work independently from each other. The SM can be considered a single SIMT-unit (Single Instruction Multiple Threads). So it can execute one instruction in parallel on lots of threads [17]. The SMs load data from the main memory of the GPU, which is also called Video RAM or VRAM, and are able to make use of an intermediate second

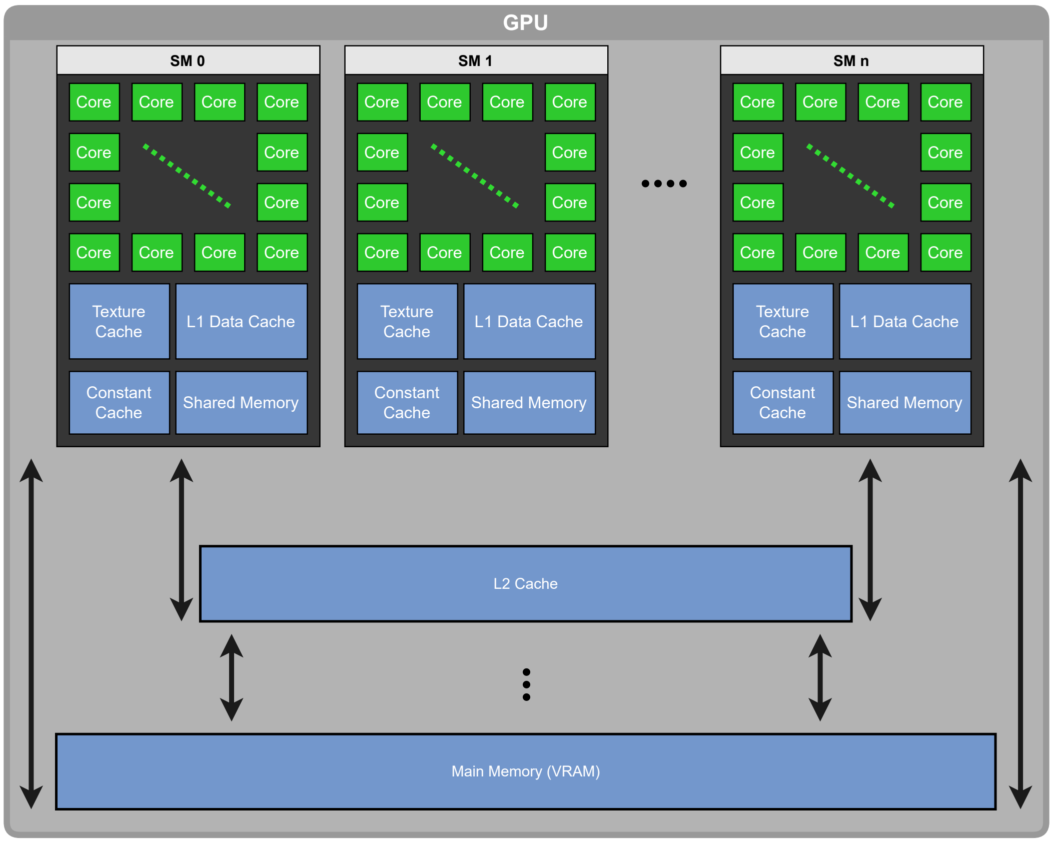

Figure 2.1.: Exemplary GPU architecture

level L2 Cache. The L2 Cache is usually the largest cache in a GPU. The SM (See Figure **2.2 Exemplary SM-internal architecture (Kepler)**) comprises a huge amount of so-called CUDA Cores, which normally contain an *Integer Arithmetic Logical Unit* and a *Floating Point Unit* [18] [19] [20, p.6-9]. These cores have access to additional caches and shared memory inside the SM. In terms of memory each SM contains:

- Register File
- L1 Data Cache
- Texture Cache
- Read-Only Cache
- Constant Cache
- Instruction Cache
- Shared Memory

The *Register File* is a collection of lots of 32-bit registers. It contains general purpose registers that can be arbitrarily used by the cores, special purpose registers that hold individual values such as the thread id (`threadIdx.x`) of a thread or the current clock cycles (`clock`-register) and predicate registers that can control if specific instructions will be executed or not [22].

The *L1 Data Cache* is an additional data load and store layer between the cores and the L2 Cache. Note that the physical L1 Data Cache is often the same cache that is used for texture caching and read-only data caching. In addition, some GPUs do not support the caching of global loads into the L1 Data Cache (e.g. some Kepler-GPUs [23, Ch.1.4.4.2]).

The *Texture Cache* is the cache that is operated on while using the two available texture interfaces (Texture Reference and Texture Object) for loading data. The texture cache is read-only during a kernel call. This indicates the cache is not kept coherent during the execution of a kernel. If the index of a texture is modified and afterwards loaded within the same kernel call, the behavior and the loaded data are undefined [24, Ch. 2.3] [25] [26].

The *Read-Only (Data) Cache* is a cache, that is kept in read-only mode during a kernel call just the same way as the texture cache is operated. Therefore, the Read-Only Cache

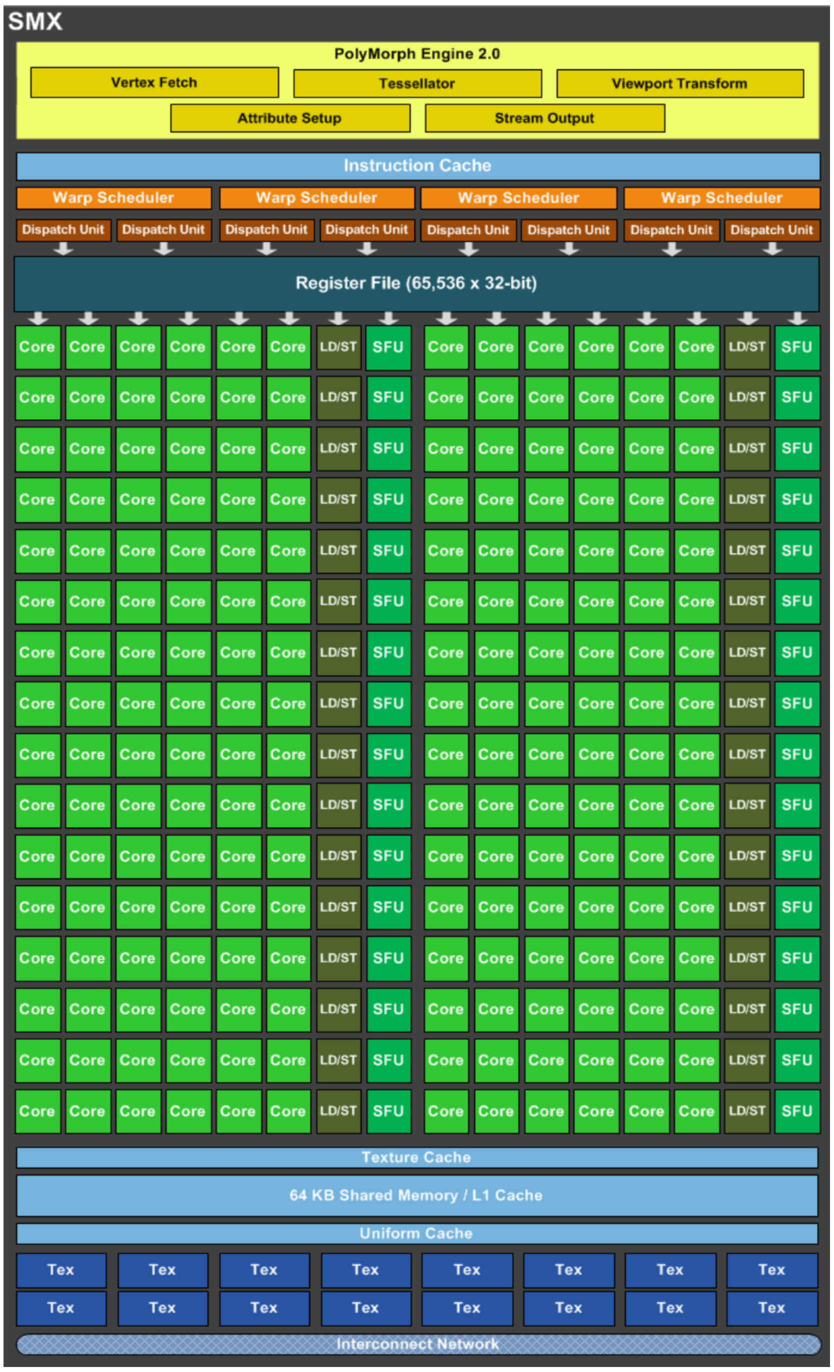

Figure 2.2.: Exemplary SM internal architecture (Kepler) [21]

is mostly the same physical cache as the Texture Cache [23, Ch. 1.4.4.3].

The *Constant Cache* is a cache for values that are constant during the kernel execution. The difference between constant cache and read-only cache is, that it is possible to write back to the read-only cache even if the changes are propagated only after the current kernel has finished. It is not possible to write back to the constant cache during the kernel execution. One important difference between the Constant Cache and the other caches is, how cache accesses are handled. If threads access different addresses of the Constant Cache these accesses are serialized. As a result, the more often unique addresses are accessed by the threads the slower is the overall loading process. Thus, the Constant Cache is most efficient when all threads access the same address at the same time [27, Ch. 9.2.6].

The *Instruction Cache* stores instructions that will be executed in the future. Hence, it ensures a speed-up for the instruction fetch of the execution cycle (Instruction Fetch -> Instruction Decode -> Execute -> Write Back). This cache is barely covered by this work, only theoretically in **6.2 Instruction Cache**.

The *Shared Memory* is a small memory piece that can be accessed by all cores within a SM [24, Ch. 3.3]. With this memory, the cores can collaborate with each other and cooperatively calculate results in a more optimized and faster way.

Overall it is noteworthy, that some microarchitectures or their corresponding SM-internal architectures contain caches that are split up into two or more parts. As an example, the Maxwell microarchitecture has officially a 24 KiB cache that is used as L1 Data, Texture and Read-Only Cache. This cache is split up into two 12 KiB caches physically (See Figure **4.18 SM-internal architecture for Maxwell**).

2.2.3. CUDA Model

While the last chapter covered the GPU mainly from the hardware perspective, this chapter describes the software perspective on the GPU. Nvidia GPUs can be used by developers for general purposes via the CUDA-interface (Compute Unit Device Architecture) or the corresponding CUDA SDK, respectively. CUDA code is very close to C/C++ code, with only a few additional modifiers. In order to utilize the GPU as a developer, a function must be declared as a *kernel* function, which can be done via the declaration specifier `__global__` in front of the function name as in:

```
__global__ void kernel_func (int* data);
```

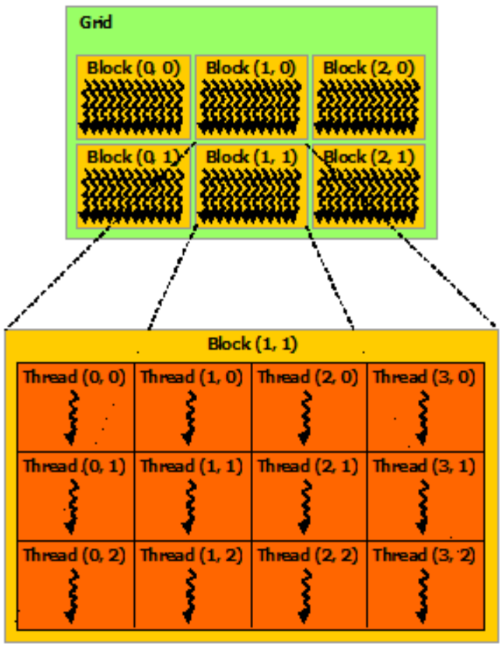

Figure 2.3.: The CUDA grid concept [28, Ch.2.2]

The kernel function is then executed by calling it with a grid specifier that defines how many threads and thread blocks will run on the GPU. Conceptually, the grid (See Figure **2.3 The CUDA grid concept**) consists of a fixed number of blocks, which are mapped to a specific SM after calling the kernel function. Each block is created with an equal number of threads inside that are then executed on the cores of the SM [28, Ch.2]. The CUDA grid specifier is a two-dimensional value that is specified for a kernel call with `<<<numBlocks, numThreads>>>`. The first value defines the number of blocks of the grid, whereas the second value determines the number of threads in each block. Both values can also be two- or three-dimensional. As an example, the following two lines execute $(10 * 5)$ blocks and overall $(10 * 5) * 8$ threads:

```
dim3 blocks(10, 5, 1);
kernel_func<<<blocks, 8>>>();
```

On the GPU a group of 32 threads (*warp*) is put together and then executed in parallel. More warps are created, if the number of threads in the thread block is larger than 32. The execution of a warp is managed by a warp scheduler. Each warp is assigned statically to one warp scheduler [28, Ch.K]. When the warp executes an instruction, all threads execute the instruction unless the instruction is inside of a conditional branch. Then the warp disables threads that are not in the current branch and only executes the threads that take the execution path through the branch (this is called *Branch divergence* [28, Ch. 4.1]).

The CUDA software perspective fundamentally distinguishes between different types of possible memory. There is [28, Ch. 5.3.2] [29]:

- Local Memory
- Global Memory
- Texture Memory
- Constant Memory
- Shared Memory

The *Local Memory* is only accessed for some variables that the compiler automatically decided to have locally; for example, if the kernel function tries to use more registers than available (*Register Spilling*).

Global Memory can be allocated via `cudaMalloc()` and is a read- and write-memory. It corresponds to the device DRAM and can be accessed and modified from the device (GPU) during a kernel execution and from the host (CPU) before the kernel is launched or after the kernel has finished.

The *Texture Memory* is read-only memory during a kernel call. Texture data can be loaded from texture memory via texture fetches. A texture fetch call in CUDA looks as follows: `tex1Dfetch<type>(texture, index);`

The *Constant Memory* is stored in the main memory and constant data can be cached in the Constant Cache. The values of the constant memory are read-only during a kernel call. The difference between constant memory and read-only memory is, that it is indeed possible to write back to read-only memory but the change is propagated only after the current kernel has finished.

The *Shared Memory* corresponds to the physical on-chip SM-internal shared memory.

While the local memory is mainly used by the compiler, all other memories will be used by the tool to determine the size of the corresponding caches or some of their other properties.

2.2.4. PTX

Moreover, CUDA provides a lower-level virtual-GPU instruction set architecture, called *parallel-thread-execution* (PTX-ISA), which can be used to execute low-level assembler-like code instructions. It is, however, not the real assembler code that is executed on the cores but an intermediate layer between the CUDA C code and the real low-level assembler code, called *SASS* [30] that is further compiled into a CUDA binary (*cubin*) [31]. While it is currently not possible to write and compile SASS-code manually, it is possible to do that with PTX-code. This can be done either in a separate .ptx-file or via inlining PTX code within CUDA C/C++ code. Inlining PTX code is realized with the `asm()`-statement, which takes the PTX code in a special format as input. The format of the `asm()`-statement looks like this [32]:

```
asm("template-string":"constraint"(output):"constraint"(input));
```

The first string contains the sequence of instructions that will be inlined, the second part defines all output variables and the last part specifies the input variables. It is also possible to add the string "*memory*" as an additional fourth option to declare that the sequence of instructions contains (desired) side effects to memory. This tells the compiler, that no memory optimizations should be applied for these instructions during the PTX code generation of the whole CUDA code. To stop reordering or optimizations on the inline PTX code, developers can further use the keyword *"volatile"*. Here is an example of the usage of inline PTX assembly:

```
asm volatile("st.u32 [%0], %1;" :  "r"(p), "r"(x) ::  "memory");
```

The "%0" placeholder references the first variable or register, respectively, that is stated after the first colon as output, namely the register that is used for the variable p. The "%1" placeholder is a reference to the input variable x. The keyword *"volatile"* is used to prevent the compiler from optimizing parts of the inline PTX assembly away. The following PTX instructions will be relevant for the scope of the work:

- `mov.u32`
- `ld.global.u32`
- `st.shared.u32`
- `add.u32`

- `cvta.to.shared.u64`

It can be observed, that the structure of the instructions consists of the type of instruction (`mov`, `load`, `store`, etc.) and the type of operands (`u32`, `u64`). Some instructions allow further options like specifying the type of memory, that is used for the load and store instructions.

3. Design

This chapter mainly covers the proposed design of the approach for obtaining the cache sizes. First, a specific benchmark technique is applied for each cache, which generates lots of loading time data. This data can be used by the change point detection to deduce/extract the size of a cache. Moreover, this benchmark technique can be modified to get additional information like the cache line size, the latency, if specific caches use the same physical cache or how many specific caches are inside of an SM.

3.1. Benchmark Technique

One of the most important essentials of the approach is the application of a specific microbenchmarking technique that has already been used quite often in the past to reveal memory characteristics of a system, the so-called *Pointer-Chase* (P-Chase) microbenchmark method [9] [10] [11] [33] [34] [35] [36]. In particular, the approach applies the more advanced *Fine-Grained Pointer-Chase* (Fine-Grained P-Chase) method that was proposed and demonstrated for GPUs in [9] and [10].

3.1.1. General P-Chase Method

Fundamentally, the goal of the P-Chase method is to execute an arbitrarily long series of reads from a specific cache with each read being dependent on the preceding read, while measuring the elapsed time. It provides such a way by creating an array whose values are always used as indices of the next read. As an example, in order to sequentially pass the array, it needs to be filled with incrementing values starting with the value "1" at the first index (See Array at **3.1 P-Chase Array**). Before the pointer chase starts, the index variable is initialized with zero and is then consecutively used as the index and target for the reads (See Figure **3.1 Conceptual Code for P-Chase**).

1	2	...	$n-1$	0

Table 3.1.: P-Chase Array

```
int j = 0;
// Round 1: Fill values in cache
for (int i = 0; i < arrSize; ++i) {j = array[j];}
// Round 2: Load values and measure time
// Measure time before loop
for (int i = 0; i < NUMREADS; ++i) {j = array[j];}
// Measure time after loop
```

Figure 3.1.: Conceptual Code for P-Chase

The dependence of each read on the preceding read is necessary in order to avoid optimizations and reordering of the code by the GPU, which would distort the benchmark or even prevent it completely. The P-Chase method consists of two rounds: The first round is a loop, which loads all values of the array exactly once to avoid cold cache misses and to have all values in the specific cache. The second round executes a fixed number of reads from the cache and measures the overall required time to do so. It starts the benchmark with a rather small array and applies the above steps. Then the array size is continuously increased until the measured time suddenly rises due to more and more cache misses for each iteration (See **3.1.3 P-Chase Cache Perspective**). Afterwards, it is possible to conclude the cache size from the array size that was used in the iteration in which the increase of the measured time started.

3.1.2. Fine-Grained P-Chase Method

The Fine-Grained P-Chase method is an advanced modification of the previous P-Chase method, since it applies the measurement for each load individually to obtain more detailed information about the whole benchmark procedure. Specifically, it modifies the second step to put the time measurement inside of the loop around the data access (See code at Figure **3.2 Code for concept of Fine-Grained P-Chase**). Unfortunately, it is possible, that the CUDA compiler nvcc decides to reorder the time measurement instructions and the data access due to the independence of these instructions. Moreover, it is possible, that the data access and the time measurement are started concurrently due to the support of *"dual-issue"*-handling of independent instructions. This means, two independent instructions can be executed concurrently on the GPU. As a consequence, it is possible, that the time measurement after the loading instruction finishes before the load is completed. Hence, it is necessary to include dependent code between the data access and the second time measurement to prevent reordering and *"dual issue"*-handling. Since it records the data accesses for each index, the resulting data consists of lots of more loading times (for each array size and each index). Details about the concrete implementation of that can be found at **4.2 Cache Size**.

```
for (int i = 0; i < NUMREADS; ++i) {
    // Measure time before access
    j = array[j];
    // Code to prevent reordering
    // Measure time after access
}
```

Figure 3.2.: Code for concept of Fine-Grained P-Chase

3.1.3. P-Chase Cache Perspective

Figure 3.3 illustrates how the cache is filled during the benchmark if the array size is smaller than the size of the cache. The left side describes the moment of the benchmark while the right side shows the cache(lines). The empty cache lines are filled with values that are not relevant to the benchmark, as they are loaded into the cache by other programs prior to the benchmark execution. At the beginning (third row), no values are loaded to the cache. After the first round the cache is filled with the values of the benchmark (fourth row). Since the array size is smaller than the cache size, there is still a location in the cache that is unused by the benchmark. The second round will reload the values from the cache and will not have any cache misses (last row).

Figure 3.4 shows how the cache progresses if the array size is larger than the size of the cache. The first round loads all the values into the cache (fourth row, fifth row) until the first conflict/capacity cache miss occurs after which the cache replaces one cache line according to the replacement policy (marked in red, first cache line without loss of generality). In the second round the cache tries to reload all values and is confronted with cache misses for the values that were replaced in the first round (last row). Consequently, there is at least one cache miss in the second round and the cache needs to replace another cache line again (marked in red, second cache line w.l.o.g.). Depending on the cache replacement policy and cache associativity, the second round will have at least one but more likely a few more cache misses. This increases the measured time of the benchmark compared to a run with an array size smaller than the cache size with no cache misses.

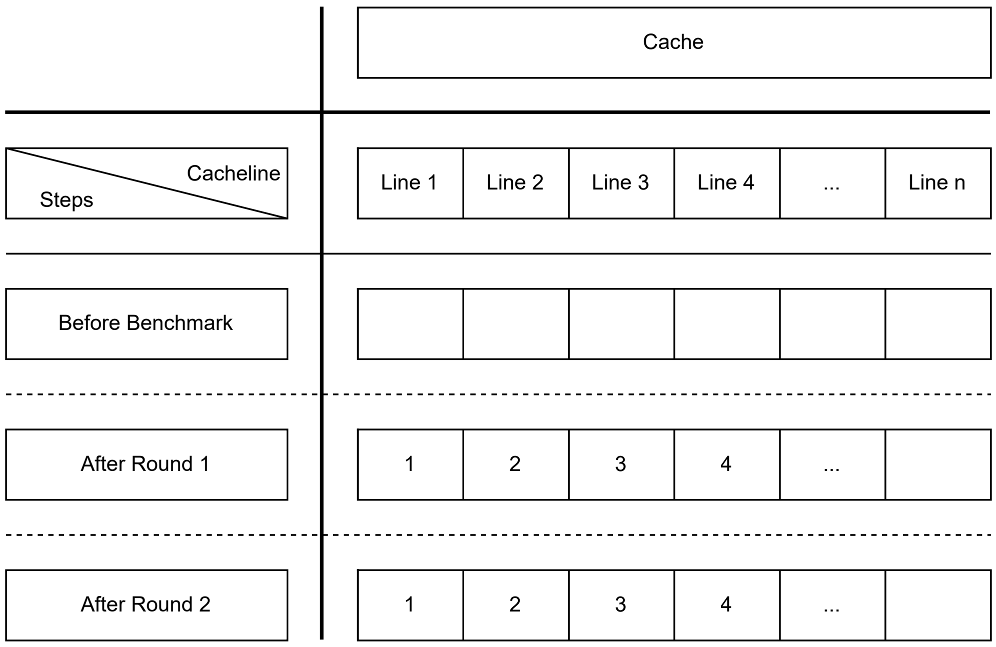

Figure 3.3.: Cache during P-Chase (Array size < Cache Size)

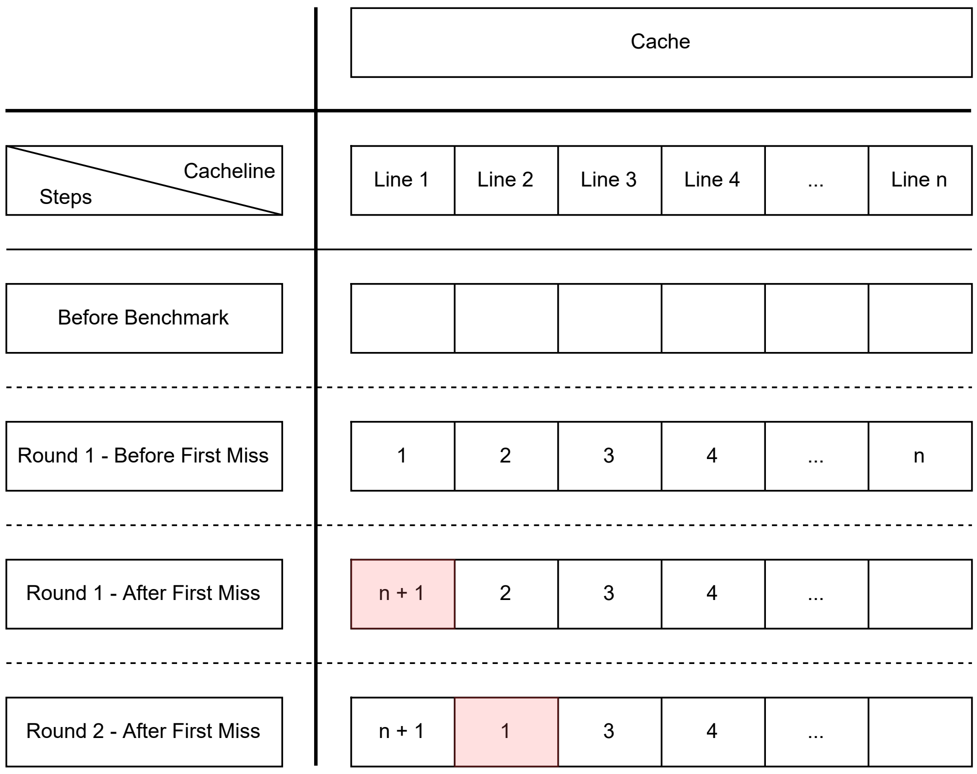

Figure 3.4.: Cache during P-Chase (Array size > Cache Size)

3.2. Efficient Search for Cache Size

As it can take quite a lot of time to execute the P-Chase benchmark on lots of arrays, it would be rather inefficient to run it sequentially with a very small array as starting point and a very huge array as endpoint even if the first or last hundreds of array sizes do not contain the point of significance. Due to this circumstance it is proposed to apply a special search to narrow down the region in which the cache size resides. The search consists of two steps (also illustrated in Figure **3.5 Searching for Cache Size)**:

1. Search by doubling the array size: It starts with an array size of 1 KiB. Then it always doubles the array size until a significant change in the loading time is detected. Thus, it uses the array size 2 KiB, 4 KiB, 8 KiB, etc. It passes to the second step with the lower boundary, which did not have an increased loading time and the upper boundary in which the increase was detected.

2. Search binary-wise: In the region that was given by the first step, first check if the change occurs in the first or second half of the region. It recursively narrows down the region until the lower and upper boundaries are close enough.

3.3. Change Point Detection

3.3.1. General

Change Point Detection (CPD) is the process of detecting and verifying change points in a series of input values, often a signal or a time-based series. More formally, a series of values can be defined $S = \{x_1, x_2, ..., x_n\}$ with x_i being either one- or multidimensional. The values x_i can be seen as the results of their corresponding random variables X_i, which follow the distribution $F_i(X)$ [37] [38]. The change point is the index i with

$$F_0(X) = F_1(X) = ... = F_{i-1}(X) \neq F_i(X) = F_{i+1}(X) = ... = F_n(X)$$

In other words, the first $i - 1$ values all follow the same distribution until a sudden change in a characteristic of the distribution occurs whereby the rest of the values follow a different distribution. This could mean a shift in the mean, the variance or a completely other distribution function. The problem formulation of finding one change point can be generalized into finding an arbitrary number of change points in the series if this is required. Furthermore, there are other interesting approaches for solving the problem of CPD including machine learning algorithms or autoregression models [38].

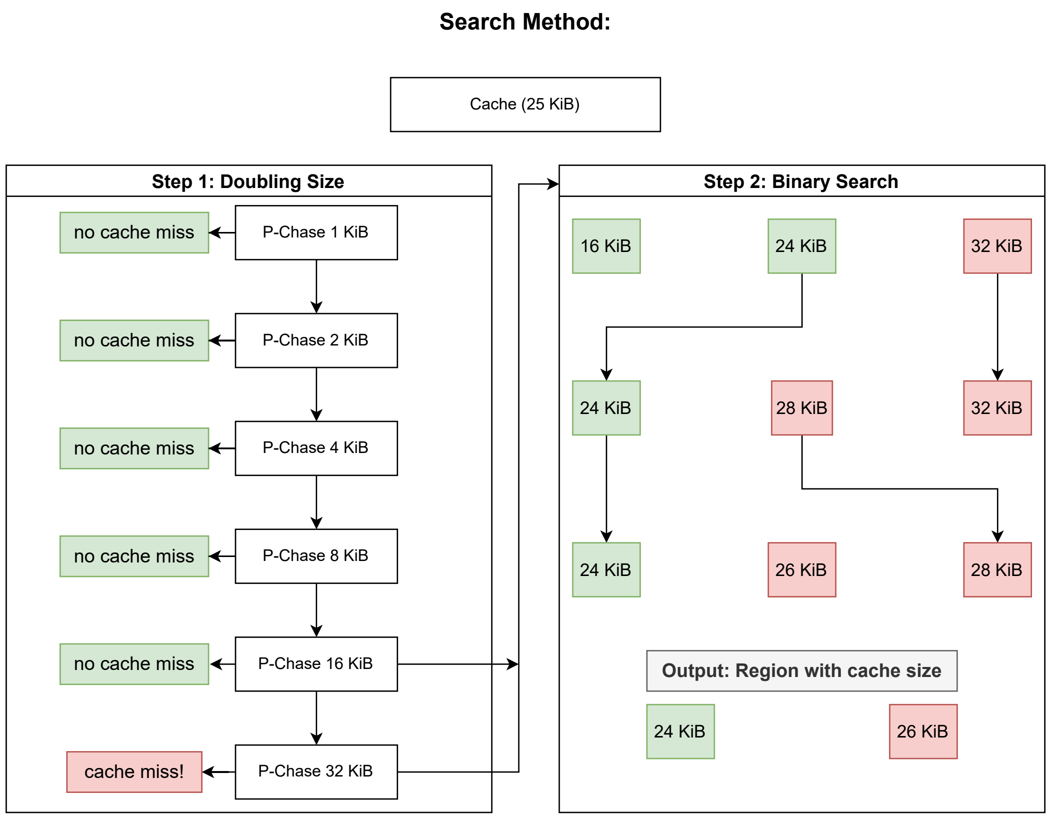

Figure 3.5.: Searching for Cache Size

The two main tasks for CPD are in particular to estimate the most likely change point i in a series of values (estimation) and to calculate a statistical measure, which is able to confirm or negate if the estimated change point is a real change point (testing). If the series consists of multidimensional values, an additional step can be applied at first, the reduction of the complexity of the problem.

3.3.2. Reduction

There are two possible options in case of multidimensional data. While the first one stays in the multi-dimensional sphere, which increases the computation complexity, another way is to reduce the problem of multiple dimensions into a one-dimensional problem. Simple examples to transform a multidimensional x_i^k would be the calculation of the average or the vector length for each multidimensional value. Another possibility to reduce the overall complexity of the problem is stated in [39], which first applies a filtering of the collected values followed by the creation of *censored* series of values. By this means, the problem stays multi-dimensional but only considers the values that are relevant for the CPD. A well-described method to reduce the complexity into one dimension can be found in [40]. Here the reduction utilizes geometric structures to map an n-dimensional to a single-dimensional value. It uses a reference vector - for example the vector $\mathbb{1} = (1, ..., 1)^T$ - to calculate for each n-dimensional value the distance to the reference vector and the angle between both vectors. Testing of these methods revealed the geometric mapping works best for the desired purpose since the other ones were either less accurate (average) or had a considerably higher computation time (censored time series).

3.3.3. Estimation

Assuming that the problem is a one-dimensional one, the estimation of the most likely change point can be accomplished in various ways. Since the size of a cache can be analyzed by exactly one point in the series of values that are obtained by the benchmark, the estimation can be limited at this point to the finding of exactly one change point. The optimal change point value can be retrieved via a dynamic programming approach `Opt` as described in [41]. Because the proposed estimation algorithm works with a varying number of change points, it can be simplified due to the limitation of only one change point. The simplified algorithm can be found at the outlined code section **Algorithm 1: Simplified Opt**. Intuitively, it creates for each subvector of the series of values $S_{u...v}$ a measure of "*equivalence*" (homogeneity in [41]), the *cost*-value $c(u, v)$. The cost function is built in a way such that the cost value gets lower, the more all values in a vector are similar. Consequently, it is necessary, to find the index i with

$$i = argmin_i \left[cost(0, i) + cost(i + 1, n)\right] \tag{3.1}$$

Algorithm 1: Simplified Opt with only one change point $\mathcal{O}(T^2)$

```
  Input: Series of values S with size T
  Output: Change Point Index $t^*$
1 forall $(u, v), 1 \leq u < v \leq T$ do
      $C_1(u, v) \leftarrow cost(u, v, S)$ ; // cost of each subvector $S_{u...v}$
  end
  $s \leftarrow T - 1$;
  $t^* \leftarrow argmin_{1 \leq t < s}[C_1(1, t) + C_1(t + 1, s)]$;
  return $t^*$;
```

The previously mentioned `cost()`-function can be built in many possible ways, e.g the empiricial mean of a subvector [41], maximum log-likelihood-based cost values [42], rank statistic values [37] or many others.

3.3.4. Testing

After obtaining an estimation for the most significant change point, it is necessary to verify whether the estimated value is a real change point. Potentially, the change at the estimated change point could be small enough not to count as a real change point. As an example, the series of values (1,1,1,1,1.3,1.3,1.3,1.3) has a rather small shift compared to the series (1,1,1,1,4,5,6,7). With the definition of a change point, this verification step can be redefined as a hypothesis test with a given sample S, the null hypothesis H_0: "*All distributions are equal*" and the alternative H_1: "*There is a change in the distribution*" or more formally:

1. $H_0 : F_0(X) = F_1(X) = ... = F_n(X)$
2. H_1 : There exists an index $0 < i < n$ with $F_0(X) = F_1(X) = ... = F_{i-1}(X) \neq F_i(X) = F_{i+1}(X) = ... = F_n(X)$

The confidence level α can control how strong the change needs to be to count as a real change point. One hypothesis test that can be used for the given hypothesis, is the *Kolmogorov-Smirnov hypothesis test*, in particular the Two-Sample Kolmogorov-Smirnov test [41] [43]. This tests if two given samples follow the same distribution, in this case the first (sub-)sample from index 0 to the estimated change point i and the second sample from (i+1) to the end. In summary, this test can be applied with a given confidence level α and the following procedure [44, Ch.5.4 p.191] [45, Ch.7 p.318-322]:

Let F(x) and G(x) be the empirical cumulative distribution functions (ECDF) of the first and the second sample. Then the test statistic D can be calculated via

$$D = max_x \left|(F(x) - G(x))\right| \tag{3.2}$$

Now the critical value d_α, corresponding to the given α-value, can be approximated (on the condition that the samples are sufficiently large) via [46, Ch.5 p.152]:

$$d_\alpha = \sqrt{\frac{1}{2} \cdot \frac{n+m}{n \cdot m} \cdot \log(\alpha/2)} \tag{3.3}$$

If $D > d_\alpha$ holds then the change point can be considered a real change point, as the change is large enough between the two samples. If $D \leq d_\alpha$ then the change point will be discarded, as the change is too low.

3.4. Overview

The proposed design of the approach can be described as follows: For a cache that is to be benchmarked, first the Fine-Grained P-Chase method needs to be applied. Ideally, the method starts at an array size that is already relatively close to the cache size and then stops shortly after the cache size is reached. For that reason, the more efficient search algorithm for the cache size is executed. After the search gives the region in which the cache size resides, the Fine-Grained P-Chase method is applied sequentially. This returns the required data that consists of the loading times of each index for every array size in the region, thus, it is multidimensional. At this point, the CPD is started. Initially, it reduces the multiple dimensions into a single dimension by applying the previously described geometric mapping. The resulting one-dimensional series of values is searched for the most likely change point by executing the `Opt`-algorithm. The obtained change point is verified by the two-sample Kolmogorov-Smirnov hypothesis test, which either verifies the validity of the change point or rejects it. If the change point is rejected then the region did not contain a valid change point. In such a case the next step is - depending on the cache - either to widen the region for a larger search or to use the information of having no existing change point by stating that the cache is definitely larger than the last size in the region. Moreover, the P-Chase method is also used to deduce the cache line size, the approximate latency of each memory and cache and for other benchmarks described in the next chapter.

4. Implementation

This chapter explains the details of the concrete implementation of the proposed design. The first section briefly explains what is receivable by CUDA-interfaces before it goes into detail about the sizes of each lower-level cache. Afterwards, the implementation for the latency values is highlighted. Additionally, this chapter contains a section about how to check which caches share/use the same physical cache and how to obtain the number of specific caches per SM. The last section explains the possibility of further optimizations, which should boost the overall execution time of the tool.

4.1. Nvidia/CUDA interface

Before the implementation of the design is described, this section serves as a listing of which information can already be retrieved via CUDA or Nvidia interfaces directly. This is what the `cudaGetDeviceProperties()`-interface makes available:

- Name: the name of the GPU, e.g. Nvidia RTX 2080 Ti
- Compute Capability: the compute capability of the GPU
- Shared Memory per Thread-Block and per SM
- Number of registers per Thread-Block and per SM
- The maximum available constant memory
- The Main Memory size: not provided by `cudaGetDeviceProperties()` but `cudaMemGetInfo()`
- The L2 Cache Size
- the number of SMs
- the maximum number of threads per block
- the maximum number of thread-blocks
- the memory clock rate

- the GPU clock rate
- the memory bus width

In addition, the number of cores in the GPU or per SM can be added with a header (`cuda_helper.h`) that is not provided in the CUDA SDK but in a separate Nvidia-controlled Github repository [47] that was added into the final implementation as a git submodule. For future releases of new GPUs, it is necessary to get the latest version via `git submodule update --init --recursive`. Since the Nvidia NVML library also contains the querying of the total number of cores in a GPU, it would be possible to migrate to NVML as well. However, the current implementation still uses the mentioned header file, since the respective query NVML function is not included in CUDA 11.4, which is the CUDA version of one of the testing systems. More about the NVML library can be found at **7.5 NVML (Related tool)**.

4.2. Cache Size

This section covers capturing the sizes of each cache in the GPU apart from the L2 Cache size, which is already obtained directly from CUDA. As described in the previous chapter, the implemented tool first uses a search algorithm to narrow down a region in which the cache size will be searched for. Afterwards, every array size in the region is benchmarked sequentially. Both the search algorithms and the sequential benchmarking make use of the Fine-Grained P-Chase method. All concrete results can be viewed in chapter **5 Results**.

4.2.1. L1 Data Cache

Since some GPUs do not support caching of global loads in the L1 Data Cache, the tool first evaluates if caching of global loads is supported for that. The evaluation is done with two runs of the P-Chase benchmark method on a very small array. The first run applies the P-Chase benchmark and tries to use the L1 Cache for all loads. The second run does the same with the restriction of only using the L2 Cache for all loads. Fortunately, this can be controlled since CUDA Version 2.0 by explicitly stating with two cache operators how global loads are handled [24]:

- `ld.global.ca.u32 - .ca`: Try to cache global loads at all cache levels (L1, L2)
- `ld.global.cg.u32 - .cg`: Only cache global loads at L2 Cache

Essentially, the first run measures the loading time while using `ld.global.ca.u32`, whereas the second run measures the time using `ld.global.cg.u32`. If the L1 Data Cache supports the caching of global loads, then the loading times of the two runs are fairly different. But if the L1 Data Cache does not support the caching of global loads, then `ld.global.ca.u32` is only able to utilize the L2 Cache and, consequently, the loading time of both runs is very much the same. In that case it is not possible to apply benchmarks on the L1 Data Cache, therefore, it is left out or ignored by the tool.

If caching of global loads into the L1 Data Cache is supported, then the search, described in **3.2 Efficient Search for Cache Size** is executed. Afterwards, the Fine-Grained P-Chase benchmark is applied sequentially. The creation of the array that is used by the GPU in each benchmark iteration is executed as illustrated in Figure **4.1 Array Initialization**:

```
unsigned int* h_a = (unsigned int*) malloc(sizeof(unsigned int) * N);
unsigned int* d_a;
cudaMalloc((void **)&d_a, sizeof(unsigned int) * N);
for (int i = 0; i < N; i++) {h_a[i] = (i+1) % N;}
cudaMemcpy(d_a, h_a, sizeof(unsigned int) * N, cudaMemcpyHostToDevice);
```

Figure 4.1.: Array Initialization with CUDA

It is worth mentioning, that allocation of memory and copying have extra functions for CUDA that are required for utilizing the memory of the GPU. The array is then created just like it was described in **3.1.1 General P-Chase Method** with incrementing values. After that, the kernel function is called, which is implementing the Fine-Grained P-Chase benchmark. This is essentially structured into two rounds, as it was described in the design proposal. The first round loads the whole array such that it is stored in the cache (See Figure **4.2 Round 1: Loading array into L1**).

```
unsigned int* ptr; unsigned int j = 0;
for (int k = 0; k < array_length; k++) {
    ptr = my_array + j;
    asm volatile("ld.global.ca.u32 %0, [%1];"
    : "=r"(j) : "l"(ptr) : "memory");
}
```

Figure 4.2.: Round 1: Loading array into L1 Data Cache

Here inline PTX assembly is applied to ensure, that the L1 Data Cache is used for the caching of those loads, as for some GPUs C-like loads are only cached on the L2 Cache. For example, it is described in [23], that the caching of global loads can be switched on for some Kepler devices via compile flag `-Xptxas -dlcm=ca`, which then results in all load instructions having the `.ca` modifier. Consequently, it is possible to drop the compile flag and to add the `.ca` modifier for load instructions where it is necessary. The second round is then executed (See code in Figure **4.3 Round 2: Measuring loading time**). The first line of the inline assembler stores the current clock time before the load

```
for (int k = 0; k < measureSize; k++) {
    ptr = my_array + j;
    asm volatile ("mov.u32 %0, %%clock;\n\t"
        "ld.global.ca.u32 %1, [%3];\n\t"
        "st.shared.u32 [smem_ptr64], %1;"
        "mov.u32 %2, %%clock;\n\t"
        "add.u64 smem_ptr64, smem_ptr64, 4;"
        : "=r"(start_time), "=r"(j), "=r"(end_time)
        : "l"(ptr) : "memory");
        s_tvalue[k] = end_time-start_time;
}
```

Figure 4.3.: Round 2: Measuring loading time

instruction is executed. Due to instruction level parallelism and *"dual issue"*-handling, it is necessary to have an instruction after the load that is dependent on the loaded data. Otherwise, the second `mov` of the clock register could and most likely will finish before the load was completed. For this reason, an additional store instruction of the loaded value to the shared memory is inserted. This workaround of using shared memory was first used by [10]. The shared memory pointer needs to be set with the PTX instruction `cvta.to`, which forms in connection with the modifier `shared` a generic address into an address to shared memory (See code in Figure **4.4 Shared Memory Pointer Conversion**).

After the second round, the measured loading time and the loaded indices are copied back to the host memory with `cudaMemcpy` and the modifier `cudaMemcpyDeviceToHost`. Those values are then further used by the change point detection to extract the cache size of the cache. For demonstration and debugging purposes, those values are also used to calculate the average loading time for each array size. The plot of these average values can be found at Figure **4.5 Average Loading time for L1 Data Cache**. In the

```
__shared__ long long s_tvalue[measureSize];
__shared__ unsigned int s_index[measureSize];
    asm volatile(" .reg .u64 smem_ptr64;\n\t"
                 " cvta.to.shared.u64 smem_ptr64, %0;\n\t"
                 :: "l"(s_index));
```

Figure 4.4.: Shared Memory Pointer Conversion

plot the average loading time stays equal until the array size goes beyond the size of 25.5 KiB, which marks approximately the cache size. The plot of the calculated geometric distances can be seen in Figure **4.6 Geometric Distance**. This shows one of the advantages of using the geometric reductions, as the increase in the distance is by far the highest at the location of the cache size, whereas the average loading time increase at the cache size is roughly the same as the other shifts in the average loading time plot. This helps the CPD quite a lot to correctly identify the cache size.

4.2.2. Texture Cache

The workflow of obtaining the Texture Cache size stays fundamentally the same as it was described for the L1 Data Cache size. However, there are some important differences to the L1 Data Cache, which are explained in this section. The creation of the array stays equivalent to the benchmark for the L1 Data Cache but it is necessary to bind the array into a texture. This can be done in two ways:

- Texture Reference
- Texture Object

Since the usage of texture references is already marked as deprecated in the latest CUDA version, it was decided to use texture objects for that purpose. A code fragment that binds an array to a texture object can be observed at the code in Figure **4.7 Creation of Texture Object**, which utilizes and modifies the example from [48]. This code fragment is executed by the host prior to the kernel invocation. The `cudaTextureObject_t` will be given to the kernel as a parameter. In line 4 the resource pointer will be set to the pointer of the array. In the subsequent lines the type, the size of the type and the size in bytes are specified. In the last line the texture object is finally created.

During the kernel execution the first and the second round also have important differences compared to the L1 Cache size implementation (See Figure **4.8 Round 1:**

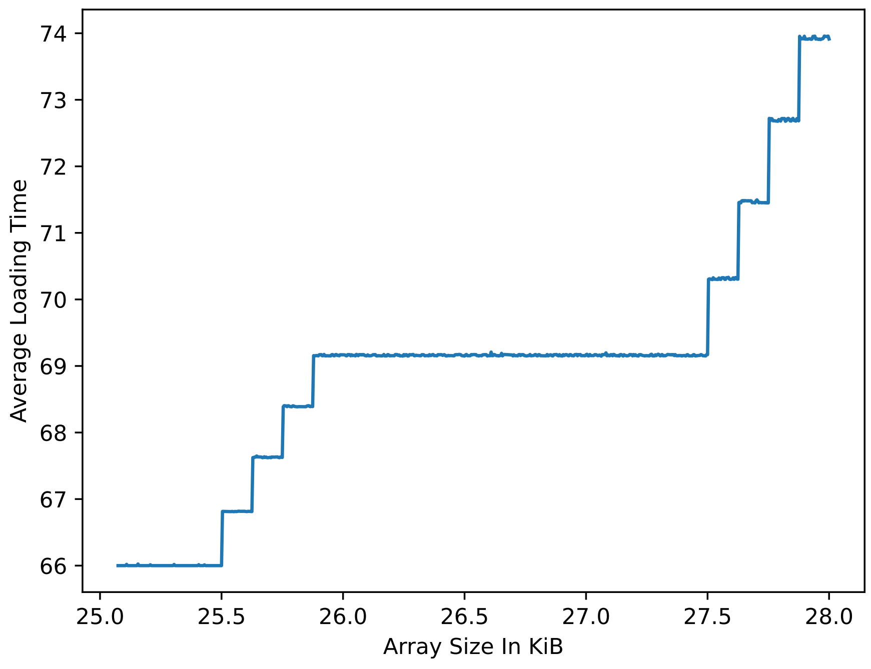

Figure 4.5.: Average Loading time for L1 Data Cache

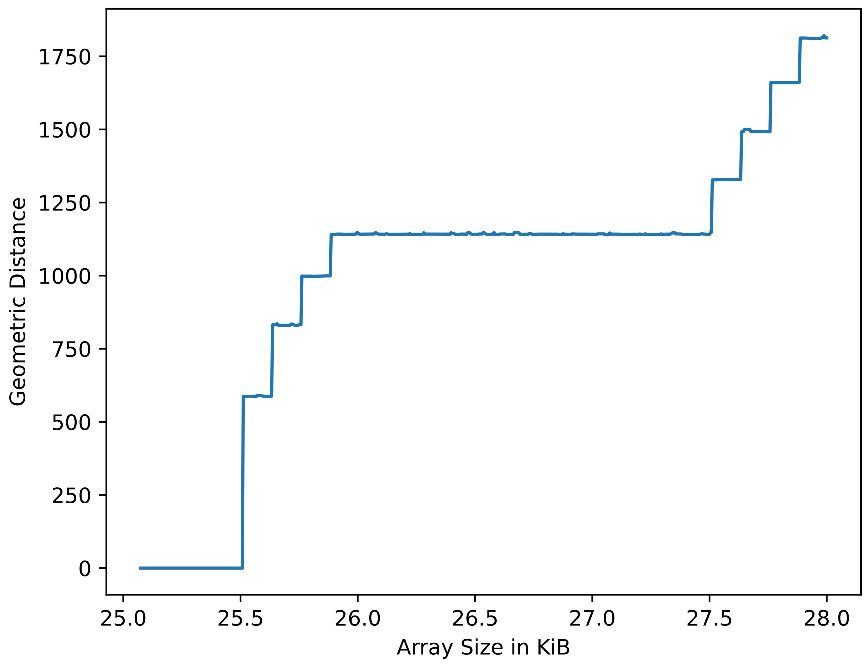

Figure 4.6.: Geometric Distance

```
cudaTextureObject_t tex = 0;
cudaResourceDesc resDesc = {}; memset(&resDesc, 0, sizeof(resDesc));
resDesc.resType = cudaResourceTypeLinear;
resDesc.res.linear.devPtr = d_a; //created with cudaMalloc, cudaMemcpy
resDesc.res.linear.desc.f = cudaChannelFormatKindSigned;
resDesc.res.linear.desc.x = 32; // bits per channel
resDesc.res.linear.sizeInBytes = N*sizeof(int);
cudaTextureDesc texDesc = {}; memset(&texDesc, 0, sizeof(texDesc));
texDesc.readMode = cudaReadModeElementType;
cudaCreateTextureObject(&tex, &resDesc, &texDesc, nullptr);
```

Figure 4.7.: Creation of Texture Object

Texture Cache Size and Figure **4.9 Round 2: Texture Cache Size**). The most significant change is the loading of the values, which is done for textures via `tex1Dfetch<int>()`.

```
__global__ void texture_size (cudaTextureObject_t tex,...) {
   [...]
   int j = 0;
   for (int k = 0; k < size; k++) {j=tex1Dfetch<int>(tex, j);}
```

Figure 4.8.: Round 1: Texture Cache Size

```
for (int k=0; k < measureSize; k++) {
    start=clock();
    j=tex1Dfetch<int>(tex, j); s_index[k] = j;
    end=clock();
    s_tvalue[k] = (end -start);
}
```

Figure 4.9.: Round 2: Texture Cache Size

The plot of the average loading time can be found at Figure **4.10 Average Loading time for Texture Cache**. It can be observed, that the proposed change point detection is robust against outliers since it correctly returns the cache size of approximately 25 KiB for this plot. It does not return the size of 24.25 or 24.6 KiB at which cache misses occurred randomly a bit before the actual cache size is reached.

4.2.3. Read-Only Cache

For the Read-Only Cache benchmark the code of the preparation prior to the kernel invocation stays exactly the same as the L1 Data Cache benchmark code. However, the kernel code has some slight modifications that are mentioned now. First, the array parameter must be declared as const and restricted in order to use the Read-Only Cache:

```
__global__ void RO_size (const unsigned int* __restrict__ my_array, ...)
```

Furthermore, loading the values is adjusted to:

```
j = __ldg(&my_array[j]);
```

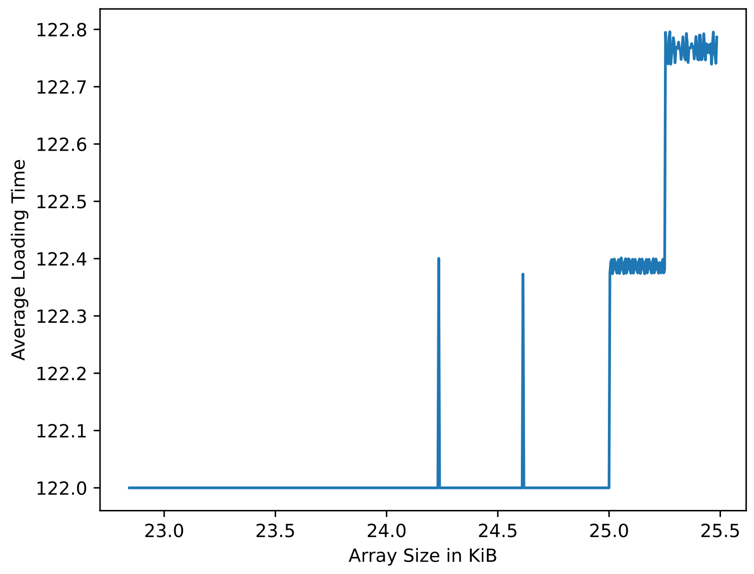

Figure 4.10.: Average Loading time for Texture Cache

Please note that the `__ldg()`-instruction was introduced with CUDA version 3.5, therefore, the tool is restricted to GPUs with CUDA 3.5 or above. As a consequence, all microarchitectures since Nvidia Kepler are supported. The plot of the Read-Only Cache is illustrated at Figure **4.11 Average Loading time for Read-Only Cache**

4.2.4. Constant Cache

The Constant Cache can be handled differently than before regarding the Fine-Grained P-Chase method since the constant memory usage is limited to 64 KiB per CUDA module for developers [27, Ch.9.2.6]. Therefore, it is possible to copy the corresponding array to the constant memory pointer in every iteration - similar to the handling for the other caches. However, an even better option is to assign one 64 KiB array to the constant memory and then modify the P-Chase to do the jumps to the start of the array accordingly. The second option also saves many array allocations and array copies. The array is then directly created at its definition with the additional modifier `__constant__`:

```
static __device__ __constant__ unsigned int arr[constArrSize] =
{1, ..., 16047, 0};
```

The two benchmark rounds have both one additional instruction to jump back to the

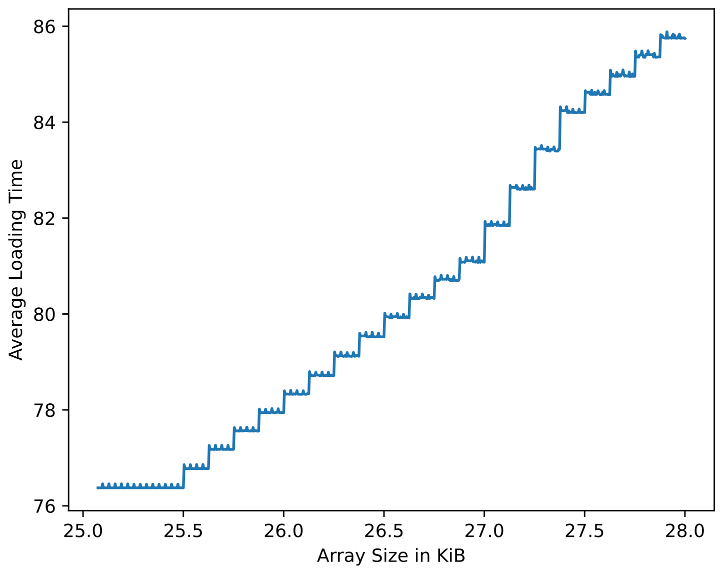

Figure 4.11.: Average Loading time for Read-Only Cache

beginning after the size is reached that is currently measured: `j = j % N;`
Additionally, it was discovered during testing and confirmed by other papers too, that there are actually two caches - the Constant L1 Cache and the Constant L1.5 Cache [33] [34] [49]. It was named Constant L1.5 Cache by these research papers because the loading time of this cache lies between the L1 Caches and the L2 Cache. This phrasing (Constant L1.5 Cache) is adopted by this thesis and the corresponding tool. The benchmark run for the Constant L1 Cache is sequential from the region of a few bytes until the array size goes beyond 5 KiB because the Constant L1 Cache size is never above 2 and 4 KiB. The benchmark run for the Constant L1.5 Cache starts afterwards and runs sequentially until the limit of 64 KiB of constant memory is reached. For the L1.5 Cache it is possible, that the cache is larger than the limit that Nvidia allows the developers to use for a CUDA module (64 KiB). Then the change point detection will reject the most likely change point and the information that the tool returns, is, that the size of the Constant L1.5 Cache is larger than 64 KiB. A plot of both runs can be found at Figure **4.12 Average loading time for Constant L1 Cache** and Figure **4.13 Average loading time for Constant L1.5 Cache**, respectively.

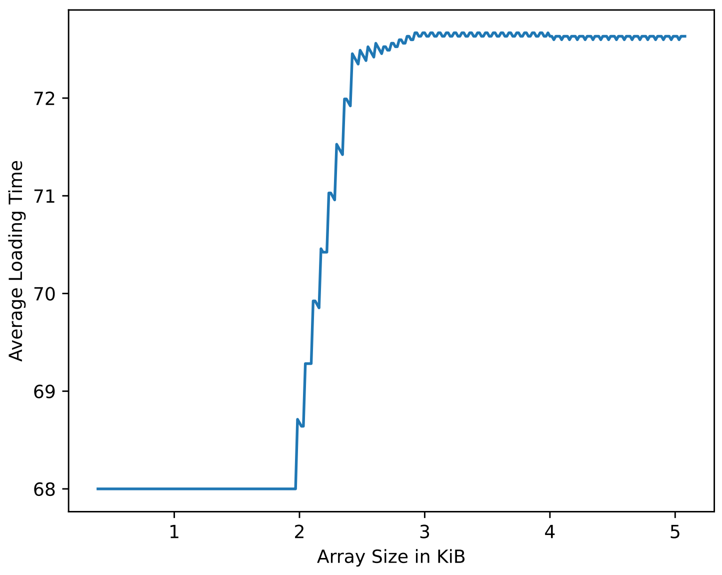

Figure 4.12.: Average loading time for Constant L1 Cache

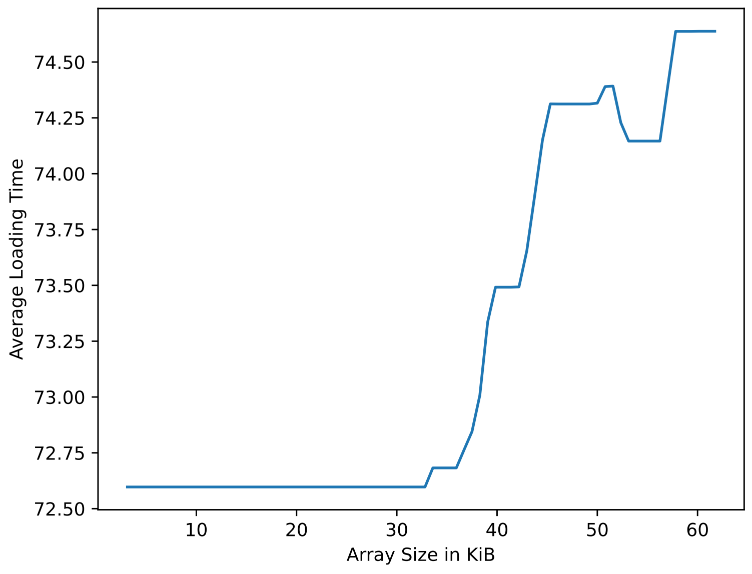

Figure 4.13.: Average loading time for Constant L1.5 Cache

4.3. Cache Line Size

The cache line size is obtained by applying the Fine Grained P-Chase benchmark and subsequently analyzing the distances between the cache misses. Since the Fine Grained P-Chase method returns the loading time for each index, the cache misses and their indices in the array can be detected, as the loading times at these indices are much higher compared to cache hits (See also Figure **4.27 Excerpt of fine-grained measurement**). So the benchmark is started for an array with a size significantly larger than the measured cache size. Thus, the second round of the benchmark produces many cache misses consecutively. If a specific index produces a cache miss, then the next values that can be contained in one cache line are also cached. So the distance from one cache miss to the next cache miss is the cache line size (multiplied by four due to the size of an integer). In theory, the line size could also be deduced in a similar way with cold cache misses (no first round) but testing showed, it is considerably more stable with the Fine-Grained P-Chase benchmark with both rounds and a larger array.

4.4. Latency

The latency benchmarks follow the same principle but the more general P-Chase method instead of the Fine-Grained P-Chase benchmark is applied in order not to interfere with the concrete loading time values. For the L1 Data Cache, the Texture Cache, the Read-Only Cache, the Constant L1 Cache and the L2 Cache, this means, that a small array is loaded into the cache in the first round and is then repeatedly loaded in the second round several times while measuring the whole loop instead of each iteration (See code at Figure **4.14 Latency of the L1 Data Cache**). As a consequence, the additional

```
start_time = clock();
for (int k = 0; k < iter; k++) {
    ptr = my_array + j;
    asm volatile ("ld.global.ca.u32 %0, [%1];"
    : "=r"(j) : "l"(ptr) : "memory");
}
s_index[0] = j;
end_time = clock();
time[0] = (end_time-start_time) / iter;
```

Figure 4.14.: Latency of the L1 Data Cache

overhead produced by the shared memory access is removed. The loop overhead is also reduced due to optimizations done by the compiler, as it partially unrolls the loop. Nevertheless, there is additional overhead because of address computation, which can be revealed by keeping the generated PTX assembly file during compilation. This can be done via the nvcc compile flag `-keep`. A small fragment of the generated file can be found at Figure **4.15 Generated PTX code for latency**. In the `mul`-line the register `%r38` holds the value of `j`. The `mul` instruction computes the shift amount from the start of the array to the correct index. The `add` instruction adds this shift to the array pointer, which relies in register `%rd1`. The `ld`-line loads the value from the computed address. These three instructions are repeatedly linked together in the PTX file. So overall the measured latency has an additional overhead of one `mul` and one `add` instruction. During the evaluation of the results, this overhead was exactly eight clock cycles, which complies with the findings of [34] in which the required cycles of a multiplication instruction and an addition instruction were benchmarked to be both four cycles. This creates an overhead of eight cycles since both need four cycles while they are dependent on the result of the predecessor. Thus, no instruction-level parallelism or *"Dual issue"*-handling could be applied. For the other memories and caches (Shared Memory, Main

```
mov.u32  %r28, %clock;
[...]
mul.wide.u32  %rd33, %r38, 4;
add.s64  %rd34, %rd1, %rd33;
ld.global.u32  %r39, [%rd34];
[...]
st.shared.u32  [s_index], %r64;
mov.u32  %r45, %clock;
```

Figure 4.15.: Generated PTX code for latency

Memory, Constant L1.5 Cache), the latency can be measured in the same way with noteworthy adjustments. First, the shared memory latency is measured likewise but the array initialization needs to be put to the kernel side, as shared memory arrays cannot be set by the host side. The adjustments for the main memory and the Constant L1.5 Cache are mainly necessary in order to ensure, that no other caches are used during the measurement. The main memory latency is measured by using the `.cg` load modifier and by removing the first round of the benchmark. Thus, only the main memory is used if the loads jump over a L2 cache line after each load. Otherwise, the L2 Cache would be used after one cache miss because the consecutive values would be also cached in the L2 Cache. Therefore, the array is initialized with an adjusted shift/stride for the values in the array with the amount of one cache line size (s) (See array at Table

s	$s+1$	...	0	1	...	$s-1$

Table 4.1.: P-Chase array for Main Memory latency (s=L2 Cache Line Size)

4.1 P-Chase array for Main Memory latency). As a consequence, a cache line is cached into the L2 Cache after each load but since the next index is beyond the loaded cache line, the L2 Cache is never used for the loading process. The Constant L1.5 Latency requires further adjustments since the values need to be cached, but the Constant L1 Cache needs to be ignored. Hence, the first round of loading the values is necessary. However, the size of the array that is completely loaded in the first round, needs to have a size that is double or better a higher multiple of the Constant L1 Cache but still below the L1.5 Cache Size. During the first round, the second half of the array will then evict pretty much the whole first part of the array in the Constant L1 Cache. Once the second round of the measurement starts from the beginning of the array, the first part of the array is no longer cached in the Constant L1 Cache but still in the L1.5 Cache. It is again required to add a stride in the array for the Constant L1 Cache line size. The array size of 4000 integers, 16 kB, was used because there is no Constant L1 Cache, whose size is larger than 8 kB or even 4 kB. Indeed, the largest measured Constant L1 Cache, according to a benchmark reference [49], is 4 kB large for an architecture (Fermi) that is not supported by the implemented tool due to its advanced age, as it is also not supported by the latest CUDA versions. It is worth mentioning, that overall the L2 latency and main memory latency suffer from a small variance, as both latency values are also dependent on the performance of structures that were not taken into account by the implementation such as the Translation Lookaside Buffer and Page Table(s).

4.5. Cache Sharing

The next issue that is tackled by the tool, is the question if specific caches are located in the same physical cache. This is possible for the L1 Data Cache, the Texture Cache and the Read-Only Cache. For some microarchitectures, these caches are joint together into one physical cache, which can simply be used in different ways (texture, read-only, read-write) for various purposes. An example of a microarchitecture in which the L1 Data Cache and Texture/Read-Only Cache are separated, is the Kepler microarchitecture. So far, all benchmarks managed to achieve their goal of measuring size and latency with only one thread and one thread block. This next benchmark will use two threads and one thread block, as it combines two Fine-Grained P-Chase benchmarks into one with each thread executing one benchmark. The fundamental idea is to merge the two benchmark runs in such a way, that there will be cache misses if the two caches are located on the same physical cache. If the caches are separated, then there should not be any cache misses. This can be achieved by sequentially controlling, which thread runs its rounds within one kernel function. At this point the CUDA branch divergence handling comes in handy: If a conditional branch is only taken by one thread, the other thread is deactivated during this program path execution. This is stated in [28]: "*If threads of a warp diverge via a data-dependent conditional branch, the warp executes each branch path taken, disabling threads that are not on that path*". Thus, it

```
if (threadIdx.x == 0) {
    for (int k = 0; k < L1_N; k++) {
        ptr = myArray + j;
        asm volatile("ld.global.ca.u32 %0, [%1];"
        : "=r"(j) : "l"(ptr) : "memory");
    }
}
__syncthreads();
if (threadIdx.x == 1) {
    for (int k = 0; k < TextureN; k++) {
        j2 = tex1Dfetch<int>(tex, j2);
    }
}
```

Figure 4.16.: First rounds of both threads

is possible to control the execution order of the rounds of each thread by having each round in a conditional branch that is only taken by the corresponding thread. This is realized with a condition on the special register/variable value `threadIdx.x`, that identifies each thread in a thread block. Since the thread id values are assigned starting from zero until the number of specified threads is reached, the conditions can simply check for equality with zero or one. The execution order can be further controlled by `__syncthreads()`, because all threads stop at this function and proceed only, once all threads have reached this point. The code section shown at Figure **4.16 First rounds of both threads** contains the controlled execution of the first rounds for both threads. First, thread zero executes the first round of the benchmark on the L1 Data Cache followed by the other thread, which executes the first round for the Texture Cache. The arrays, on which these benchmarks are executed, `myArray` and `tex`, respectively, are a little bit smaller than their corresponding cache sizes such that there are no cache misses if the two caches are separated. If the two caches share the same location, then the second conditional branch will evict most of the values of the first conditional branch. The

```
if (threadIdx.x == 0) {
    // Fine grained second round l1 data
}
__syncthreads();
if (threadIdx.x == 1) {
    // Fine grained second round texture
}
```

Figure 4.17.: Second rounds of both threads

second rounds are executed directly afterwards with their corresponding fine-grained measurement similar to their size benchmarks (See code at **4.17 Second rounds of both threads**). Before this kernel function is executed, first the corresponding single-thread benchmarks for the two caches are applied in order to get reference values. For the check, if there are cache misses in the two-thread benchmark, it is sufficient to compare the average loading time values and check for enough deviation from the reference values and by doing so, the tool can automatically analyze if L1 Data Cache and Texture Cache and Read-Only Cache use the same physical cache or not.

4.6. Number of Caches Per SM

The subsequent issue was discovered during testing on a GPU with the microarchitecture Maxwell. The size of the Read-Only/Texture Cache was benchmarked to be approximately 12 KiB. However, according to [28, K.4. Compute Capability 5.x] the SM of this microarchitecture has a total of 24 KiB. Moreover the SM-architecture for Maxwell shows an L1 Data/Texture Cache that is split up into two fragments (See Figure **4.18 SM-internal architecture for Maxwell**). Hence, it can be concluded, that the benchmark only measures/gets the size of exactly one of those fragments. To overcome this issue, an additional benchmark was created. The idea of this benchmark is to count the number of cores, which share the same cache within one SM. This can be accomplished with a technique similar to the benchmark of the previous chapter which checks if L1 Data, Texture and Read-Only Cache are the same physical cache. However, now it is necessary to use the same cache and to iteratively check for the first and second core if they share the same cache, then for the first and third core, etc. Therefore, it is necessary to use two P-Chase arrays again - one array for each thread. The first important remark is, that a specific thread block is distributed to exactly one SM and will stay on it until all threads are finished [28, Ch.4]. Besides, thread ids are assigned to all threads in a thread block increasingly, starting from thread id 0, which

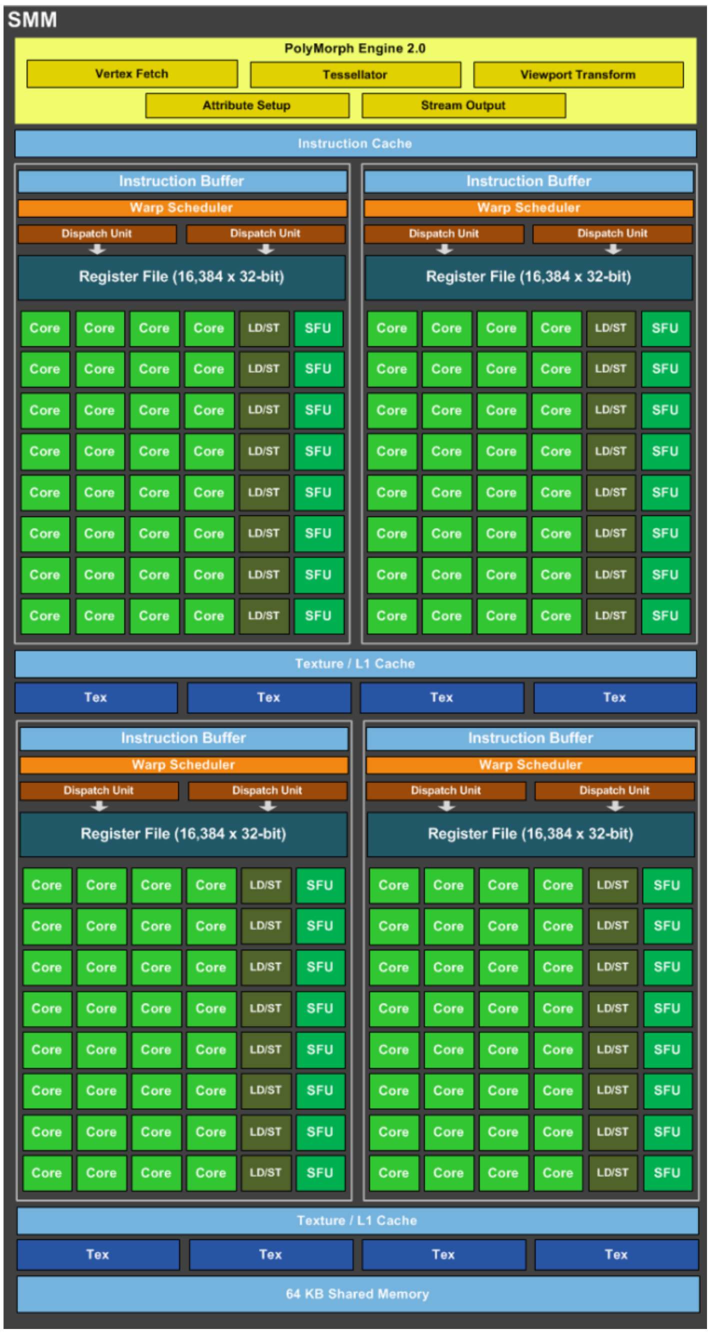

Figure 4.18.: SM-internal architecture for Maxwell [50]

is definitely assigned to the first warp. Moreover, each warp is assigned statically to one warp scheduler [28, Ch.K]. Thus, it is necessary to execute a kernel with at least as many threads as there are cores in one SM, as the threads are distributed to warps and the warps are executed on the cores. It is reasonable to assume, that the threads are distributed equally over all the cores, as the alternative of a mapping of lots of threads to one or only few cores would be highly inefficient. However, it could also be possible, that the warp scheduler assigns the active thread always to the first (available) core of the warp. Even if this is the case, it does not change/falsify the result, since all warps that do not share the same cache are detected by this. The resulting number of threads/cores that share the same cache would then be equivalent to the result if the threads are equally distributed over the cores.

The implementation consists of two rounds again with the first round loading the values into the cache. This is shown in the code at Figure **4.19 First rounds (# of Caches)**. Here the `baseCore` variable will always be zero, and the `testCore` variable

```
if (threadIdx.x == baseCore) {
    //Load N times from first array with p-chase into L1 Data
}
__syncthreads();
if (threadIdx.x == testCore) {
    // Load N times from second array with p-chase into L1 Data
}
__syncthreads();
```

Figure 4.19.: First rounds (# of Caches)

will iterate from one to a specific limit that is described later. For this benchmark the `__syncthreads()`-calls are very important because the threads can be distributed among different warps, however, branch divergence does not deactivate threads of different warps. This means, threads of other warps would jump over the condition and proceed further, which is not intended. So it is necessary to actively block the threads between each conditional branch. This can be accomplished with the `__syncthreads()` function, as it synchronizes all the threads of one thread block, even threads of different warps. The first rounds are followed by the second rounds with the fine-grained measurements, which is shown at Figure **4.20 Second rounds (# of Caches)**. If the threads - assigned to two cores - share the same cache (either L1 Data, Texture, Read-Only), the measurements would contain some cache misses again even if the array sizes are lower than the cache size. If the two threads do not access the same cache

```
if (threadIdx.x == baseCore) {
    // Fine grained measurement with first array
}
__syncthreads();
if (threadIdx.x == testCore) {
    // Fine grained measurement with second array
}
__syncthreads();
```

Figure 4.20.: Second rounds (# of Caches)

because the specific cache is split up like in Maxwell, there should not be any cache misses in the measurements. The code is applied with variable `testCore` and static `baseCore` 0. The deviation of the resulting fine-grained benchmarks from the reference with only one thread is calculated - again with the average loading time. A plot of these deviation values is shown for the Texture Caches of Maxwell and Turing (See Figure **4.21 Texture Cache Maxwell: Disturbance Flow** and Figure **4.22 Texture Cache Turing: Disturbance Flow**). It is necessary, to outline the architectures of Maxwell and Turing, in order to explain both plots. The GPU of the Maxwell plot has 128 cores per SM and a texture cache that is split up into two caches. The Turing GPU has 64 cores per SM and only one texture cache structure. Therefore, the Turing plot does not show any change over the thread ids, whereas the Maxwell plot has exactly a half of the threads with disturbances clearly above another half of the threads, which consequently means, one half of the cores share the cache, whereas another half of the cores access a separate cache. Both plots hint another important property, the limit of the thread id that is tested against thread id 0. It does not use the number of cores per SM but the maximum number of threads per thread block obtained via CUDA. Interestingly, it was observed if, as an example, 256 threads are used for the kernel on a GPU with 128 cores, then the pattern of deviation values repeats after 128, meaning the sequence of deviation values from 1 to 128 is more or less equal to the deviation values from 129 to 256. Most likely, this happens because the warps are distributed equally among the warp schedulers. Using the maximum number of threads per block was actually a workaround for an issue on the Kepler microarchitecture, which can be avoided by this. Initially, the implementation used as many threads as there are cores. Kepler has 192 cores, but its deviation sequence pattern starts to repeat after 128 threads or 256 threads, similar to the Maxwell plot. This indicates, 64 of the cores are apparently not used for the threads or 64 more threads were used. This could be due to various reasons: First, it is possible but very unlikely, that the kernel is also able

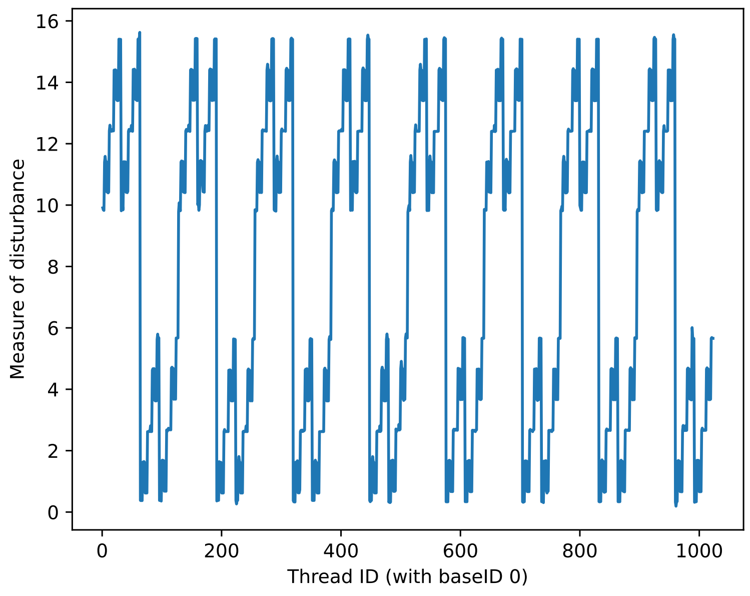

Figure 4.21.: Texture Cache Maxwell: Disturbance Flow

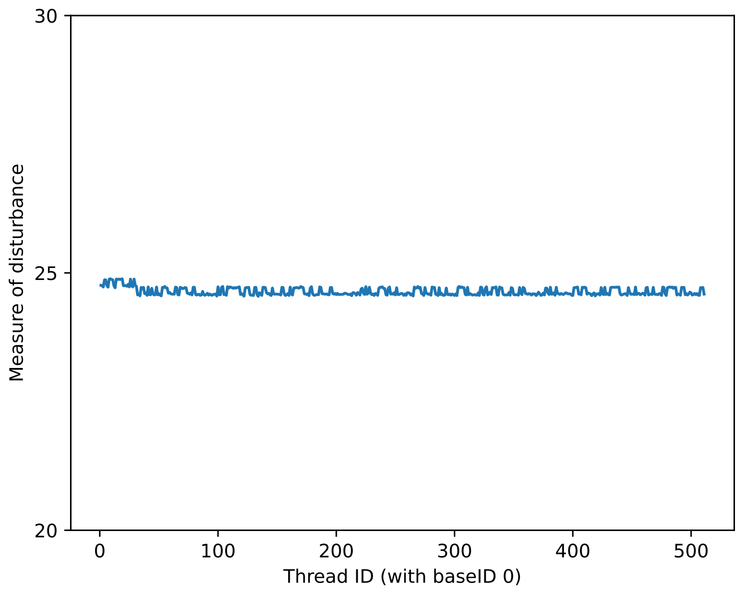

Figure 4.22.: Texture Cache Turing: Disturbance Flow

to use double-precision units for the threads, which would explain that the deviation sequence repeats its pattern also after 256 threads since there are 64 additional separate double-precision units in the SM. Another reason could be, that it is not possible to use the full 192 cores for a specific cache like the Texture Cache. But most likely, the GPU is only able to use 128 cores for the threads, since the architecture has only four warp schedulers. Four warp schedulers result in four parallel warps and as a consequence 128 threads. Now it can be questioned, why the GPU has 192 cores if only 128 can be used. However, all of the 192 cores could be utilized on two independent instructions, since the Kepler GPU supports "Dual issue" [51]. That means, the GPU is able to execute two consecutive independent instructions concurrently. By implication, only three warps with dual-issue instructions are necessary to use all of the 192 cores. For the sake of completeness, the plots for the other microarchitectures are subsequently shown as well.

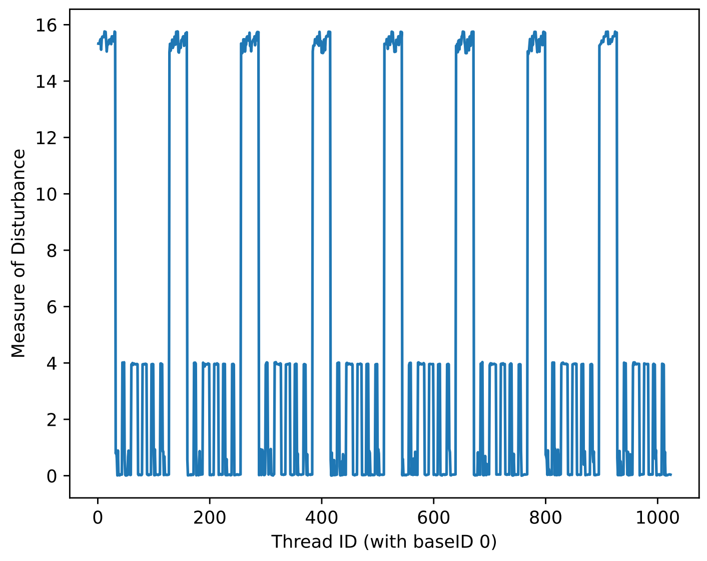

Figure 4.23.: Texture Cache Kepler: Disturbance Flow

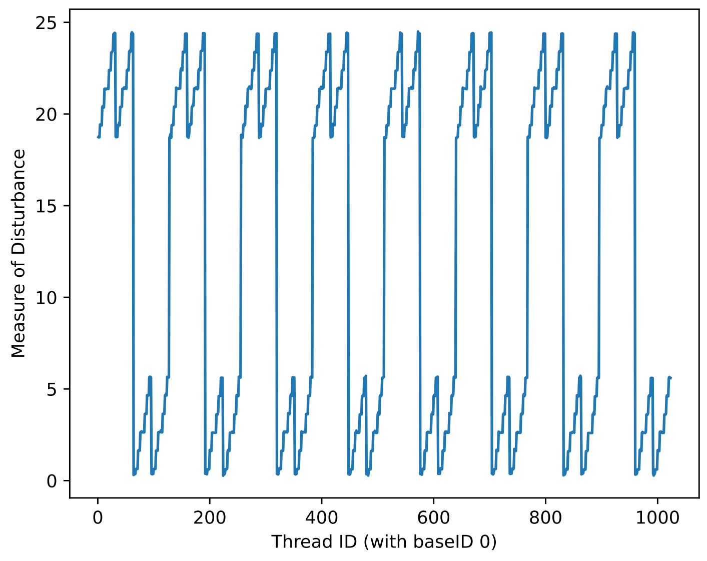

Figure 4.24.: Texture Cache Pascal: Disturbance Flow

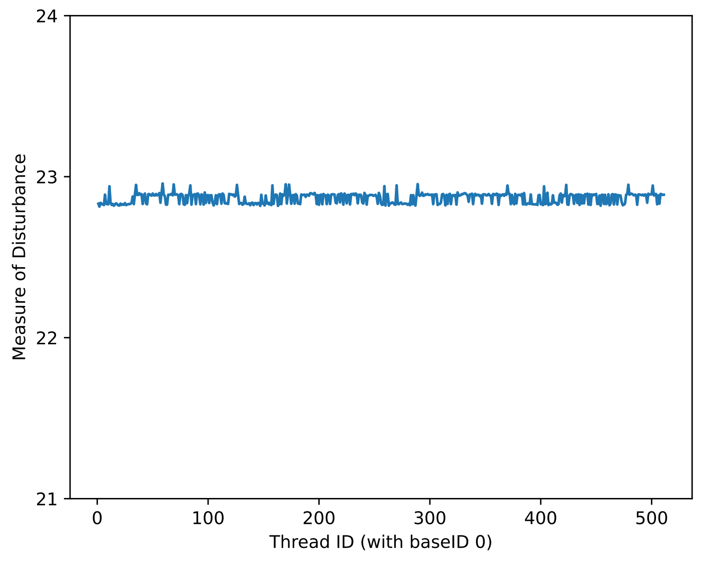

Figure 4.25.: Texture Cache Volta: Disturbance Flow

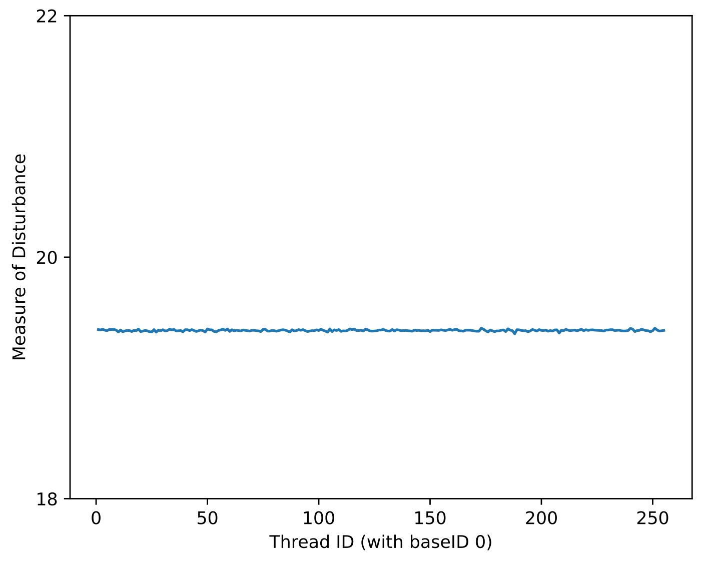

Figure 4.26.: Texture Cache Ampere: Disturbance Flow

```
233 274.000000
234 66.000000
235 66.000000
236 66.000000
237 66.000000
238 66.000000
239 66.000000
240 66.000000
241 273.000000
242 66.000000
243 66.000000
244 66.000000
245 66.000000
246 66.000000
247 66.000000
248 66.000000
249 273.000000
```

Figure 4.27.: Excerpt of fine-grained benchmark measurement.

4.7. Possible Performance Optimizations

This section briefly discusses possible optimizations regarding improved efficiency, that were omitted because of other priorities, more important issues and time conflicts. Nevertheless, it is described how the efficiency of the benchmarks could be further improved.

4.7.1. Jumping with Cache Line Size

One Fine-Grained P-Chase benchmark on one array size executes lots of load instructions, whereas some of the load instructions would be quite unnecessary if the cache line size of the corresponding cache is obtained before (See Figure **4.27 Excerpt of fine-grained measurement**). Thus, the cache size benchmarks could be improved by using a stride for the array values to always jump to the next cache line because those values are the relevant ones, as they determine in the result if the array still fits into the cache. After modifying it accordingly, this adjustment would save lots of load instructions for each iteration. How many loads would be saved concretely, depends on the cache line size of the cache. As an example, for a cache line size of 32 B only one of the eight integer values of one cache line needs to be loaded, hence the overall number of required load instructions per array size is reduced by a factor of 1/8. This applies to both the first round to fill the cache and the second round to measure the loading time.

4.7.2. Utilize all SMs

Since most of the benchmarks focus on SM-internal caches, it is possible to further improve and speed up the benchmarks by encapsulating a number of n iterations/array sizes, where n is the number of SMs in the GPU. Currently, the cache sizes are obtained by sequentially applying the benchmark on one thread block with only one thread. For the improved version one iteration could apply the benchmark for the array size i on the first SM, the array size $i + 1$ on the second SM, etc. until the last SM handles the benchmark for the array size $i + n$. So instead of using one thread inside of one thread block and by that one SM for benchmarking the cache sizes, it should be possible to apply n thread blocks, which should be distributed amongst the SMs in order to fully utilize the GPU. Consequently, the number of iterations would decrease due to parallelism by a factor of n. As an example, if the current implementation applies 800 iterations, so the Fine-Grained P-Chase benchmark is currently executed on 800 array sizes, the improved version puts n array sizes in one iteration and executes them in parallel. This means, for instance with eight SMs, only 100 iterations are required. Therefore, it would be very beneficial for the tool in terms of performance.

5. Results

This chapter outlines the results obtained by the implemented tool.

5.1. Comparison with LIKWID

This section compares the amount of information returned by the implemented tool with an exemplary performance/analysis tool, LIKWID, that provides only a part of the relevant information of the memory topology of a GPU. There may be additional tools, however, they mostly do not offer much more information than LIKWID. The exemplary output of the LIKWID tool for the GPU topology can be viewed at the Figure **5.1 The LIKWID GPU-specific output**.

In total, LIKWID provides concrete numbers for the size of the main memory and the L2 Cache size. Moreover, it delivers GPU and memory clock rate as well as the number of SMs in the GPU in the line with "`Number of SPs`". At last, it gives the memory bus width in bits and the shared memory in bytes. Consequently, there are many details and lots of information that is not covered by LIKWID. In comparison to that, the implemented tool returns approximate values for:

- On SM-level
  - L1 Data Cache: size, cache line size, latency and more
  - Texture Cache: size, cache line size, latency and more
  - Read-Only Cache: size, cache line size, latency and more
  - Constant L1 Cache: size, cache line size, latency and more
  - Constant L1.5 Cache: size, cache line size, latency
  - Shared Memory: size, latency
- On GPU-level
  - L2 Cache: size, cache line size, latency
  - Main Memory: size, latency

```
*********************************************************
GPU Topology
*********************************************************
GPU count: 1
---------------------------------------------------------
ID: 0
Name: Quadro RTX 6000
Compute capability: 7.5
L2 size: 6.00 MB
Memory: 24.00 GB
Clock rate: 1770000 kHz
Memory clock rate: 7001000 kHz
Number of SPs: 72
Max. threads per SP: 1024
Max. threads per block: 1024
[...]
Shared mem per block: 49152
Memory bus width: 384
[...]
--------------------------------------------------------
```

Figure 5.1.: The LIKWID GPU-specific output

5.2. Tested GPU devices

For each supported microarchitecture, the tool is tested on one concrete GPU. Listed below are the existing Nvidia microarchitectures of the past (and future) years.

Architecture	Compute Capability	Release year	Supported[1]
Tesla	1.x	2006/2007	no
Fermi	2.x	2010	no
Kepler	3.x	2012	yes
Maxwell	5.x	2014	yes
Pascal	6.x	2016	yes
Volta	7.0, 7.2	2017	yes
Turing	7.5	2018	yes
Ampere	8.0, 8.6, 8.7	2020	yes
Ada Lovelace	8.9	End of 2022[2]	not yet
Hopper	9.0	2023[2]	not yet

[1] Supported by latest CUDA SDK
[2] Unreleased

Table 5.1.: Nvidia microarchitectures

All architectures currently supported by the CUDA SDK are also supported by the implemented tool. The next table contains information about the GPUs and the corresponding systems (Processor & OS) on which the tool was tested.

GPU	Architecture	Processor	OS
Nvidia Tesla[1]K20c	Kepler	AMD Ryzen Threadripper PRO 3955WX	Ubuntu 20.04.4 LTS
Nvidia GeForce 840M	Maxwell	Intel i7-4710MQ	Ubuntu 22.04.01 LTS
Nvidia Quadro P6000	Pascal	AMD Ryzen Threadripper PRO 3955WX	Ubuntu 20.04.4 LTS
Nvidia Tesla[1]V100	Volta	Intel(R) Xeon(R) Platinum 8360Y	SUSE Linux Enterprise Server 15 SP3
Nvidia RTX 2080Ti	Turing	Intel i9-9900K	Windows 11 22H2
Nvidia A100	Ampere	AMD Ryzen Threadripper 2990WX	Ubuntu 20.04.4 LTS

[1] not to be mixed up with the Tesla microarchitecture, as this is also the name of a series of General-Purpose GPUs

Table 5.2.: Tested GPUs & Systems

The subsequent table shows the specifications for each tested GPU collected from various Nvidia datasheets and CUDA programming or tuning guides provided officially by Nvidia [28, Ch. K] [20] [50] [52] [53] [54] [55] [56] [57].

	K20c	**GF840M**	**P6000**	**V100**	**2080Ti**	**A100**
Compute Capability	3.5	5.0	6.1	7.0	7.5	8.0
L1 Data Cache Size	16 KiB	24 KiB	48 KiB	32 KiB	32 KiB	28 KiB
Shared With[1]	-	TXT, RO	TXT, RO	TXT, RO	TXT, RO	TXT, RO
Texture Cache Size	48 KiB	24 KiB	48 KiB	32 KiB	32 KiB	28 KiB
Shared With[1]	RO	L1D, RO	L1D, RO	L1D, RO	L1D, RO	L1D, RO
Read-Only Cache Size	48 KiB	24 KiB	48 KiB	32 KiB	32 KiB	28 KiB
Shared With[1]	TXT	L1D, TXT	L1D, TXT	L1D, TXT	L1D, TXT	L1D, TXT
Constant Cache Size	?	?	?	?	?	?
Shared Memory Size	48 KiB	64 KiB	96 KiB	96 KiB	64 KiB	164 KiB
L2 Cache Size	1.5 MiB	?	?	6 MiB	5.5 MiB	40 MiB
Main Memory Size	5 GiB	2 GiB	24 GiB	32 GiB	11 GiB	40 GiB

[1] L1D: L1 Data Cache - TXT: Texture Cache - RO: Read-Only Cache

Table 5.3.: Official Specifications for the Tested GPUs

5.3. Results

The results obtained by the tool are shown subsequently for each GPU in a separate table. Please note that not every piece of information is obtained via benchmarks. Some of the information is also fetched directly from the CUDA interface. Moreover, Nvidia has never officially shared the cache line sizes of the caches, however, the obtained results from the tool comply with the results from here [33] [34].

5.3.1. Nvidia Tesla K20c - Kepler

The Kepler SM-architecture can be viewed at Figure **2.2 Exemplary SM-internal architecture (Kepler)**. In summary, one SM of the Nvidia Tesla K20c GPU consists of 192 cores, 48 KiB for the Texture/Read-Only Cache, which is strictly separated from the 16 KiB L1 Data Cache. It also employs 48 KiB of shared memory, an L2 Cache of 1.5 MiB and overall 5 GiB of main memory. The 48 KiB of shared memory and the 16 KiB L1 Data Cache can also be configured to vice versa in terms of the memory since the shared memory and the L1 Data Cache share the same physical memory, yet strictly separated. The figure of the architecture also shows a *Uniform Cache*, which could be the Constant Cache, however, no information regarding this cache is available. The tool captured the following results:

	Size	**Line Size** (Bytes)	**Lat.** (cycles)	**Shared By**[1]	**#/SM**
L1 Data Cache	16.137 KiB	128	64	-	1
Texture Cache	12.012 KiB	32	111	RO	4
Read-Only Cache	12.129 KiB	32	130	TXT	4
Constant L1 Cache	1.828 KiB	64	50	-	4
Constant L1.5 Cache	32.813 KiB	256	120	-	-
Shared Memory	48 KiB	-	47	-	-
L2 Cache	1.25 MiB	32	223	-	-
Main Memory	4.633 GiB	-	339	-	-

[1] L1D: L1 Data Cache - TXT: Texture Cache - RO: Read-Only Cache

Table 5.4.: Tool Results for Nvidia Tesla K20c

As expected, the tool obtains approximately the correct size for the L1 Data Cache with about 16 KiB, a measured cache line size of 128 B and a latency of 64 cycles.

There exists only one L1 Data Cache in the SM. Interestingly, the Texture/Read-Only Cache seems to be split up into four fragments, as the measured size is 12 KiBand the measured number of caches per SM is four. This matches the expected total size of 48 KiB. The Texture/Read-Only Cache was measured to have a cache line size of 32 B and a latency of 111 or respectively 130 cycles. It is also worth mentioning, that this GPU works as proof, that the benchmarks, that check if L1 Data Cache, Texture Cache and/or Read-Only Cache are on the same physical cache, function as desired since the Kepler microarchitecture has a separate L1 Data Cache. This is detected correctly by the tool. The Constant Caches seem to be approximately 2 KiB or 32 KiB, respectively and have cache line sizes of 64 B or 256 B. Both were measured to have a latency of 50 and 120 cycles. In general, all obtained Constant Cache results coincide with the results from [34]. The rather high latency of the Texture Cache and Read-Only Cache is also remarkable since the size of the cache is relatively small. However, it can be observed later, that the Texture Cache often has an increased latency compared to the L1 Data Cache (See **5.3.7 Summary**). Furthermore, the shared memory with a size of 48 KiB seems to be the fastest memory component in terms of latency (47 cycles). Besides, the L2 Cache was measured to have a line size of 32 B and a latency of approximately 223. At last, the main memory showed a latency of 339 cycles. Interestingly, the size of the main memory, which is directly fetched from the CUDA interface, differs a bit from the officially shared size of 5 GiB. Since the Nvidia tool *deviceQuery* that is deployed with the CUDA SDK, also returns the size of 4.633 GiB, this seems to be a documentation error on the side of Nvidia.

5.3.2. Nvidia GeForce 840M - Maxwell

The Maxwell SM-architecture can be found at Figure **4.18 SM-internal architecture for Maxwell**. Overall, the SM of the Nvidia GeForce 840M has 128 cores and 64 KiB of shared memory. It further has two subdivided Texture Caches, which can also be used as L1 Data Caches. Unfortunately, there is no precise L2 Cache specification regarding the size of this GPU because Nvidia has already removed the corresponding whitepaper information from their resources. However, this information is fetched directly from the CUDA interface anyway. The tool captured these results:

	Size	**Line Size** (Bytes)	**Lat.** (cycles)	**Shared By**[1]	**#/SM**
L1 Data Cache	-	-	-	-	-
Texture Cache	12.129 KiB	32	92	RO	2
Read-Only Cache	12.129 KiB	32	92	TXT	2
Constant L1 Cache	2.109 KiB	64	33	-	1
Constant L1.5 Cache	30.469 KiB	256	86	-	-
Shared Memory	64 KiB	-	31	-	-
L2 Cache	1 MiB	32	173	-	-
Main Memory	1.958 GiB	-	448	-	-

[1] L1D: L1 Data Cache - TXT: Texture Cache - RO: Read-Only Cache

Table 5.5.: Tool Results for Nvidia GeForce 840M

As mentioned before, some older GPUs do not support the caching of global loads. The Nvidia GeForce 840M is an example of such a GPU. Thus, before the L1 Data Cache benchmarks start, the support of caching of global loads is evaluated and according to the result, the tool will ignore the benchmarks for this cache. The result for the Texture and Read-Only Cache is noteworthy: As described in the section **4.6 Number of Caches Per SM** the expected size of 24 KiB is split up into two caches of size 12 KiB. Consequently, the official shared size is consistent with the result. The Texture/Read-Only Cache shows a cache line size of 32 B again as well as a latency of 92 cycles. The Constant Caches did not really change much compared to the previous GPU apart from a lower latency of 33 (Constant L1) and 86 cycles (Constant L1.5). The shared memory is with a latency of 31 cycles the fastest memory segment again. The L2 Cache shows a latency of roughly 173 cycles. However, the main memory seems to have an increased latency of 448 compared to the last GPU, most likely due to the fact that the Nvidia Tesla K20c is a GPU for HPC, whereas the Nvidia GeForce 840M is not made for that purpose.

5.3.3. Nvidia Quadro P6000 - Pascal

The architecture of the Pascal SM is shown in the Appendix (See Figure **A.1 SM-architecture for Pascal**). The Nvidia Quadro P6000 features 64 or 128 cores per SM depending on the GPU. It also has a shared memory of 96 KiB and a total of 48 KiB for the L1 Data/Texture/Read-Only Cache. It is further able to use 24 GiB of main memory. This is the obtained result of the tool:

	Size	**Line Size** (Bytes)	**Lat.** (cycles)	**Shared By**[1]	**#/SM**
L1 Data Cache	24.012 KiB	32	92	TXT, RO	2
Texture Cache	24.012 KiB	32	85	L1D, RO	2
Read-Only Cache	24.012 KiB	32	94	L1D, TXT	2
Constant L1 Cache	2.141 KiB	64	33	-	1
Constant L1.5 Cache	31.25 KiB	256	94	-	-
Shared Memory	96 KiB	-	31	-	-
L2 Cache	3 MiB	32	243	-	-
Main Memory	23.877 GiB	-	412	-	-

[1] L1D: L1 Data Cache - TXT: Texture Cache - RO: Read-Only Cache

Table 5.6.: Tool Results for Nvidia Quadro P6000

The expected size of the L1 Data, Texture and Read-Only Cache (48 KiB) is again split up into two cache structures of the size of 24 KiB. This was not documented by Nvidia before. The line sizes and latency stay almost the same as well as the measured values for the Constant Caches. As before the shared memory has the lowest latency in comparison with the other memory components.

5.3.4. Nvidia Tesla V100 - Volta

The Volta architecture is illustrated in the Appendix (See Figure **A.2 SM-architecture for Volta**). The Nvidia Tesla V100 GPU has 64 cores, 96 KiB of shared memory and 32 KiB for the L1 Data/Texture/Read-Only Cache, though the shared memory and L1 Data/Texture/Read-Only Cache can be configured to be partitioned differently for a total of 128 KiB, e.g. 64 KiB of shared memory and 64 KiB for the L1 Data Cache. However, the tests were executed on default settings. Moreover, it can use a 6 MiB L2 Cache and a total of 32 GiB of main memory. The tool captures:

	Size	**Line Size** (Bytes)	**Lat.** (cycles)	**Shared By**[1]	**#/SM**
L1 Data Cache	25.629 KiB	32	36	TXT, RO	1
Texture Cache	24.023 KiB	32	80	L1D, RO	1
Read-Only Cache	25.629 KiB	32	36	L1D, TXT	1
Constant L1 Cache	2.078 KiB	64	32	-	1
Constant L1.5 Cache	> 62.5 KiB	256	105	-	-
Shared Memory	96 KiB	-	27	-	-
L2 Cache	6 MiB	64	215	-	-
Main Memory	31.748 GiB	-	430	-	-

[1] L1D: L1 Data Cache - TXT: Texture Cache - RO: Read-Only Cache

Table 5.7.: Tool Results for Nvidia Tesla V100

One issue of the benchmark can be observed in the results of Volta. The expected size of 32 KiB is not fully detected as there is a 6–8 KiB discrepancy. This behavior was first mentioned in [33]. The deviation was explained to be caused by a new cache replacement policy that was first applied for Volta, which is responsible for the cache misses starting too early. A decreased latency for L1 Data and Read-Only Cache can be observed, too. One other remarkable aspect is the obtained size of the Constant L1.5

Cache. Since CUDA only allows developers to use a maximum of 64 KiB of constant memory within one CUDA `.cu` module, the benchmark cannot exceed the size of 64 KiB. However, it can be concluded from the fine-grained result of each array access, that there is nevertheless an intermediate cache due to a loading time in between the Constant L1 loading time and an even larger L2 loading time. Since the CUDA module needs another 1.5 KiB of constant memory for something else in the program, the limit is decreased to 62.5 KiB. One other remarkable change can be seen for the L2 Cache. The cache line size changed from 32 B to 64 B for this generation.

5.3.5. Nvidia RTX 2080 Ti - Turing

The Turing architecture is depicted in the Appendix (See Figure **A.3 SM-architecture for Turing**). The Nvidia RTX 2080Ti utilizes 64 cores per SM, 64 KiB of shared memory and 32 KiB for the L1 Data/Texture/Read-Only Cache but they can be partitioned differently again for a total of 96 KiB. It further has access to a 5.5 MiB L2 Cache and altogether 11 GiB of main memory. This is what the tool measured:

	Size	**Line Size** (Bytes)	**Lat.** (cycles)	**Shared By**[1]	**#/SM**
L1 Data Cache	26.137 KiB	32	40	TXT, RO	1
Texture Cache	25.012 KiB	32	86	L1D, RO	1
Read-Only Cache	26.137 KiB	32	40	L1D, TXT	1
Constant L1 Cache	2.078 KiB	64	32	-	1
Constant L1.5 Cache	38–46 KiB	256	105	-	-
Shared Memory	64 KiB	-	30	-	-
L2 Cache	5.5 MiB	64	230	-	-
Main Memory	11 GiB	-	425	-	-

[1] L1D: L1 Data Cache - TXT: Texture Cache - RO: Read-Only Cache

Table 5.8.: Tool Results for Nvidia RTX 2080Ti

The Turing results still show the 6–8 KiB deviation. This was also measured by [34]. Moreover, the Constant L1.5 size result showed a slight variance on this device, as the size ranges between 38 and 46 KiB. It is unclear if the usage of the Windows OS and this deviation are only by coincidence on the same test environment or if there might be a causality since this is the only GPU of the tested GPUs, which showed this kind of variance for the Constant L1.5 Cache size. The shared memory showed the fastest latency again followed by the Constant L1 Cache and the L1 Data/Read-Only Cache.

5.3.6. Nvidia A-100 - Ampere

The Ampere SM-architecture is shown in the Appendix (See Figure **A.4 SM-architecture for Ampere**). The Nvidia A-100 features 64 cores as well as 164 KiB of shared memory and 28 KiB of L1 Data/Texture/Read-Only Cache; those can be partitioned differently as well for a total amount of 192 KiB. Furthermore, the GPU is able to use 40 MiB for the L2 Cache and 40 GiB of main memory. There even exists an Nvidia A-100 version with 80 GiB of main memory. The tool was able to capture the following information:

	Size	**Line Size** (Bytes)	**Lat.** (cycles)	**Shared By**[1]	**#/SM**
L1 Data Cache	22.254 KiB	32	39	TXT, RO	1
Texture Cache	21.191 KiB	32	89	L1D, RO	1
Read-Only Cache	22.254 KiB	32	38	L1D, TXT	1
Constant L1 Cache	2.141 KiB	64	34	-	1
Constant L1.5 Cache	> 62.5 KiB	256	114	-	-
Shared Memory	164 KiB	-	28	-	-
L2 Cache	40 MiB	64	223	-	-
Main Memory	39.409 KiB	-	567	-	-

[1] L1D: L1 Data Cache - TXT: Texture Cache - RO: Read-Only Cache

Table 5.9.: Tool Results for Nvidia A-100

Again the Ampere result has the same 6–8 KiB deviation for the L1 Data, Texture and Read-Only Cache. Furthermore, the Constant L1.5 Cache seems to be larger than the developer limit of 64 KiB once more. The latency of the main memory is also noteworthy since it is fairly increased due to the really huge memory size of 40 GiB. The shared memory seems to be permanently the fastest memory component in terms of the loading time/latency.

5.3.7. Summary

The higher latency of the Texture Cache for many GPUs is rather conspicuous. There are some potential reasons for that: First function call overhead produced by `tex1Dfetch<int>()` can be excluded since the same latency result is obtained by using the inline PTX assembler texture load instruction. However, the PTX assembler equivalent to the `tex1Dfetch<int>()`-function can still be a reason for that (See code at Figure **5.2 PTX Texture Load Instruction**): The texture load instruction is able to effectively load a

```
asm volatile(
    "tex.1d.v4.u32.s32 {%0, %1, %2, %3}, [tex_ref, {%4}];"
    : "=r"(values.x), "=r"(values.y), "=r"(values.z),
    "=r"(values.w) : "r"(j)
);
```

Figure 5.2.: Texture Load Instruction in PTX

value while specifying four coordinate values in one loading process. This might be at cost of the latency due to the address calculation. Another potential reason for the increased latency is the Texture Unit, which every texture load needs to pass. The Texture Unit inherently executes specific filtering techniques and is also able to apply low-precision interpolation [28, Ch. J].

Overall the results look very promising in terms of correctness. Other than the constant 6–8 KiB deviation for some GPUs no other issues were encountered. Moreover, in addition to Maxwell, information about the split/separated caches on Kepler and Pascal was revealed as well. The tool is able to automatically obtain information about the sizes, line sizes and latency of the caches as well as the number of partitions for some of the caches. It is able to return if the L1 Data Cache, the Texture Cache, and The Read-Only Cache are located at the same physical cache structure. It can also be observed by the results, that the structure of the SMs seems to be unified or standardized in the past years, probably in order to keep compatibility and simplicity over more generations. This can be inspected by checking the results of the rather older architectures compared to the more recent generations. The number of partitions for the L1 Data, Texture and Read-Only Cache varied on the older generations and is always one by now. Kepler has separated L1 Data from Texture and Read-Only Cache. These are combined into one cache since Maxwell. The older generations sometimes were not able to cache global loads like the tested Maxwell GPU, all newer generations are always able to cache global loads.

6. Potential Extensions

This chapter hints possible extensions or new benchmarks that could be added as new features to the implementation, respectively.

6.1. Bandwidth

Since latency is only one of the two core information aspects for memory/data movements, the other information, namely the bandwidth, could be introduced into the tool as a new feature in form of an additional benchmark. A few years ago, the bandwidth of Turing and Volta was measured with a microbenchmark similar to the implemented benchmarks [33] [34]. Again, it is therefore necessary to load the values of an array into the appropriate cache in a first round. Then similar to the latency benchmark of the tool implementation, the array is completely loaded and the required time for this process is measured. Afterwards, the throughput rate of the process in bytes per cycle can be calculated by dividing the array size with the required overall loading time. To get as close to the theoretical bandwidth of the corresponding cache or memory, two additional aspects are important. First, the loading process must be repeated with all available data type sizes for the load instructions (8-bit, 16-bit, 32-bit, 64-bit). The second important aspect is a varying number of threads and thread blocks, as the peak of bandwidth is most likely reached at a specific constellation of threads and thread blocks, which all need to be tested. Thus, every or many combinations of load instruction size and number of threads and thread blocks need to be passed, to get as close to the real bandwidth of each cache and memory as possible. A very similar benchmark was also applied here [10] for the throughput of global and shared memory.

6.2. Instruction Cache

One of the caches that were completely left out by the implementation, is the instruction cache or the layers of instruction caches, respectively. The implementation would benefit from additional benchmarks for this purpose. However, there is only few literature about measuring the instruction cache sizes of GPUs. These research papers [33] [34] contain a pretty well-explained approach for microbenchmarking the instruction cache.

The fundamental idea is similar to the P-Chase benchmark, though, it is unnecessary to execute load instructions on arrays for the instruction cache but to execute instruction sequences of varying size. As with our implementation, it starts with an instruction sequence of small size and executes it in a first round to fill it in the instruction caches and to avoid cold cache misses. Then it executes the sequence in a second round while measuring the required time to do so. Most likely, the fine-grained approach would not work for this case, since it would use the instruction caches, which would tamper with the results. It might be necessary to utilize preprocessor techniques in order to generate instructions of varying sizes, since it may be not possible to get instruction sequences of arbitrary size during runtime, although PTX is JIT-compiled according to the documentation of the Nvidia compiler nvcc [58]. It is required to evaluate further on that to get a more detailed understanding of the possibilities.

6.3. New Nvidia microarchitectures & Additional features

At the present time of this thesis, the new microarchitecture generations were announced by Nvidia, Ada Lovelace and Hopper. Nvidia has already shared the overview graphical sheets, hence it can already be estimated whether the implementation will still work with the new architectures.

6.3.1. Ada

The Ada Lovelace microarchitecture shows huge similarity with its predecessor Ampere (See Figure **A.5 SM-architecture for Ada Lovelace** in Appendix). Therefore, it is justifiable to assume, most of the benchmarks will still work on this microarchitecture. Nonetheless, it is remarkable, that this new microarchitecture is the first (with Hopper) whose main CUDA cores are partially only able to do 32-bit floating point operations and no integer arithmetic. While still specifying 128 CUDA cores per SM officially, it is relatively unclear how this works internally. According to the Ada architecture whitepaper [59], 16 CUDA cores of one warp are explicitly dedicated for processing FP32 operations in one warp while 16 are able to do both FP32 and INT32 operations. However, the benchmarks should not be influenced by this change in a way, which interferes with correct functionality.

6.3.2. Hopper

The Hopper microarchitecture shows the same change concerning CUDA cores as the Ada Lovelace architecture. Moreover, while the microarchitecture still has many similarities with the current architectures like Ampere or Turing, this new Hopper architecture seems to have a so-called *Tensor Memory Accelerator*, which is able to "efficiently transfer large blocks of data between global memory and shared memory" [60]. Since this new feature is mainly related to the usage of tensor cores and tensor computation, plus Nvidia describes it as a feature "to help feed the powerful new H100 Tensor Cores", there should not be any interference with the current implementation. However, it could be worth checking with additional benchmarks how the latency and bandwidth change with and without using the Tensor Memory Accelerator, which can be utilized by special `TMA` copy operations. In addition, this microarchitecture is the first to extend the current CUDA model of threads and thread blocks, as it adds another layer of thread block clusters. A thread block cluster is a group of thread blocks that are definitely "scheduled concurrently onto a group of SMs" [60]. The idea is to be able to cooperatively compute results over more than one SM. For that purpose dedicated lanes between all SMs are implemented into the microarchitecture, which allows an SM to access the shared memory of other SMs within the thread block cluster. While this new feature will not influence the current implementation, it could be worth adding an additional benchmark to measure the latency of loading shared memory data from/between other SMs.

6.4. Potential Port to AMD devices

As AMD GPUs are incompatible with Nvidia CUDA code, the tool works only for Nvidia devices. Thus, it is required to evaluate if the same microbenchmarking approach could be used for AMD devices as well. This looks promising since there exists already research for microbenchmarking other features by using OpenCL [61], although they did not use it for the cache or memory sizes. Furthermore, there is already some research for microbenchmarking a specific AMD microarchitecture (GCN) by using the P-Chase method for the caches of the AMD GPU [36]. This looks very encouraging, as they used roughly the same approach without the automatic extraction of the information. It is noteworthy, that they used the corresponding AMD programming interface to CUDA, which is called Heterogeneous Interface for Portability (HIP), which has the advantage to be able to run both on Nvidia and AMD GPUs. Interestingly, there are tools to support the porting process from CUDA to HIP [62] by automatically translating CUDA code to HIP. However, this will most likely not work completely for the tool implementation, since it relies also on inline PTX assembly code. Nevertheless, it can be useful for porting the tool to HIP. AMD and its open-source contributors also provide a helpful table for the CUDA syntax and the corresponding HIP syntax[1]. It can be questioned if a direct one-to-one port

[1]https://github.com/ROCm-Developer-Tools/HIP/blob/master/docs/markdown/hip_terms.md

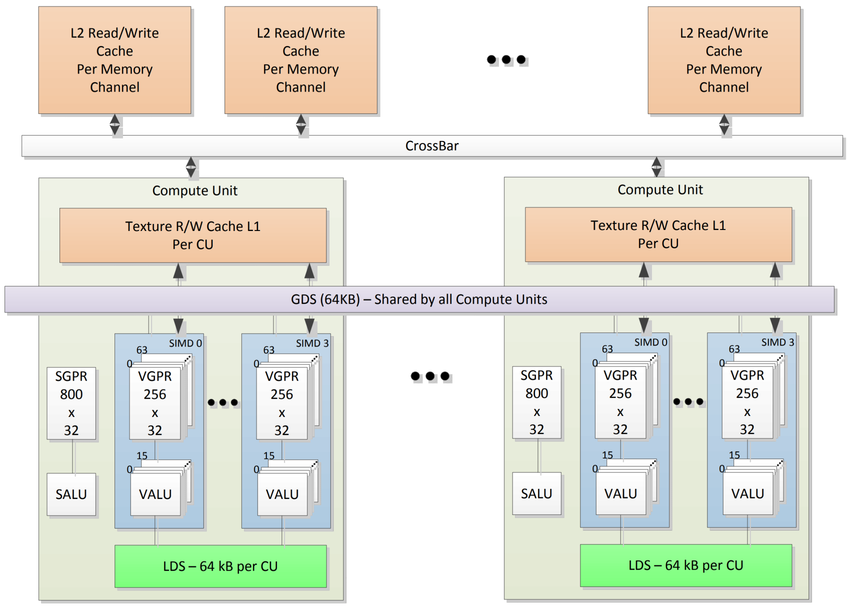
L2 Read/Write Cache Per Memory Channel
L2 Read/Write Cache Per Memory Channel
L2 Read/Write Cache Per Memory Channel
CrossBar
Compute Unit
Texture R/W Cache L1 Per CU
Compute Unit
Texture R/W Cache L1 Per CU
GDS (64KB) – Shared by all Compute Units
SIMD 0
SIMD 3
SGPR 800 x 32
VGPR 256 x 32
SALU
VALU
LDS – 64 kB per CU

Figure 6.1.: AMD GCN Compute Unit Architecture (Simplified) [63]

can be conducted, since the microarchitecture of AMD GPUs is a bit different from Nvidia microarchitectures. For the illustration of the AMD GCN architecture (See Figure **6.1 AMD GCN Compute Unit Architecture**) of the SM-equivalent for AMD (Compute Unit) it can be observed, there is also an L1 Data Cache/Texture Cache on the "Compute Unit"-level as well as most likely a "Shared Memory"-equivalent, the "Load Data Share"-memory (LDS). Moreover, it has an additional "Global Data Share"-memory shared by all Compute Units. It is unclear if this architecture also contains a Constant Cache. Overall it is likely, that the same approaches for Nvidia can be used likewise for AMD and the corresponding caches, though the implementation will probably differ plus there may be additional - or less - caches or memories in one Compute Unit.

7. Related Work

7.1. Microbenchmarking

Microbenchmarking has been used quite often in the past to benchmark specific GPUs for various microarchitectural features. In [11] microbenchmarking was used on the Nvidia GT200 GPU to measure the latency of specific instructions like `add`, `mul`, `div` or more expensive mathematical operations like `sinf()`, `logf()` and `expf()`. They also applied benchmarks to evaluate which execution unit types these instructions require. As an example, their benchmark revealed, the `add` instruction only uses a single precision execution unit (either FP32 or INT32 core), whereas the `mul` floating point instruction uses a single precision unit plus a special function unit. The same paper also gives insights into benchmarks for the branch divergence behavior, the memory translation via Translation Lookaside Buffer (TLB) for which the presence of two levels of TLBs was presented. Their research also covered approaches for the instruction cache. More about instruction caches can be found in chapter **6.2 Instruction Cache**. The research paper further described a benchmark and the result for checking if the Constant and Instructions Caches share the same space. Benchmarks for the TLB and instruction cache(s) are also covered by [33] and [34]. Both of those references share additional insights into how Nvidia controls the execution of warps as well as how instructions are encoded. Furthermore, they benchmarked characteristics of the tensor cores that were added to the SM since Volta. Eventually, the Fine-Grained P-Chase benchmark was first used and applied for the GPU by [10] to benchmark the sizes of the caches as well as the latency and the throughput of some memories. In addition, the research describes benchmarks to check how the shared memory is structured as it is divided into memory banks which can produce bank conflicts with loads.

7.2. Performance Analysis Model for GPU Architectures

This paper [64] describes another type of microbenchmarking, which takes also a given program into account. It first applies microbenchmarks to model the GPU behavior in terms of metrics like the instruction pipeline, shared memory access and global memory access. Then with a given program, the performance bottleneck(s) of the program are investigated to give the developer of a program hints for possible optimizations. This is done via a simulator program *Barra* that is able to simulate/emulate the execution of CUDA applications. More information about this simulator can be found at [65]. The identified issues/bottlenecks are then propagated to the developers, e.g. low computational density, insufficient parallel warps, insufficient parallelism, etc. It is further described how the gained information can be used to optimize a program by reference to the optimization process of a typical GPU function, a specific type of matrix multiplication called *Dense Matrix Multiply*.

7.3. OpenCL Microbenchmark Suite

Interestingly, one previous work [61] created a benchmark suite in OpenCL to measure device characteristics. OpenCL is an open-source framework to write and execute code on various types of devices (CPU, GPU, FPGA, etc.). As an example, this work applies benchmarks for the performance of specific mathematical instructions, the bus bandwidth between the host device and compute device or memory bandwidth. Since the benchmark suite is written for OpenCL, it could be compatible with both Nvidia and AMD GPUs. Moreover, the properties of a CPU can be benchmarked with OpenCL as well.

7.4. Discovering Microarchitectural Features Via GPU ISA Encoding Solver

A complex approach to the exposure of microarchitectural characteristics is described in [66]. The authors of this work developed their own Nvidia GPU ISA encoding solver by using the encoded binary and the `cuobjdump`-generated disassembly sass file as input. Afterwards, they implemented their own version of an assembler for the Kepler GPUs. The assembler translates every instruction and adds an ELF header at the beginning in order to obtain an executable CUDA binary file (cubin). The microbenchmarks are then implemented in sass code. The microbenchmarks cover the exploration of register banks and bank conflicts, the meaning of control codes in terms of *"single-issue"*- or *"dual-issue"*-handling instructions. Further, they microbenchmarked the maximum arithmetic throughput of the GPU and the throughput of shared and global memory. Later they applied the obtained knowledge to optimize an important method for GPU computation, the single precision general matrix multiply (SGEMM). They managed to create an optimized SGEMM version, which is overall 15% faster than the official Nvidia implementation provided by the cuBLAS API.

7.5. NVML (Related tool)

Nvidia provides an additional interface to query and monitor various other information about the GPU that does not correspond to the memory topology but could be helpful nevertheless. This is called Nvidia Management Library (NVML). With the latest CUDA version (CUDA 11.8) one of the following NVML query functions may be interesting for the tool in the future:

- `nvmlDeviceGetArchitecture`: Returns integer identifying the microarchitecture of the GPU
- `nvmlDeviceGetCurrPcieLinkWidth`: Returns the link width of the PCIe in usage
- `nvmlDeviceGetFanSpeed`: Returns the current fan speed in percentage
- `nvmlDeviceGetGpuOperationMode`: Returns the operating mode (computing mode, graphics application mode)
- `nvmlDeviceGetMaxClockInfo`: Gives back the maximum possible clock speed
- `nvmlDeviceGetMemoryBusWidth`: Gets memory bus width
- `nvmlDeviceGetNumGpuCores`: Gets the total number of CUDA cores in the GPU, at some point this could be used for the number of cores
- `nvmlDeviceGetPcieThroughput`: Can be used to measure the PCIe throughput within a 20ms interval

8. Conclusion

This thesis presented the design and implementation of a tool to capture the memory topology of an Nvidia GPU. This tool was designed to provide great support for GPU programmers and developers, especially but not solely in the region of High-Performance Computing in order to further modify and optimize their GPU programs according to the memory topology of the GPUs. The proposed design and workflow start with an efficient binary-wise search followed by a sequential search for the cache size while consistently employing the Fine-Grained P-Chase benchmark. The obtained data is then used by the described Change Point Detection to automatically extract the sizes of each cache. In addition, other implemented benchmarks are able to return the latency and cache line size of all caches and memory components. Further benchmarks retrieve additional information like the number of L1 Data, Texture and Read-Only Caches per Streaming Multiprocessor or whether these caches share the same physical space. The implementation was severely tested on numerous GPUs; every microarchitecture currently supported by the CUDA SDK was examined. The obtained results were overall consistent with the officially shared specifications of the GPUs, although the last few GPUs showed a constant 6–8 KiB discrepancy to the published L1 Data, Texture and Read-Only Cache sizes. The results for Kepler and Pascal also revealed a couple of subdivided caches within the SM, which were not clearly documented by Nvidia. Further improvements in terms of performance were proposed and highlighted. Additional benchmarks for even more information like bandwidth and instruction cache data were suggested and described, too. First input towards a possible port to AMD devices was provided at the end of the thesis. While new information and additional properties could be added as new benchmarks to the implementation as well, the created tool and the obtained results look quite promising. The tool might already be used for a better understanding of the GPU memory topology and for further optimizing CUDA code in favor of a more efficient cache and memory usage.

A. Appendix

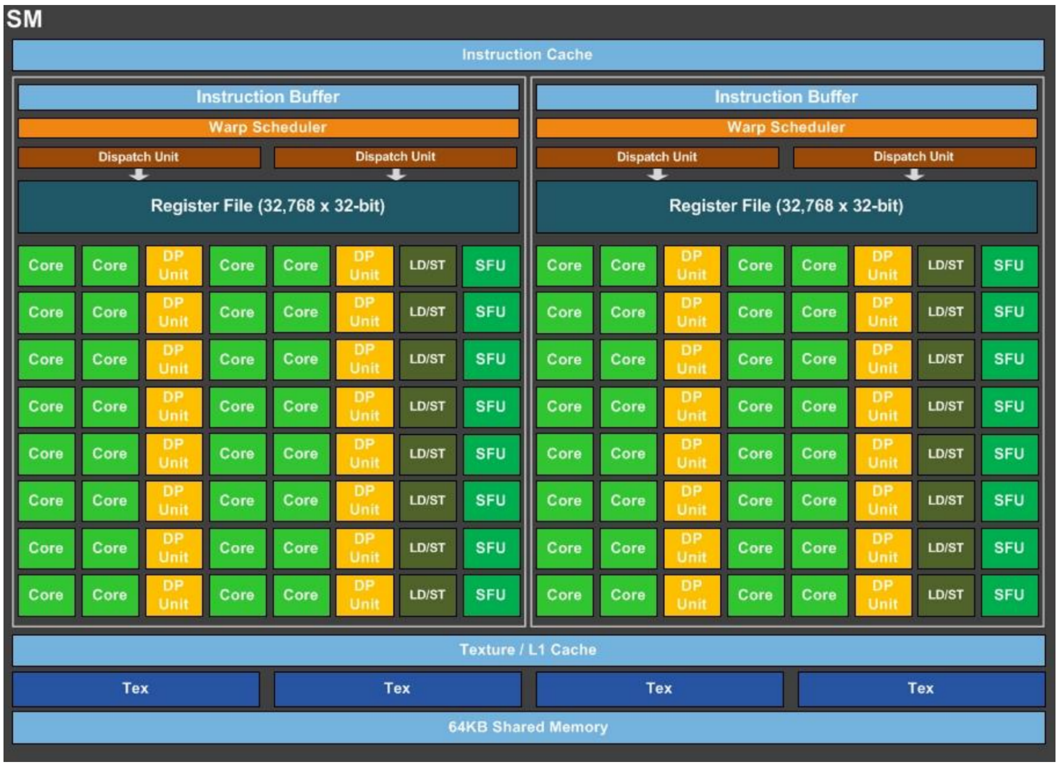

Figure A.1.: SM-architecture for Pascal [67]

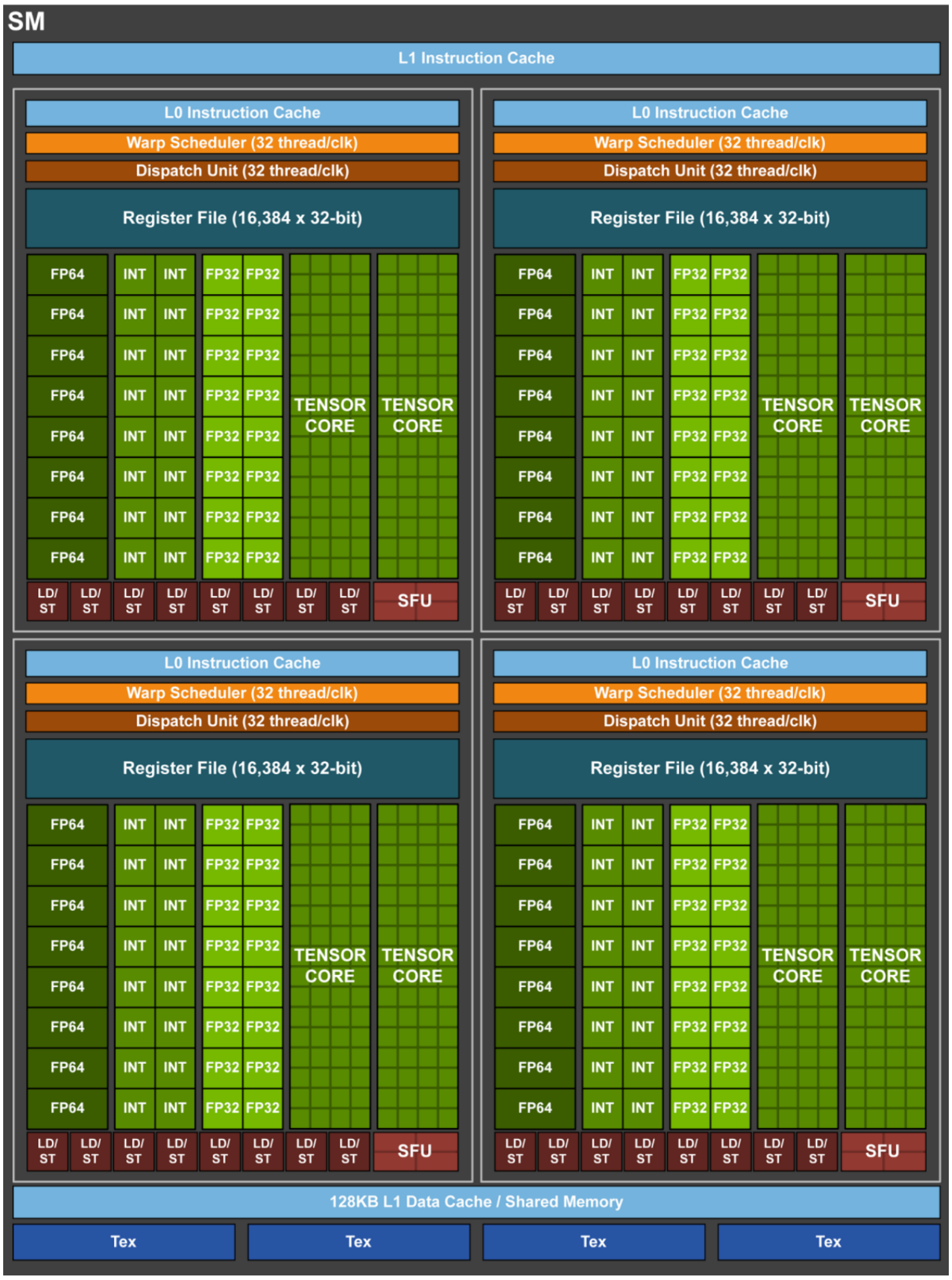
SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ ST
SFU
128KB L1 Data Cache / Shared Memory
Tex

Figure A.2.: SM-architecture for Volta [53]

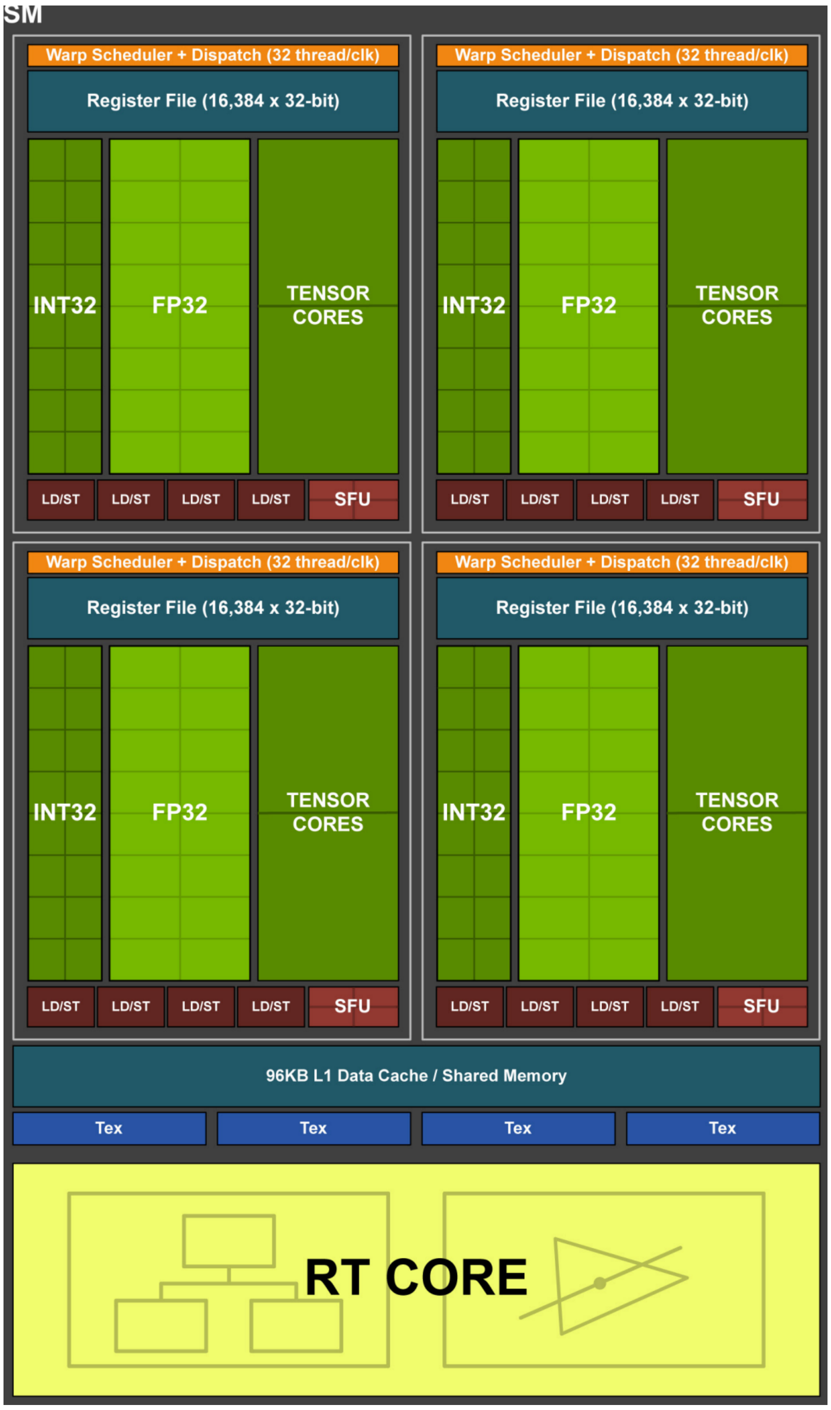
SM
Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
TENSOR CORES
LD/ST
SFU
96KB L1 Data Cache / Shared Memory
Tex
RT CORE

Figure A.3.: SM-architecture for Turing [55]

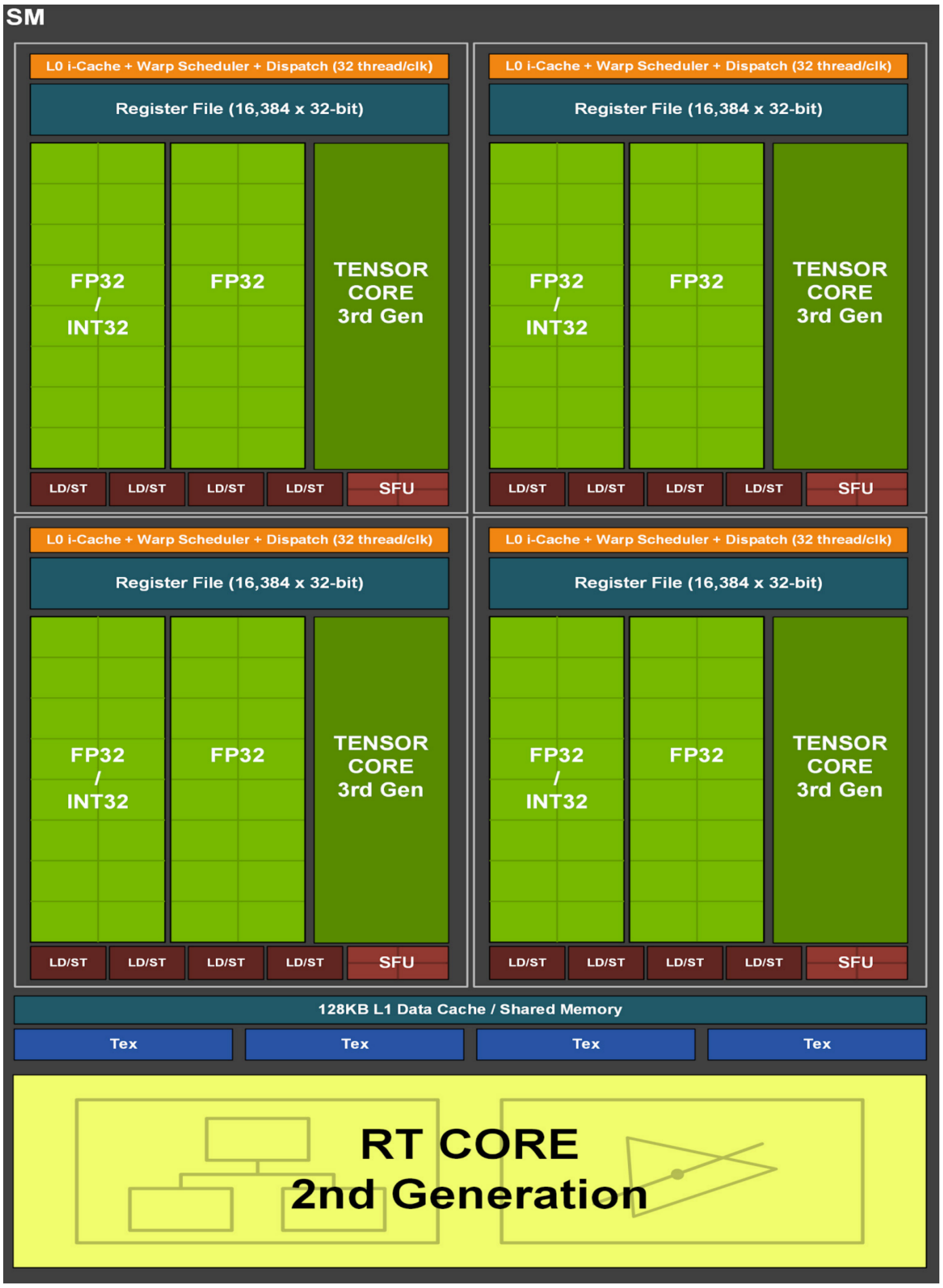
SM
L0 i-Cache + Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
FP32 / INT32
FP32
TENSOR CORE 3rd Gen
LD/ST
SFU
128KB L1 Data Cache / Shared Memory
Tex
RT CORE 2nd Generation

Figure A.4.: SM-architecture for Ampere [56]

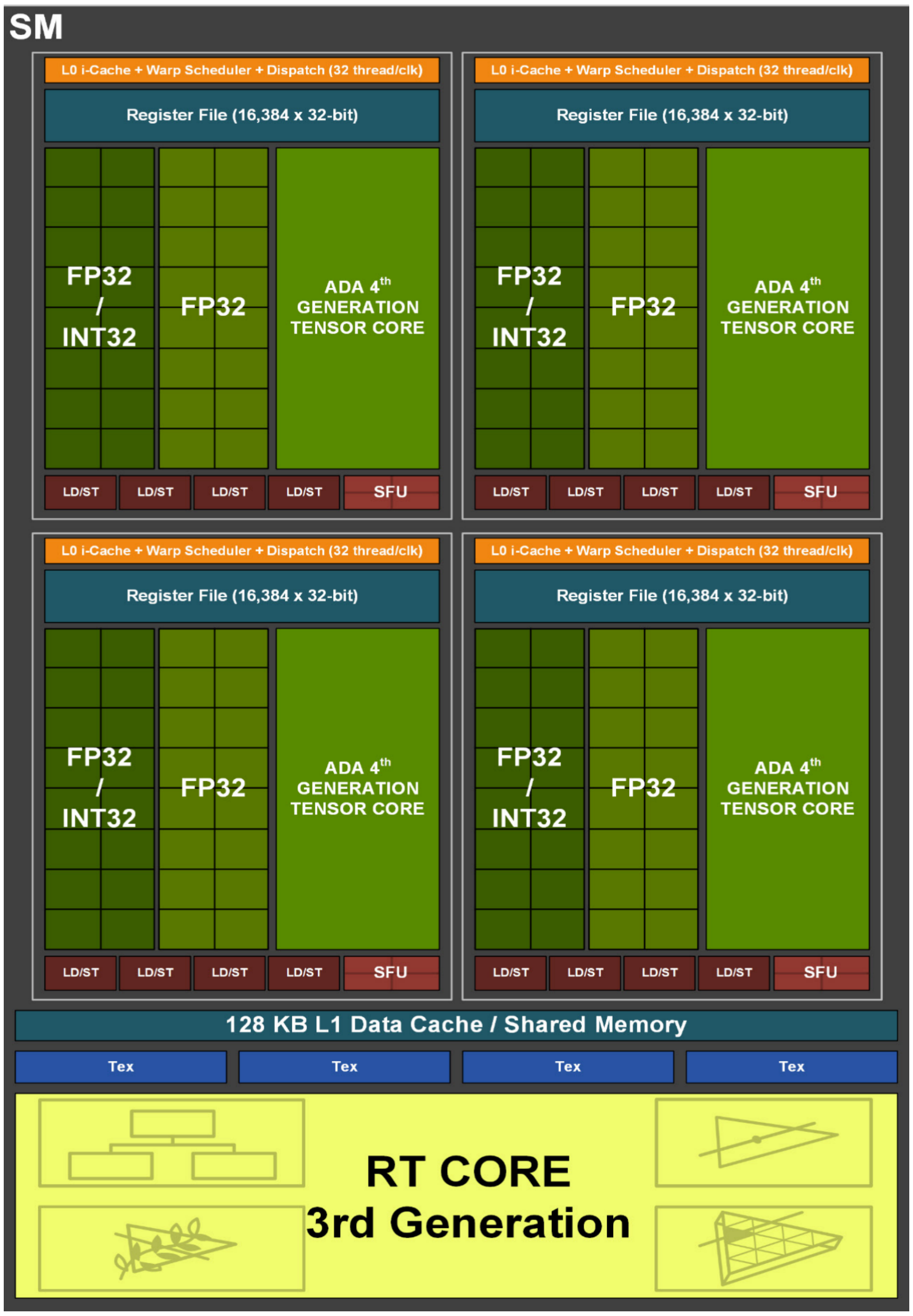
SM
L0 i-Cache + Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
FP32 / INT32
FP32
ADA 4th GENERATION TENSOR CORE
LD/ST
SFU
128 KB L1 Data Cache / Shared Memory
Tex
RT CORE 3rd Generation

Figure A.5.: SM-architecture for Ada Lovelace [59]

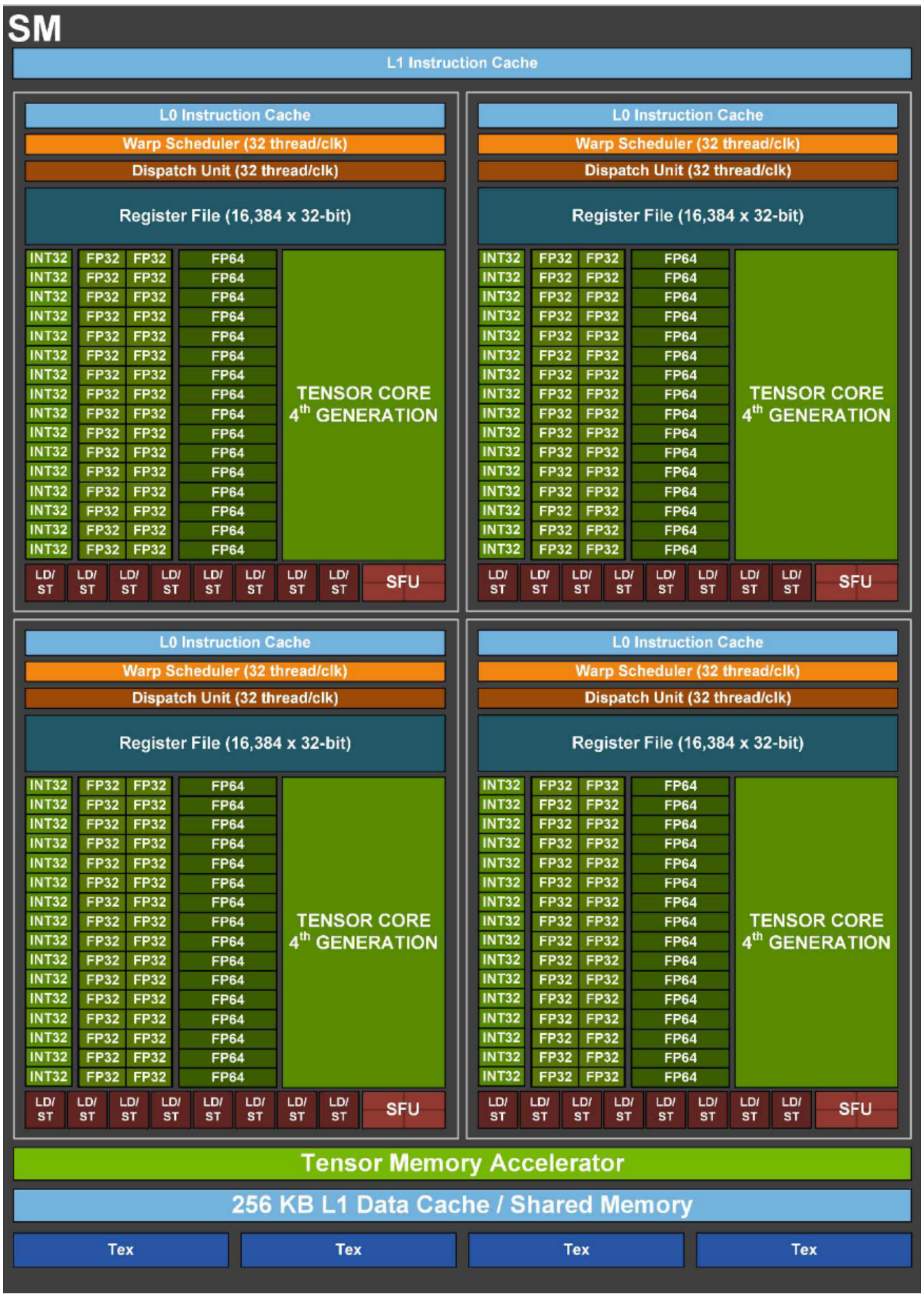
SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE 4th GENERATION
LD/ ST
SFU
Tensor Memory Accelerator
256 KB L1 Data Cache / Shared Memory
Tex

Figure A.6.: SM-architecture for Hopper [60]

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] TOP500. *June 2022 TOP500*. 2022. URL: `https://www.top500.org/lists/top500/2022/06/` (visited on 08/10/2022).

[2] A. Snell and L. Segervall. *HPC APPLICATION SUPPORT FOR GPU COMPUTING*. 2017. URL: `https://www.nvidia.com/content/intersect-360-HPC-application-support.pdf` (visited on 08/22/2022).

[3] FSU RESEARCH COMPUTING CENTER INFORMATION TECHNOLOGY SERVICES. *HPC Benchmarks*. 2022. URL: `https://rcc.fsu.edu/docs/hpc-benchmarks` (visited on 08/10/2022).

[4] Y. Kawano, F. Valdez, and O. Castillo. "Performance Evaluation of Optimization Algorithms based on GPU using CUDA Architecture." In: *2018 IEEE Latin American Conference on Computational Intelligence (LA-CCI)*. 2018, pp. 1–6. DOI: `10.1109/LA-CCI.2018.8625236`.

[5] G. Rokos. "Accelerating Optimisation-Based Anisotropic Mesh Adaptation using nVIDIA's CUDA Architecture." In: (Sept. 2022).

[6] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco. "GPUs and the Future of Parallel Computing." In: *IEEE Micro* 31.5 (2011), pp. 7–17. DOI: `10.1109/MM.2011.89`.

[7] S. Yeluri. 2022. URL: `https://community.juniper.net/blogs/sharada-yeluri/2022/08/22/tearing-down-memory-wall` (visited on 09/06/2022).

[8] T. Gruber, J. Eitzinger, G. Hager, and G. Wellein. *LIKWID*. 2022. URL: `https://github.com/RRZE-HPC/likwid`.

[9] X. Mei, K. Zhao, C. Liu, and X. Chu. "Benchmarking the Memory Hierarchy of Modern GPUs." In: Sept. 2014.

[10] X. Mei and X. Chu. "Dissecting GPU Memory Hierarchy Through Microbenchmarking." In: *IEEE Transactions on Parallel and Distributed Systems* 28.1 (2017), pp. 72–86. DOI: `10.1109/TPDS.2016.2549523`.

[11] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. "Demystifying GPU microarchitecture through microbenchmarking." In: *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. 2010, pp. 235–246. DOI: 10.1109/ISPASS.2010.5452013.

[12] A. Khan. "Brief Overview of Cache Memory." In: Apr. 2020. DOI: 10.13140/RG.2.2.22359.21921.

[13] A. N. SLOSS, D. SYMES, and C. WRIGHT. "CHAPTER 12 - CACHES." In: *ARM System Developer's Guide*. Ed. by A. N. SLOSS, D. SYMES, and C. WRIGHT. The Morgan Kaufmann Series in Computer Architecture and Design. Burlington: Morgan Kaufmann, 2004, pp. 402–459. DOI: https://doi.org/10.1016/B978-155860874-0/50013-7. URL: https://www.sciencedirect.com/science/article/pii/B9781558608740500137.

[14] A. J. Smith. "Cache Memory." In: *Encyclopedia of Computer Science*. GBR: John Wiley and Sons Ltd., 2003, pp. 180–187. ISBN: 0470864125.

[15] A. Abayomi, A. Olukayode, and G. Olakunle. "An Overview of Cache Memory in Memory Management." In: *Automation, Control and Intelligent Systems* 8 (Jan. 2020), p. 24. DOI: 10.11648/j.acis.20200803.11.

[16] B. Jacob, S. Ng, and D. Wang. *Memory Systems: Cache, DRAM, Disk*. Jan. 2008. ISBN: 978-0-12-379751-3.

[17] A. Tatourian. *NVIDIA GPU ARCHITECTURE & CUDA PROGRAMMING ENVIRONMENT*. 2013. URL: https://tatourian.blog/2013/09/03/nvidia-gpu-architecture-cuda-programming-environment/ (visited on 09/20/2022).

[18] M. Peker, B. Sen, and H. Guruler. "Rapid Automated Classification of Anesthetic Depth Levels using GPU Based Parallelization of Neural Networks." In: *Journal of medical systems* 39 (Feb. 2015), p. 197. DOI: 10.1007/s10916-015-0197-3.

[19] C. Wittenbrink, E. Kilgariff, and A. Prabhu. "Fermi GF100 GPU architecture." In: *IEEE Micro* 31 (Mar. 2011), pp. 50–59. DOI: 10.1109/MM.2011.24.

[20] Nvidia. *NVIDIA's Next Generation CUDA^{TM} Compute Architecture: Kepler TM GK110/210*. 2014. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf (visited on 09/20/2022).

[21] Nvidia. *NVIDIA GeForce GTX 680 Technology Overview*. 2012. URL: https://www.nvidia.com/content/PDF/product-specifications/GeForce_GTX_680_Whitepaper_FINAL.pdf (visited on 09/17/2022).

[22] D. Koppelman. *Nvidia GPU Microarchitecture*. 2020. URL: https://www.ece.lsu.edu/koppel/gp/notes/set-nv-org.pdf (visited on 09/20/2022).

[23] Nvidia. *Nvidia Kepler Tuning Guide*. 2022. URL: `https://docs.nvidia.com/cuda/kepler-tuning-guide/index.html` (visited on 09/20/2022).

[24] Nvidia. *Parallel Thread Execution ISA Version 7.7*. 2022. URL: `https://docs.nvidia.com/cuda/parallel-thread-execution/index.html` (visited on 09/22/2022).

[25] Nvidia. *Memory Statistics - Texture*. 2015. URL: `https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/memorystatisticstexture.htm` (visited on 09/20/2022).

[26] M. Doggett. "Texture Caches." English. In: *IEEE Micro* 32.3 (2012), pp. 136–141. ISSN: 0272-1732. DOI: `10.1109/MM.2012.44`.

[27] Nvidia. *CUDA C++ Best Practices Guide*. 2022. URL: `https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html` (visited on 09/30/2022).

[28] Nvidia. *CUDA 11.7.1 C++ Programming Guide*. 2012. URL: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html` (visited on 09/20/2022).

[29] M. Harris. *How to Access Global Memory Efficiently in CUDA C/C++ Kernels*. 2013. URL: `https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/` (visited on 09/20/2022).

[30] Nvidia. *PTX and SASS Assembly Debugging*. 2015. URL: `https://docs.nvidia.com/gameworks/content/developertools/desktop/ptx_sass_assembly_debugging.htm` (visited on 09/22/2022).

[31] Nvidia and W. Hoenig. *View Assembly Code Correlation in Nsight Visual Studio Edition*. 2013. URL: `https://developer.nvidia.com/blog/cuda-pro-tip-view-assembly-code-correlation-nsight-visual-studio-edition/` (visited on 09/26/2022).

[32] Nvidia. *Inline PTX Assembly in CUDA*. 2022. URL: `https://docs.nvidia.com/cuda/inline-ptx-assembly/index.html` (visited on 09/28/2022).

[33] Z. Jia, M. Maggioni, B. Staiger, and D. Scarpazza. "Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking." In: (Apr. 2018).

[34] Z. Jia, M. Maggioni, J. Smith, and D. Scarpazza. "Dissecting the NVidia Turing T4 GPU via Microbenchmarking." In: (Mar. 2019).

[35] M.-M. Papadopoulou, M. Sadooghi-Alvandi, and H. Wong. "Micro-benchmarking the GT 200 GPU." In: 2009.

[36] X. Zhang and E. Shcherbakov. "DELTA: Validate GPU Memory Profiling with Microbenchmarks." In: Sept. 2020, pp. 97–104. DOI: `10.1145/3422575.3422784`.

[37] A. Lung-Yut-Fong, C. Lévy-Leduc, and O. Cappe. "Robust changepoint detection based on multivariate rank statistics." In: June 2011, pp. 3608–3611. DOI: `10.1109/ICASSP.2011.5946259`.

[38] S. Aminikhanghahi and D. Cook. "A Survey of Methods for Time Series Change Point Detection." In: *Knowledge and Information Systems* 51 (May 2017). DOI: `10.1007/s10115-016-0987-z`.

[39] C. Lévy-Leduc and F. Roueff. "Detection and localization of change-points in high-dimensional network traffic data." In: *The Annals of Applied Statistics* 3 (Aug. 2009). DOI: `10.1214/08-AOAS232`.

[40] T. Grundy, R. Killick, and G. Mihaylov. "High-dimensional changepoint detection via a geometrically inspired mapping." In: *Statistics and Computing* 30.4 (Mar. 2020), pp. 1155–1166. DOI: `10.1007/s11222-020-09940-y`. URL: `https://doi.org/10.1007%5C%2Fs11222-020-09940-y`.

[41] C. Truong, L. Oudre, and N. Vayatis. "A review of change point detection methods." In: Jan. 2018.

[42] R. Killick, P. Fearnhead, and I. Eckley. "Optimal Detection of Changepoints With a Linear Computational Cost." In: *Journal of the American Statistical Association* 107 (Dec. 2012), pp. 1590–1598. DOI: `10.1080/01621459.2012.737745`.

[43] S. Rabanser, S. Günnemann, and Z. Lipton. "Failing Loudly: An Empirical Study of Methods for Detecting Dataset Shift." In: Oct. 2018.

[44] M. Hollander, D. A. Wolfe, and E. Chicken. *Nonparametric Statistical Methods, 3rd Edition*. July 2015. ISBN: ISBN: 978-1-119-19603-7.

[45] J. W. Pratt and J. D. Gibbons. "Kolmogorov-Smirnov Two-Sample Tests." In: *Concepts of Nonparametric Theory*. New York, NY: Springer New York, 1981, pp. 318–344. ISBN: 978-1-4612-5931-2. DOI: `10.1007/978-1-4612-5931-2_7`. URL: `https://doi.org/10.1007/978-1-4612-5931-2_7`.

[46] R. Wilcox. *Introduction to Robust Estimation and Hypothesis Testing. 5th Edition*. Nov. 2021. ISBN: ISBN: 978-0-12-820098-8.

[47] Nvidia. *Nvidia CUDA Samples*. 2022. URL: `https://github.com/NVIDIA/cuda-samples`.

[48] M. Clark. *Kepler Texture Objects Improve Performance and Flexibility*. 2013. URL: `https://developer.nvidia.com/blog/cuda-pro-tip-kepler-texture-objects-improve-performance-and-flexibility/` (visited on 09/30/2022).

[49] Chips and Cheese. *GPU Memory Latency's Impact, and Updated Test*. 2021. URL: `https://chipsandcheese.com/2021/05/13/gpu-memory-latencys-impact-and-updated-test/` (visited on 09/30/2022).

[50] Nvidia. *NVIDIA GeForce GTX 750 Ti Whitepaper*. 2014. URL: http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce-GTX-750-Ti-Whitepaper.pdf (visited on 10/06/2022).

[51] Nvidia. *Instruction Statistics*. 2015. URL: https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/instructionstatistics.htm (visited on 10/03/2022).

[52] Nvidia. *NVIDIA QUADRO P6000*. 2018. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/productspage/quadro/quadro-desktop/quadro-pascal-p6000-data-sheet-a4-nv-704590-r1.pdf (visited on 10/10/2022).

[53] Nvidia. *NVIDIA TESLA V100 GPU ARCHITECTURE*. 2017. URL: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf (visited on 10/10/2022).

[54] Nvidia. *NVIDIA V100 Tensore Core GPU*. 2020. URL: https://images.nvidia.com/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf (visited on 10/10/2022).

[55] Nvidia. *NVIDIA TURING GPU ARCHITECTURE*. 2018. URL: https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf (visited on 10/10/2022).

[56] Nvidia. *NVIDIA AMPERE GA102 GPU ARCHITECTURE*. 2021. URL: https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.1.pdf (visited on 10/10/2022).

[57] Nvidia. *NVIDIA Ampere Architecture In-Depth*. 2020. URL: https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/ (visited on 10/10/2022).

[58] Nvidia. *NVIDIA CUDA Compiler Driver NVCC*. 2022. URL: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html (visited on 10/06/2022).

[59] Nvidia. *NVIDIA ADA GPU MICROARCHITECTURE*. 2022. URL: https://images.nvidia.com/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf (visited on 10/03/2022).

[60] Nvidia. *NVIDIA H100 Tensor Core GPU Architecture*. 2022. URL: https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper (visited on 10/06/2022).

[61] X. Yan, X. Shi, and Q. Sun. "An OpenCL Micro-Benchmark Suite for GPUs and CPUs." In: *2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies*. 2012, pp. 53–58. DOI: `10.1109/PDCAT.2012.52`.

[62] AMD ROCm. *HIPIFY*. 2022. URL: `https://github.com/ROCm-Developer-Tools/HIPIFY` (visited on 10/07/2022).

[63] AMD. *AMD Graphics Core Next Architecture, Generation 3 Reference Guide*. 2016. URL: `https://developer.amd.com/wordpress/media/2013/12/AMD_GCN3_Instruction_Set_Architecture_rev1.1.pdf` (visited on 10/08/2022).

[64] Y. Zhang and J. D. Owens. "A quantitative performance analysis model for GPU architectures." In: *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. 2011, pp. 382–393. DOI: `10.1109/HPCA.2011.5749745`.

[65] 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*. 2010, pp. 351–360. DOI: `10.1109/MASCOTS.2010.43`.

[66] X. Zhang, G. Tan, S. Xue, J. Li, K. Zhou, and M. Chen. "Understanding the GPU Microarchitecture to Achieve Bare-Metal Performance Tuning." In: Jan. 2017, pp. 31–43. DOI: `10.1145/3018743.3018755`.

[67] Nvidia. *NVIDIA Tesla P100 Whitepaper*. 2016. URL: `https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf` (visited on 10/10/2022).