SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Visual Presentation of GPUscout GPU Bottleneck Analysis

Tobias Stuckenberger

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Visual Presentation of GPUscout GPU Bottleneck Analysis

Visuelle Präsentation der GPUscout GPU Bottleneck Analyse

Author:	Tobias Stuckenberger
Examiner:	Prof. Dr. Martin Schulz
Supervisor:	M.Sc. Stepan Vanecek
Submission Date:	15.02.2025

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

Munich, 15.02.2025 Tobias Stuckenberger

Acknowledgments

I would like to thank my family for supporting me not only during this thesis but my entire academic journey. Secondly, I would like to thank my advisor, Stepan Vanecek, for his continued and invaluable feedback and support throughout this thesis. Lastly, I want to thank the Chair of Computer Architecture and Parallel Systems (CAPS) at the Technical University of Munich (TUM) for providing access to hardware resources essential to testing my thesis work.

Abstract

The introduction of specialized hardware components and features alongside newer GPU architectures presents developers with growing challenges in writing optimized and efficient code. Analysis tools, such as GPUscout, aim to solve this problem by providing insights into the utilization of specific memory systems, as well as making concrete recommendations for optimization potential. In this thesis, a graphical user interface (GUI) for the GPUscout application is developed to replace its current text-based output, with the goal of improving accessibility and offering a more intuitive way to present results. Additional functionality like a code viewer, as well as the ability to easily compare two different results, further improve the overall workflow and effectiveness of using the tool. After documenting the design and development of the GPUscout-GUI application, three projects are evaluated and visualized using both tools to demonstrate the accuracy of the information presented in the interface, as well as its additional value, with a focus on their influence on the decision-making process of prospective users and the effect of proposed implementation changes.

Contents

1. Introduction

In recent years, Graphics Processing Units (GPUs) have transformed from dedicated hardware primarily for rendering images into essential tools for high-performance computing across various fields. Initially designed to enhance graphics in video games and visual applications, GPUs have demonstrated exceptional parallel processing capabilities, making them well-suited for diverse computational tasks. Their efficiency in handling large-scale data operations far surpasses that of traditional CPUs, leading to widespread use in areas such as scientific simulations, cryptocurrency mining, and, most notably, artificial intelligence (AI) [1]. Especially the recent rise of large language models (LLMs) such as OpenAI's ChatGPT has dramatically increased the demand for faster and more efficient computation as these models grow in size and complexity.

Over the past decade, GPU architecture has evolved significantly, with continuous improvements driven by core architectural enhancements and the integration of specialized hardware, for example, to accelerate AI tasks. In addition to these hardware advancements, optimizing the source code of applications to maximize GPU performance has become an increasingly important area of research [2].

Although AMD and NVIDIA are the two dominant GPU manufacturers, NVIDIA has established itself as the leading supplier of data-center GPUs. Originally known for its consumer and professional graphics cards, NVIDIA revolutionized the industry in 2006 with the introduction of the CUDA (Compute Unified Device Architecture) framework, which enabled developers to harness the massive parallel processing power of NVIDIA GPUs for general-purpose computing, paving the way for advancements in scientific simulations, deep learning, and large-scale data processing. Since then, successive GPU generations have brought significant improvements in performance, efficiency, and specialized hardware features. However, as the architecture of graphics cards becomes more complex, with additional features being constantly implemented, programmers face greater challenges in ensuring efficient software performance. Failing to optimize applications for the latest GPU architecture can result in sub-optimal utilization of expensive computing resources, hindering scalability and limiting overall system efficiency. Because of this, several tools have already been created to aid developers in finding optimization potential in the source code of CUDA applications more easily, one of which is the GPUscout application detailed in this thesis.

1.1. GPUscout

Developers seeking deeper insights into the execution of CUDA kernels have access to a variety of tools, ranging from official solutions like NVIDIA's NSight Compute [3] to third-

party options like Scalasa [4], HPCToolkit [5] and the TAU toolkit [6]. These tools provide a wide range of capabilities, from measuring valuable metrics and statistics during execution to offering deep insights into certain memory operations and thread utilization. While this information is highly useful for evaluating the efficiency of CUDA kernels, their learning curve can be steep, with a strong understanding of the underlying mechanisms typically being required to interpret collected data effectively, leaving especially inexperienced users, who would benefit most from these tools, to analyze and draw conclusions on their own.

GPUscout, which is an application developed at the Technical University of Munich, takes a unique approach to identifying memory-related bottlenecks and making optimization recommendations more accessible, particularly for less experienced users. Instead of merely presenting raw performance data, GPUscout suggests concrete optimizations to improve efficiency based on three approaches: static analysis of an intermediary representation of the source code, warp stall sampling, as well as collecting performance metrics [7].

The tool is implemented modularly, with each currently available analysis focusing on different areas of a GPU's memory system, such as recommending the use of texture or shared memory instead of global memory. By detecting certain patterns in the intermediary kernel code that indicate performance bottlenecks or inefficient uses of a certain feature, GPUscout provides users with actionable suggestions and relevant details to help pinpoint and resolve these problems more easily. While this approach already enhances accessibility compared to similar tools, the user experience could be further improved by implementing an accompanying graphical user interface to replace the current textual output.

1.2. Motivation

Presenting information in a clear and comprehensible manner is crucial, especially when aiming to make large data sets accessible to a wider audience. Unlike raw text output, which can be overwhelming and difficult to interpret, well-designed interfaces allow for dynamic exploration of the presented results, offering a more intuitive and user-friendly experience. This is particularly important when handling various types of data, where clearly displaying relationships between data points is essential. Furthermore, a GUI helps reduce the cognitive load a large amount of information imposes by simplifying and highlighting the most relevant information in an accessible format, improving decision-making and overall productivity.

This thesis aims to create a companion application for GPUscout that displays the generated results graphically, allowing for a more detailed presentation of the information in an accessible and easier to understand manner. While the output of GPUscout already provides explanations for certain statements and metrics, inexperienced users may still struggle to fully comprehend and implement the suggested changes, emphasizing the need for an alternative front end.

1.3. Outline

The following chapter introduces relevant concepts of the NVIDIA GPU architecture, as well as CUDA programming, and provides an overview of the GPUscout application and the different analyses it implements. Chapter 3 focuses on user interface design, containing some guidelines, as well as insights into the layout of related projects. The design and development process and choices of GPUscout-GUI are described in Chapter 4, with the leading sections 4.1 and 4.2 focusing on the technologies used to create the application, the general architecture, as well as the internal handling and representation of the input data, followed by detailed descriptions of the different interfaces and components created. The accuracy and usability improvements of GPUscout-GUI are evaluated in Chapter 5, where three different CUDA projects are analyzed using GPUscout. The textual output is then compared to the information presented by the newly developed GUI while emphasizing how GPUscout-GUI enhances the process of identifying bottlenecks in a more intuitive way than before, along with the added value of visualizations and metrics. Additionally, the implemented changes based on these insights and their impact on the performance of the analyzed binaries are discussed. Lastly, Chapter 6 presents some topics and potential for future research, followed by Chapter 7 concluding the thesis.

2. Background

2.1. NVIDIA GPU Architecture

Since launching its first GPU in 1998, NVIDIA has continuously introduced architectural changes to enhance the performance of its GPU line-up, with the Tesla architecture released in 2006 marking a pivotal moment by shifting from separate vertex and pixel-fragment processors to a unified shader model, significantly improving computational power. Subsequent architectures, Fermi, Kepler, Maxwell, and ultimately Pascal, brought further improvements until the groundbreaking Turing architecture. Turing introduced the first generation of Tensor Cores, delivering substantial advancements in neural network training alongside general performance enhancements. This evolution continued with the Ampere and Hopper architectures, leading to the anticipated Blackwell architecture in 2024 featuring 5th generation Tensor Cores [8, 9, 10, 11, 12, 13].

2.1.1. Streaming Multiprocessors

The essential building blocks of NVIDIA GPUs are Streaming Multiprocessors (SMs), which are capable of handling a variety of workloads. Using the Single Instruction Multiple Threads (SIMT) execution model, concurrent and independent execution of multiple threads is made possible, which allows for better utilization of available resources. In the latest NVIDIA Hopper architecture, each SM consists of four individual processing blocks, also called sub-partitions, sharing the SMs unified L1 cache and shared memory, as well as additional memory sections like texture memory only available in newer architectures.

Each SM sub-partition contains many different execution units supporting various operations, most of which are responsible for integer (INT32) or single- and double-precision floating point operations (FP32/FP64). Other execution units include tensor cores specially designed for AI workloads, load/store units (LD/ST) for handling memory operations, and special function units (SFU) for various other complex mathematical operations like sine or cosine calculations.

In a GPU, threads are not managed individually but hierarchically, each thread being part of a block (also known as Cooperative Thread Array or CTA), which is further organized into grids [14, 15]. Threads are also part of warps of up to 32 threads, each thread in a warp executing the same instruction simultaneously. To maximize the performance of each sub-partition, scheduling warps as effectively as possible is crucial. Based on execution policies like round-robin or oldest-first, a warp scheduler selects warps, which are then assigned instructions and dispatched to execution units by the dispatch unit, with frequently access instructions stored in an instruction cache to improve latency [16]. Regarding memory, each

SM sub-partition contains thread-private registers, with access to the SM's L1 cache, shared and texture memory, as well as an additional higher-level instruction cache. In the case of the latest Hopper architecture, each SM also includes a Tensor Memory Accelerator, a hardware unit designed to reduce the overhead of asynchronous data transfers of multidimensional tensors between global and shared memory, improving performance for AI and scientific workloads [10, 17, 15, 1, 18].

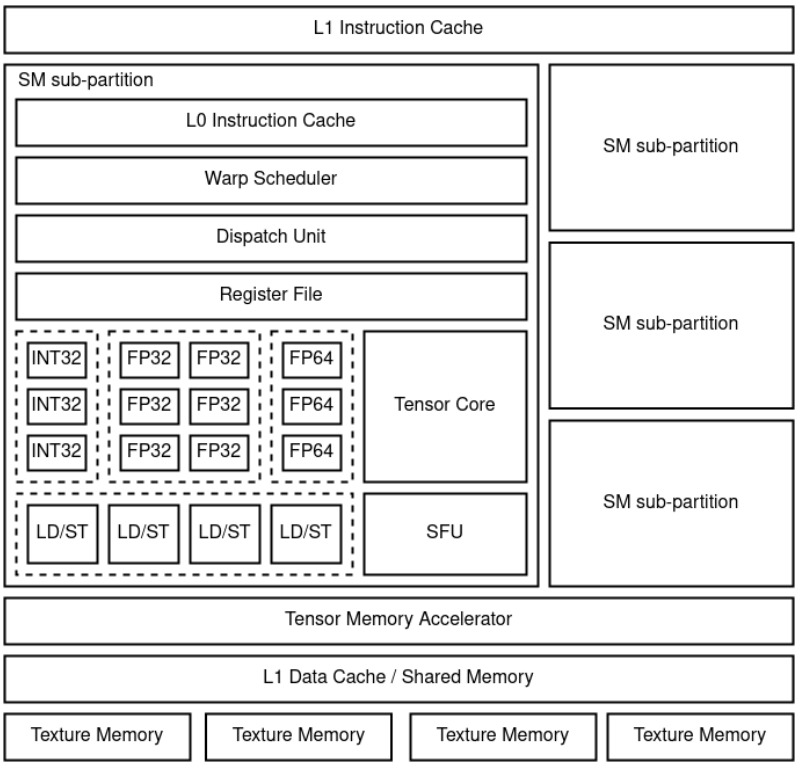


Figure 2.1.: NVIDIA Hopper SM architecture [10]

2.1.2. Memory Model

An efficient memory model is crucial to fully leverage the immense parallel processing capabilities of modern GPUs compared to CPUs, with GPU memory categorized into two main types: on-chip memory, which offers the lowest latency and is integrated within each streaming multiprocessor, and off-chip memory, providing larger storage capacity at the tradeoff of higher latency. On-chip memory includes registers, L1 and L2 caches, shared memory, and constant caches. In contrast, off-chip memory primarily consists of dynamic random-access memory (DRAM) encompassing texture and constant memory.

Register

As the fastest-growing on-chip memory structure in recent generations of NVIDIA GPUs, the register file has expanded to accommodate a significant portion of the available on-chip

memory area while also contributing substantially to power consumption. Register files are a part of every SM sub-partition and are organized in 32 banks, corresponding to the number of threads in a warp, and offer thread-private storage. This leads to the capacity of a register file being a limiting factor for the maximum number of concurrently running threads, as each thread needs to be assigned a section of the register file [19]. In the current NVIDIA Hopper architecture, the register file of each SM contains 65536 registers (256kB), with each thread assigned up to 255 registers depending on the total amount of threads used. To compensate for a shortage of registers, the compiler inserts spill and fill instructions, which transfer data to and from local memory instead, although negatively impacting performance in some cases [10, 20].

L1 Cache

The L1 cache in GPUs is hardware-managed and operates transparently to developers. It efficiently handles a high volume of concurrent requests from individual cores within a streaming multiprocessor, as well as requests from global, shared, texture, and surface memory. Unlike CPU L1 cache lines, which are narrower, GPU L1 cache lines are 128 bytes wide, allowing all 32 threads in a warp to access memory simultaneously when accessing neighboring addresses, enabling significant performance improvements. A request to the L1 cache results in either a cache hit or a cache miss, depending on whether the required cache line is already available in the L1 cache. The detailed process of handling memory requests can be seen in figure 2.2.

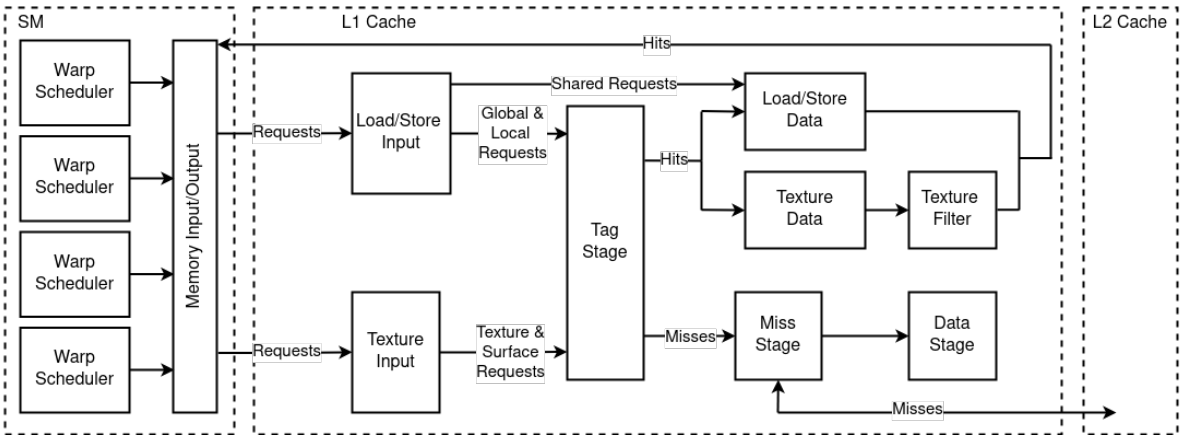


Figure 2.2.: L1 cache model [15]

Requests received by the L1 cache are not processed individually but classified and organized into sectors of 32 bytes before proceeding depending on the availability of the data in the cache. When a cache miss occurs, the miss stage retrieves the required data from a higher-level memory structure before passing the result to the data stage. In case of a hit, the requested data can be directly returned from the load/store or texture data unit without additional fetches being necessary [15].

Shared Memory

Shared memory is available per thread block and can be accessed by all threads inside it. Since the release of the NVIDIA Volta architecture, shared memory and the L1 cache have been merged into a single memory unit, with the latest Hopper architecture including 256kB of combined capacity per SM. The share between shared memory and the L1 cache can either be configured manually or is determined dynamically, allowing for more flexibility depending on the workload. The main difference between shared memory and the L1 cache now lies in the ability to control its content, with the L1 cache being managed automatically. Shared memory is internally organized into 32 banks, which can be simultaneously accessed, resulting in much higher bandwidth when reading or writing addresses of distinct banks [21, 10, 15, 22, 1].

Constant Memory

Constant memory is part of device memory and stores read-only variables. Each SM also contains dedicated constant caches shared among all threads within a thread block, providing read access speeds similar to those of registers. Because of this, constant memory is especially useful for broadcasting constants to other threads and should be used whenever possible [1].

L2 Cache

The L2 cache is another on-chip memory component that speeds up data movements between SMs and the main memory (DRAM). Although slower, the larger L2 cache still provides fast access to frequently used data and is, for example, used to look up cache misses in the L1 cache. While the L1 cache is unique to each SM, the L2 cache is shared among all SMs, with the L2 cache being organized in multiple banks, each corresponding to a different memory partition, similarly to the L1 cache or shared memory. In both caches, so-called miss status holding registers (MSHRs) are used to record and redirect missed requests to a higher memory level. The detailed structure of an L2 cache can be seen in figure 2.3 [16].

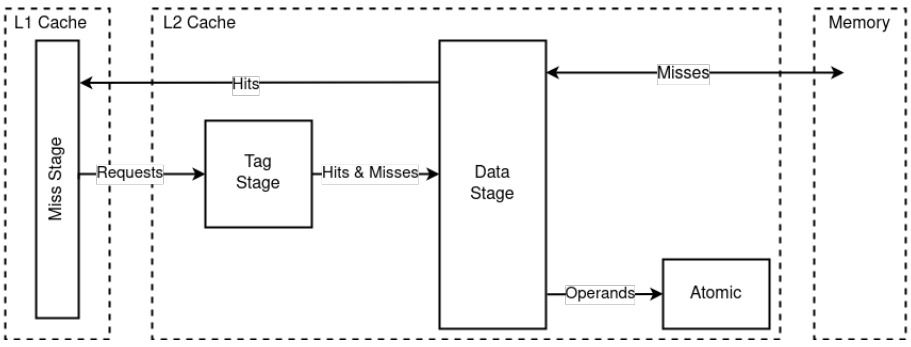


Figure 2.3.: L2 cache model [15]

The architecture of the L2 cache can be seen as a simpler version of the L1 cache. Similarly,

requests are first processed by the tag stage, where they are classified and accumulated before being redirected to the data stage. Depending on whether the sector is a hit or miss, the requested data is directly returned or first fetched from DRAM, with a special atomic unit (AOU) processing atomic operations directly. While the specifics of this process are not public, [23] assumes atomic operations to be generated in shader cores before the L2 bank acquires the relevant data and performs the operation [15, 16].

DRAM

Global memory, or DRAM, represents the majority of available memory on a GPU and is accessed through both the L1 and L2 caches. Requests between an SM and global memory are passed through a memory bus and handled by a memory controller, which schedules the requests and optimizes access patterns for optimal bandwidth utilization and minimal latency [15].

2.2. NVIDIA GPU Programming

Together with the Tesla architecture, NVIDIA released the Compute Unified Device Architecture (CUDA) programming model in 2006, enabling programmers to take advantage of the superior computing power of general-purpose GPUs. The CUDA API allows interaction with the GPU by extending the C/C++ language using additional keywords and constructs, with the most important new concepts being function execution space specifiers, which are used to define the context that functions are executed and callable from [1]. The following specifiers are available to be used:

`__global__` The `__global__` keyword is used to mark functions as CUDA kernels, indicating they will be executed on the device while still being callable from the host. In newer versions, kernels are also callable directly from the device.

`__device__` The `__device__` keyword specifies functions executed on and callable from the device and cannot be used together with the `__global__` specifier.

`__host__` For functions only operating on the host, the `__host__` specifier can be used and is also implicitly added to any function not marked with any other space modifier.

To be able to execute source files with CUDA directives, the standard C compiler can no longer be used. NVIDIA provides a special compiler called NVCC (NVIDIA CUDA Compiler), which separates host and device code to allow further compilation of the host code using conventional C compilers.

Compilation of the GPU code is achieved using an intermediate representation of the kernels, called Parallel Thread Execution (PTX). NVIDIA describes PTX as "a low-level parallel thread execution virtual machine and instruction set architecture (ISA), [which] exposes the GPU as a data-parallel computing device" [24]. The use of this concept of virtual architectures is crucial for the resulting binary to be compatible with future GPU architectures

and can be seen in the complete compilation process in figure 2.4. Two other concepts that are involved in ensuring compatibility not only with the current but also future generations of GPUs are just-in-time (JIT) compilation, as well as the use of fatbinaries. When using JIT compilation, the assembly of PTX code is delayed to application runtime by using previously mentioned virtual code architectures, another approach being fatbinaries, which can contain multiple translations of GPU kernels, with the most applicable one being selected at runtime [25, 24].

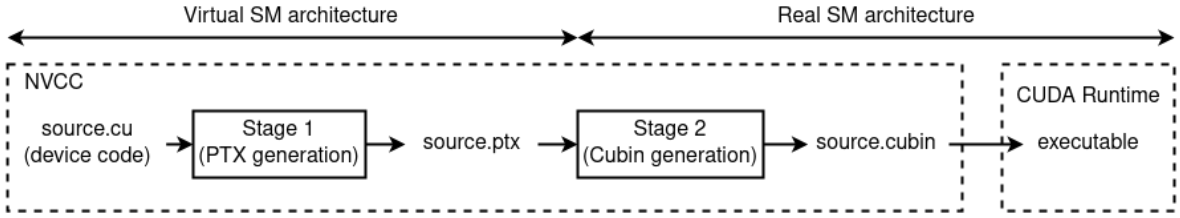


Figure 2.4.: CUDA compilation steps [25]

The CUDA kernels defined in the source file are compiled using proprietary NVIDIA compilers. In the first step, the source code is compiled into CUDA binary (.cubin) and/or PTX (.ptx) intermediary representations before being placed in a fatbinary. The host portion is then compiled and embedded into the fatbinary in a second preprocessing step, which also transforms all CUDA-specific code into standard C/C++ constructs, enabling the standard C compiler to compile the host code along with the embedded fatbinary into an object file (.obj). A simple CUDA kernel can be seen in listing 2.1, which will be an ongoing example throughout this chapter.

```
1 __global__ void axpy(int n, float a, float *x, float *y) {
2   int i = blockIdx.x*blockDim.x + threadIdx.x;
3   if (i < n) {
4     y[i] = a*x[i] + y[i];
5   }
6 }
```

Listing 2.1: Simple CUDA kernel: Vector addition

Marked by the `__global__` keyword, this kernel performs the $y = a \cdot x + y$ operation for the vectors x and y and the constant a. In addition to these three parameters, a fourth parameter, n, denotes the length of the vectors and is used as a safeguard to prevent the kernel from accessing memory out of the bounds of the vectors. The index of the current kernel is first computed based on the number and dimension of thread blocks, as well as the position of the index inside them, with users having to manually ensure the total number of threads corresponds to or exceeds the length of the input vectors. Each thread then computes a single element of the y vector based on its index, parallelizing the computation instead of iteratively updating each element sequentially.

2.2.1. PTX

Since a GPU can execute many threads in parallel, offloading compute-intensive and data-parallel tasks to the GPU is logical. As described in previous sections, this is achieved using kernel functions, which are compiled into the PTX instruction set and then translated to the target GPU instruction set at runtime. PTX provides both a stable programming model and a virtual machine model, which will be further described in this section.

Programming Model

PTX aims to provide a stable instruction set architecture (ISA), which can span multiple GPU generations while offering near-native performance. The core functionality of the PTX programming model lies in its focus on parallelism, with PTX instructions specifying the execution of concrete threads inside a parallel thread array. An array of threads executing the same kernel concurrently or in parallel is called a Cooperative Thread Array (CTA), with threads inside a CTA able to communicate at specified communication points, which will block threads until all threads have reached this point. Threads inside a CTA execute in single-instruction, multiple-thread (SIMT) fashion and are organized in batches, also called warps (typically of size 32), which execute the same instructions simultaneously. For communication between CTAs, shared memory can be used for CTAs of the same cluster, with cluster-wide barriers being able to be used to synchronize all threads within such a cluster. Because there exists a maximum amount of threads per CTA and CTAs per cluster, clusters of CTAs can be further organized into grids, allowing for even larger single kernel invocations. However, this has the disadvantage of lesser communication, as communication between CTAs of different clusters is limited.

An overview of the different memory sections available to threads depending on their configuration can be seen in figure 2.5. At the most basic level, individual threads have access to their own private registers and local memory. Within a CTA, shared memory enables data transfers between all included threads, with threads inside a CTA not only having access to the shared memory of their respective CTA but also to the shared memory sections of other CTAs inside their cluster, allowing for easier communication inside a cluster of CTAs. At the highest level, the only means of data exchange and communication between threads of different clusters is global memory, which is shared among all GPU kernels [24].

Machine Model

Similar to the previously presented model of SMs, in the PTX machine model, each multiprocessor is assigned a set of local registers per processor, shared memory shared between all processors within an SM, as well as read-only constant and texture caches shared by all processors, speeding up reads from constant and texture memory. With the introduction of the Volta architecture, NVIDIA also introduced the concept of independent thread scheduling, allowing the GPU to maintain a separate execution state for each thread instead of sharing it among all threads in a warp. This allows for better utilization of execution resources and

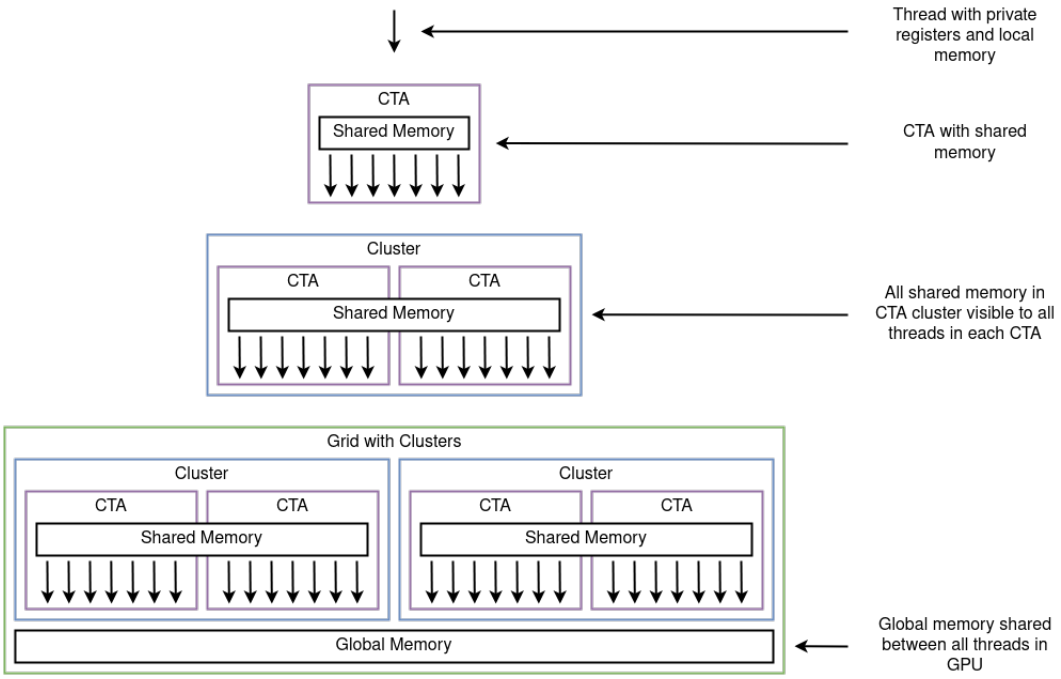


Figure 2.5.: PTX memory hierarchy [24]

enables the scheduler to switch between active parts of a warp instead of only scheduling complete warps, reducing general wait times [24, 26].

Syntax

Since PTX is designed to accommodate multiple GPU architectures, various abstractions are necessary, particularly concerning memory resources. PTX handles these abstractions through different state spaces and data types, a state space representing a storage area containing variables defined by attributes such as size, addressability, access speed, access rights, and the degree of sharing among threads. The following table lists all the state spaces available:

Keyword	Name	Sharing	Access
.reg	Register	Per-thread	R/W
.sreg	Special registers	Per-CTA	RO
.const	Constant memory	Per-grid	RO
.global	Global memory	Context	R/W
.local	Local memory	Per-thread	R/W
.param[1]	Parameters	Per-thread	R/W
.shared	Shared Memory	Per-cluster	R/W
.tex	Texture Memory	Context	RO

Table 2.1.: PTX state spaces [24]

The fundamental data types in PTX correspond to native data types supported by the target architecture, which are stored in registers and instructions are executed on. These include signed and unsigned integers (.s / .u), floating-point (.f), bits (.b), and predicate (.pred). The compiled PTX of the simple CUDA kernel introduced earlier can be seen in listing 2.2.

```
.visible .entry _Z5saxpyifPfS_(
    .param .u32 _Z5saxpyifPfS__param_0,
    .param .f32 _Z5saxpyifPfS__param_1,
    .param .u64 _Z5saxpyifPfS__param_2,
    .param .u64 _Z5saxpyifPfS__param_3
)
{
    .reg .pred  %p<2>;
    .reg .f32  %f<5>;
    .reg .b32  %r<6>;
    .reg .b64  %rd<8>;

    ld.param.u32  %r2, [_Z5saxpyifPfS__param_0];
    ld.param.f32  %f1, [_Z5saxpyifPfS__param_1];
    ld.param.u64  %rd1, [_Z5saxpyifPfS__param_2];
    ld.param.u64  %rd2, [_Z5saxpyifPfS__param_3];
    mov.u32  %r3, %ctaid.x;
    mov.u32  %r4, %ntid.x;
    mov.u32  %r5, %tid.x;
    mad.lo.s32  %r1, %r3, %r4, %r5;
    setp.ge.s32  %p1, %r1, %r2;
    @%p1 bra  $L__BB0_2;

    cvta.to.global.u64  %rd3, %rd2;
    cvta.to.global.u64  %rd4, %rd1;
    mul.wide.s32  %rd5, %r1, 4;
    add.s64  %rd6, %rd4, %rd5;
    ld.global.f32  %f2, [%rd6];
    add.s64  %rd7, %rd3, %rd5;
    ld.global.f32  %f3, [%rd7];
    fma.rn.f32  %f4, %f2, %f1, %f3;
    st.global.f32  [%rd7], %f4;

$L__BB0_2:
    ret;
}
```

Listing 2.2: Simple CUDA kernel: PTX representation

[1]When used as kernel inputs, parameters are shared per-grid and are read-only

At the top of the PTX output, some general information about the used compiler used can be, which has been omitted in this case, followed by the `.entry` keyword, which marks the kernel function definition, including the kernel name and parameters. During the compilation with the CUDA C/C++ compiler, the kernel name has been mangled, which can be recovered using the special NVIDIA utility *cu++filt*[2]. Each function parameter is denoted by the `.param` directive, followed by its type, with the first instructions in the PTX code being responsible for loading these parameters into registers using the `ld.param` and `mov` directives. Corresponding to the first line of the kernel code, the `mad` instruction is used to calculate the current thread index, with the following two instructions `setp` and `@%p1` corresponding to the if-statement $if(i < n)$. The `setp` instruction compares two numbers using the `ge` (greater equal) operator and sets a predicate depending on the outcome. This predicate is then used to conditionally branch (`bra`) to the branch labeled `$L__BB0_2`, or continue the program sequentially. If this branch is taken, the kernel returns due to the `ret` instruction in the `$L__BB0_2` branch, with the program continuing and performing the multiplication and addition inside the if-condition otherwise.

2.2.2. SASS

In the second step before generating the final executable, the PTX representation is converted into another intermediary representation called SASS, which uses the instruction set of the targeted hardware. In contrast to PTX, there exists only limited official documentation for this instruction set, with a single instruction line following the basic format

```
1 (instruction) (destination) (source1), (source2) ...
```

and valid destination and source locations being registers of one of the types listed in the following table:

Register	Name
RX	Register
URX	Uniform register
SRX	Special system-controlled register
PX	Predicate register
UPX	Uniform predicate register
c[X][Y]	Constant memory
desc[URX][RY]	Memory descriptors

Table 2.2.: SASS register types [27]

[27] provides a list of valid instructions, with the instruction set for the Hopper architecture including instructions for floating-point and integer arithmetic, datatype conversions, as well as common memory operations, among others.

[2] `https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html#cu-filt`

```
1 .global _Z5saxpyifPfS_
2 /*0008*/ MOV R1, c[0x0][0x20] ;
3 /*0010*/ S2R R0, SR_CTAID.X ;
4 /*0018*/ S2R R2, SR_TID.X ;
5 /*0028*/ XMAD.MRG R3, R0.reuse, c[0x0] [0x8].H1, RZ ;
6 /*0030*/ XMAD R2, R0.reuse, c[0x0] [0x8], R2 ;
7 /*0038*/ XMAD.PSL.CBCC R0, R0.H1, R3.H1, R2 ;
8 /*0048*/ ISETP.GE.AND P0, PT, R0, c[0x0][0x140], PT ;
9 /*0050*/ NOP ;
10 /*0058*/ @P0 EXIT ;
11 /*0068*/ SHL R2, R0.reuse, 0x2 ;
12 /*0070*/ SHR R0, R0, 0x1e ;
13 /*0078*/ IADD R4.CC, R2.reuse, c[0x0][0x148] ;
14 /*0088*/ IADD.X R5, R0.reuse, c[0x0][0x14c] ;
15 /*0090*/ { IADD R2.CC, R2, c[0x0][0x150] ;
16 /*0098*/ LDG.E R4, [R4] }
17 /*00a8*/ IADD.X R3, R0, c[0x0][0x154] ;
18 /*00b0*/ LDG.E R6, [R2] ;
19 /*00b8*/ FFMA R0, R4, c[0x0][0x144], R6 ;
20 /*00c8*/ STG.E [R2], R0 ;
21 /*00d0*/ NOP ;
22 /*00d8*/ NOP ;
23 /*00e8*/ EXIT ;
24
25 /*00f0*/ BRA `(.L_x_0) ;
26 /*00f8*/ NOP;
```

Listing 2.3: Simple CUDA kernel: SASS representation

The disassembled CUDA binary file, including the SASS instructions of the simple kernel introduced earlier, can be generated using the *nvdisasm* utility[3] and is shown in listing 2.3. Even though the instructions used are of a lower level and hardware-specific syntax, similarities with the PTX representation can still be found. As in PTX, the thread index is first computed and then compared to the length of the array in the `ISETP.GE` instruction, the result being stored in the `P0` predicate. A difference can be seen in the handling of the if-condition, as no separate branch instruction is present, with the kernel directly concluding in the case of a negative predicate. Otherwise, the multiplication and addition operations are executed in a similar manner to before. A special syntax not present in the PTX intermediary representation are brackets enclosing two or more instructions in the SASS code, as seen at addresses `0090` and `0098` in the above listing, corresponding to a so-called dual issue, with both instructions being executed in the same cycle[4].

[3] `https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html#nvdisasm`

[4] Although not available officially documented, this is confirmed in the following forum post: `https://forums.developer.nvidia.com/t/in-sass-output-generated-for-cc5-0-cc6-0-and-cc6-1/111105`

2.2.3. Memory Management

When writing CUDA kernels, a strong understanding of the various memory types available, as well as unique characteristics they come with, is crucial. As selecting the appropriate memory type for a specific task is essential for achieving optimal kernel performance, it is important to be aware of trade-offs, implications, and potential pitfalls associated with each memory type. Currently, there are six types of memory available in CUDA, which are summarized in the following table:

Memory	Location	Cached	Access	Scope	Lifetime
Register	On-chip	N/a	R/W	1 Thread	Thread
Local	Off-chip	No	R/W	1 Thread	Thread
Shared	On-chip	N/a	R/W	All threads in block	Block
Global	Off-chip	No	R/W	All threads + host	Host allocation
Constant	Off-chip	Yes	R	All threads + host	Host allocation
Texture	Off-chip	Yes	R	All threads + host	Host allocation

Table 2.3.: CUDA memory types and traits [28]

Register

Registers offer the lowest latency among all CUDA memory types and store data private to each thread. It is important to note that registers cannot be explicitly managed by the programmer and are allocated by the compiler instead. To achieve optimal performance, the number of registers used in each thread should be minimized to maximize active threads per streaming multiprocessor, the allocation depending on the GPU architecture and the usage of the `-maxrregcount` compiler flag, which can be used to manually impose a limit per thread.

Despite the availability of a large number of registers in newer architectures, a kernel may exceed its allocated register limit, leading to excess data spilling to local memory instead, which can negatively impact performance due to increased memory traffic and/or a higher instruction count. Therefore, register spilling should generally be avoided whenever possible [1, 28].

Local Memory

Local memory also stores data private to each thread and shares the same lifetime as registers. As local memory is part of the primary device memory, its usage introduces a significantly higher latency than registers, with it also not being possible to programmatically declare variables to be stored in local memory, as this is decided by the compiler (Visible in PTX `ld.local/st.local` or SASS `LDL/STL` instructions). An important relationship between registers and local memory is called **Register Spilling**. When the kernel runs out of registers, additional data is saved to local memory instead at the cost of performance. To combat the effect of register spilling, limits on lower maximum register counts should be lifted, or the L1 cache size should be increased [1].

Shared Memory

Shared memory shares the same physical memory as the L1 cache, the partitioning of which can be adjusted by the programmer. It can be accessed by all threads in a CTA and offers low latency and high bandwidth. To achieve this, shared memory is organized into equal-sized banks that can be accessed concurrently, which, depending on the access patterns of individual threads, can lead to dramatically faster access times, with the use of shared memory having to be explicitly requested and managed by the programmer using the `__shared__` decorator. The introduction of the NVIDIA Ampere architecture also made it possible to easily copy data from global to shared memory with a single function call.

The main drawback of using shared memory is the possibility of encountering **Bank Conflicts**, occurring when addresses from multiple requests are mapped to the same bank, causing them to be serialized and degrading performance as concurrent access is no longer possible. Although avoidable, this leads to a much higher workload due to programmers needing to organize the data layout in a bank-aware manner [1, 28, 8].

Global Memory

Global memory represents the most significant memory type and, as such, introduces the highest amount of latency. It is also managed by the programmer with the `__global__` decorator and is accessible from all threads, making synchronization especially important. The `__restrict__` keyword can further be used to hint the compiler that the data will not be written to, allowing the use of a separate read-only data path improving performance [1, 15].

Constant Memory

Constant memory shares the same memory as global memory but is entirely read-only. It can be beneficial if many threads access the same addresses very often, for example, in the case of widely used constants. To declare a variable to use constant memory, the `__constant__` decorator is used [1].

Texture Memory

Texture memory is read-only memory specifically optimized for spatially local access patterns. This means that threads of a warp that read texture addresses should be near each other in a 2D/3D locality to achieve the best possible performance, making it ideal for tasks like image processing and scientific computing, where data is often accessed in a localized, non-sequential manner. Texture memory is part of global memory, with accesses additionally being cached [1, 28].

2.3. GPUscout

This chapter introduces the tool GPUscout, its architecture, and functionality [7].

2.3.1. Overview

GPUscout is a command-line application capable of analyzing CUDA kernels with the goal of finding performance bottlenecks and optimization potential, focusing primarily on data movement operations within the GPU and assessing whether they can be executed more efficiently. This is achieved mainly through static analysis of the intermediary representations of kernels, detecting predefined patterns that indicate sub-optimal memory usage, which is combined with kernel-wide metrics, warp stall samples, and concrete lines in the CUDA source code, to provide users with a detailed report on optimization hotspots along with actionable insights for improvement. Additionally, these metrics enable the comparison of different versions of the same kernel, with findings being compiled and presented to the user in a textual format, a detailed description of which can be seen in the following figure:

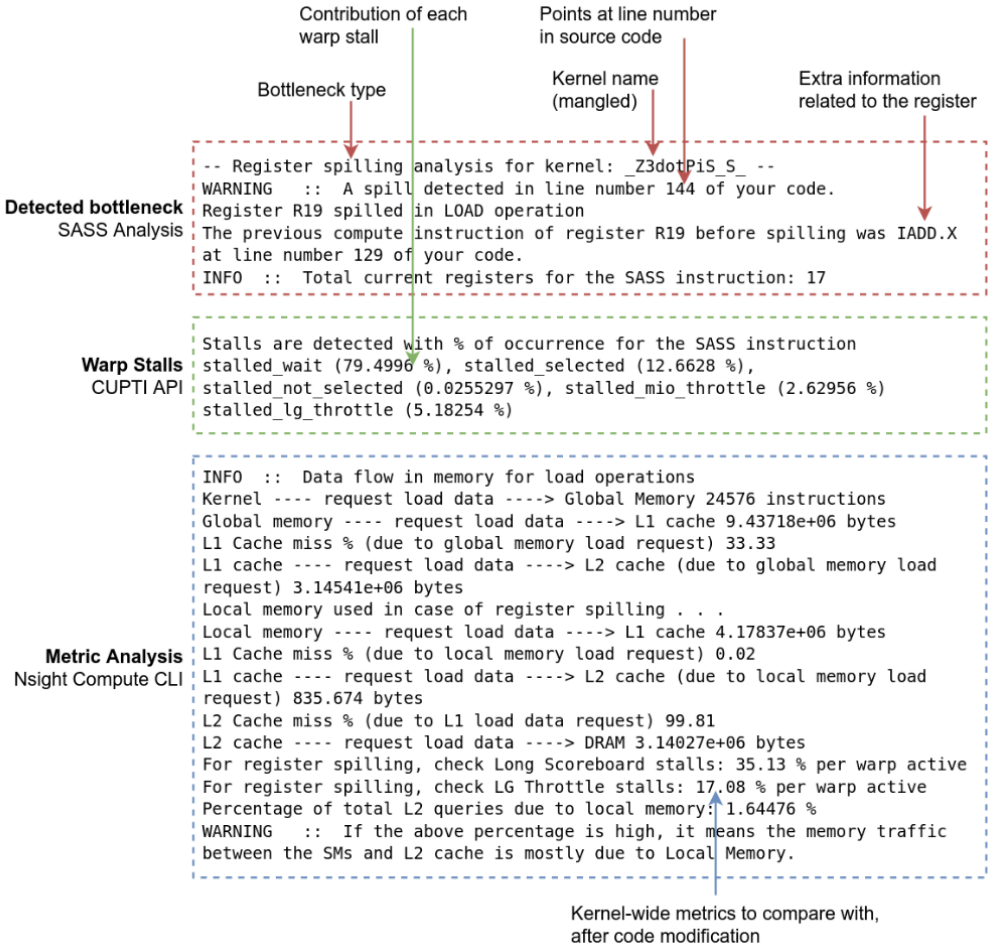


Figure 2.6.: Example GPUScout output [29]

SASS Analysis

The static analysis of the intermediary representation of the CUDA code represents the core functionality of GPUscout and is used to identify potential bottlenecks, as well as to provide concrete source code lines causing a particular issue. Depending on the analysis, either the SASS or PTX representation is used and analyzed with the goal of locating specific instruction and register usage patterns, focusing on different memory-specific behaviors. This analysis is also designed modularly to be able to easily extend the application beyond the currently implemented analyses.

Warp Stalls

Using the NVIDIA CUPTI API, instruction stalls at problematic SASS lines identified by analyses are fetched. As such stalls can occur due to a multitude of reasons, such as waiting for the completion of a certain memory operation or the availability of a resource, users are provided with a very verbose output to provide as much context as possible.

Performance Metrics

In addition to warp stalls, several other kernel-wide metrics are included in the textual output to provide additional guidance in determining the severity of a potential bottleneck, as well as the ability to more easily determine the impact of implemented changes. Certain metrics, such as high cache miss rates or low occupancy, can also, on their own, be used to quickly identify problems or optimization potential. These metrics are collected using the NVIDIA Nsight Compute CLI.

2.3.2. Analyses

This section briefly introduces all nine analyses currently implemented by GPUscout and the significant metrics associated with them.

Use Vectorized Loads

Instead of using regular 32-bit transactions when loading data from global memory, 64- or 128-bit vectorized load instructions can be used when data is aligned correctly, which can greatly reduce the number of loads by fetching multiple data pieces in one transaction. The key factor in deciding if vectorized loads are applicable in a certain situation is the existence of multiple 32-bit load operations accessing neighboring memory locations, with register pressure being the most critical metric to monitor, as an increase can negatively impact performance by leading to resource limitations and reduced occupancy. To help developers determine if a change is expected to improve performance, detailed information about the number of additional registers required by each SASS instruction is provided. Additionally, long scoreboard stalls are reported, with a high percentage indicating that threads spend significant time waiting for global or texture memory operations.

Register Spilling

When a program runs out of registers, data is temporarily stored in local memory instead, leading to an increase in latency and memory traffic and a decrease in instruction count. Users are informed about the specific registers that spilled, relevant source code lines and the previous SASS operation most likely causing the spill. As register spilling leads to an overall increase in data movements between registers and local memory, information about GPU cache usage and miss rates is also provided. This enables users to assess if these spills occur due to bandwidth limitations, and performance improvements can be expected after implementation changes. Long scoreboard and LG throttle stalls are also shown, as they indicate local and global memory throttling and can thus be used to quantify the impact of register spills and evaluate the benefits of resolving the issue.

Use Shared Memory

When data is used regularly, shared memory should be used instead of global memory, offering a lower latency. GPUscout identifies data that is accessed repeatedly (for example, in for-loops) and recommends using shared memory if global memory is used. To be able to more easily quantify the current usage of problematic registers, the number of global loads and computation instructions the relevant register has been used in are provided. Another crucial metric is the number-of-way bank conflicts, representing the maximum number of threads accessing the same memory bank simultaneously.

Use Shared Atomics

Atomic operations involve a read-modify-write sequence on global or shared memory, ensuring thread safety. While global atomics are serialized across the entire kernel, shared atomics are serialized at the block level. GPUScout helps identify excessive use of global atomics, particularly in situations like for-loops, where such operations can create memory bottlenecks. The tool also provides information on relevant LG throttle stalls, which can be reduced by switching to shared atomics, as well as MIO stall potentially increasing due to the use of shared atomics, making monitoring these metrics closely during optimization critical. Additionally, GPUScout analyzes atomic data movements within the GPU and presents relevant data for further optimizations.

Use Read-only Cache

The `__restrict__` keyword is used to inform compilers about load operations using the read-only cache, allowing for more aggressive optimizations. Because of this, GPUscout searches for pointers that are only read in the kernel and thus could benefit from using this keyword. High register pressure can hinder performance improvements, so this metric is output alongside the registers to optimize.

Use Texture Memory

When multiple loads accessing data from adjacent memory addresses are found, whose values are never written to, using texture memory can offer performance improvements. Users are also informed about the presence of spatial locality in load operations, indicating whether specifically texture memory can offer benefits. TEX throttle stalls are provided along source code lines and registers that fulfill these criteria, indicating if too many outstanding requests are filling the TEX pipeline.

Datatype Conversions

Due to their impact on the instruction count, as well as often requiring utilization of multiple pipelines on the GPU, datatype conversions are expensive and thus should be avoided where possible. GPUscout searches for and presents the user with the individual cases of conversions and their location in the source code as a general metric to lower and keep in mind when optimizing. Relevant stalls based on the type of identified conversions are also provided for users to compare before and after implementing changes.

Warp Divergence

Using conditionals like if-else statements in kernels can lead to some threads not having any work to do, resulting in sequential instead of parallel execution. Although the compiler can resolve this using predicated instructions in simple cases, conditional branches are introduced in more complex scenarios, for example, when if-else constructs are nested, leading to performance degradation. Because of this, users are provided with the locations of branches in the code, as well as the average percentage of branch divergence.

Deadlock Detection

When using atomic lock and unlock operations, there is a risk of creating deadlocks, which can lead to an irrecoverable application state. GPUScout detects specific patterns in the intermediate SASS code of the kernel that suggest the presence of a deadlock, helping to identify and mitigate potential issues.

3. User Interface Design

User interfaces (UIs) serve as the main means of communication between an application and its users, making it essential to understand user needs to create a user-friendly and intuitive interface. Previous publications have proposed various guidelines to follow when designing UIs, one example being [30]. In this chapter, some of these proposed design guidelines are introduced and compared to establish a clear set of principles to guide the design of the application developed in this thesis, with the user interfaces of similar projects additionally being presented, highlighting their distinctive features and design choices.

3.1. General Guidelines

One of the most important factors in designing user interfaces is ensuring that users who have never interacted with the application can easily understand and use the UI without needing a manual or extensive explanation, emphasized by Everett's Law, which states that "For every UI design that requires experimentation, assistance, or training to understand, there exists an alternative, self-explanatory design that doesn't" [31]. This section compiles and compares design guidelines and principles from various sources. While some sources like [32] offer specific recommendations for different stages of the design process, others, such as [33] and [34], provide a set of general principles to follow.

Interface Structure and Layout

Consistency is essential, particularly in more complex applications where users frequently navigate between multiple pages, where a consistent design language helps users to quickly recognize familiar and recurring elements. However, to avoid confusion, consistency should be limited to these recurring elements, as over-applying consistency across all pages can lead to difficulties in the user's ability to distinguish between different sections or pages, potentially hindering navigation. One example would be navigation controls or controls for accepting or dismissing an action, which should always be styled and positioned the same. This can, for example, be achieved by reusing certain components or creating templates for frequently used elements. In addition to an intuitive design, every page of an application should be limited to a clearly communicated main instruction that every element revolves around, which can be implemented using a concise and unambiguous page title, indicating what a user can expect from a page. When designing the layout of the individual components of a particular page, several rules should be followed:

Breaking up elements into sections Dividing a page into sections, with elements grouped

closer together within each section, helps users better understand their roles and reduces confusion. Elements within the same section should relate to a common task, making it easier to identify their primary purpose, while sections themselves should be distinguishable primarily by their positioning, with additional styling like background colors, headings, or separators commonly used when the layout alone is not expressive enough.

Minimalism In several aspects, the design and content of a page should be kept as minimalistic as possible. UI elements that are not clearly related to the main instruction of a page, as well as supplementary information that is not required to understand the role of a certain aspect of the interface, should be hidden in subpages or popups. This concept also applies to the number of actions required to access certain elements or reach information not displayed by default, which should be kept as low as possible.

Organizing elements in a visual hierarchy [31] suggests that users, in addition to reading left-to-right and top-to-bottom, also often scan pages in the same fashion, leading to the primary content of a page ideally being located in the top half, with action buttons located in the bottom right corner acting as the natural conclusion of the page. This can be altered by using special components that draw the user's attention and should be kept in mind when designing the layout of the elements on a page. As can be seen in figure 3.1, if not otherwise incentivized, content in the bottom left corner of an application will often go unnoticed, as the top and right sections of a page are the natural focus fields of humans.

Aligning elements All elements on a page should be aligned consistently, for example, along an invisible grid. Using only a few alignment lines makes a page appear tidy, improves readability, and can help emphasize the relationship between elements.

Following these guidelines ensures that users won't feel overwhelmed by the elements on a page while still being able to easily find the desired functionality and information [31, 32, 33, 35, 36].

Individual elements

Designing UI elements can be challenging, as small aspects like their size can have a huge impact on their perception and sometimes automatically lead to certain assumptions about functionality or purpose, the dimensions of a text input being perceived as an indicator of the expected input size, for example. The size of an element is also often aligned with its importance, so sizing all elements similarly should be avoided, as this can negatively impact readability. Ideally, the purpose of an element should be recognizable without the need for additional textual descriptions, with no risk of users accidentally performing actions they do not want or expect. Providing users with appropriate feedback after performing an action or providing input is also crucial, as it helps them understand the outcome of their interaction, indicating whether an action was successful or any errors occurred [31].

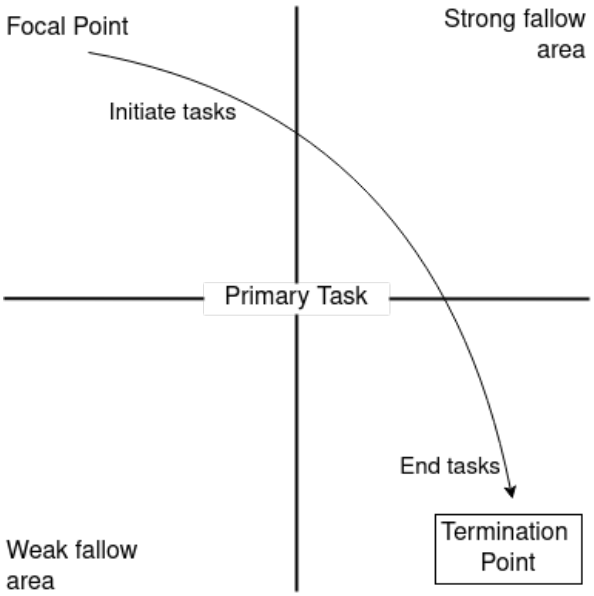


Figure 3.1.: Visual hierarchy of a page [31]

Waiting

Generally, while applications should strive to reduce latency perceived by the user as much as possible, there are situations in which users need to wait for certain operations to complete. Especially in these cases, it is important to keep users informed about the current state of the application to reduce frustration and maintain engagement. Providing clear progress indicators, such as loading spinners, progress bars, or status messages, helps set expectations and reassures users that the system is actively working, with estimated wait times or interactive elements further improving the user experience [34].

Navigation

Especially in complex applications with multiple pages and subpages, providing an easy and intuitive form of navigation is crucial, ensuring users never get into a state they do not know how to return from. Popular approaches include home buttons that redirect to the same base page, navigation panes always visible at the top or side of every page, and back buttons, which revert the interface to the previous state. Even more importantly, users should always be informed which page they are currently on [31, 32, 33].

Text

How text is presented to a user can greatly impact their ability to quickly understand the concept of a particular page or element, for example, when working with numbers, where proper formatting can reduce the time to grasp their meaning significantly. It is also important not to overwhelm users with too much text, distracting them from important information,

which can be achieved by hiding more detailed explanations behind a certain state. Apart from bold and italic text used to highlight important information, the text should also not be decorated, with differently sized text sections being a useful technique for providing a sense of visual hierarchy instead. When used correctly, textual descriptions of elements can be substituted or accompanied by icons, which can help to easily convey the type of action or element. To avoid confusion and ambiguity, the application's domain and target audience should be considered [31, 32].

Color

Using colors correctly in user interface design is crucial, as it can help emphasize important elements and inform users about the type of action they are about to perform. The following are some of the most important guidelines that should be followed in terms of coloring certain parts of a user interface:

Avoid misinterpretation Several colors accompany inherent assumptions about the elements they are applied to, with one example being the color red, which typically represents a destructive or dangerous action. Using this color for anything other than what users expect may confuse them and lead them to perform actions they did not intend to.

Use color to supplement, not convey information Colors should never be used as a sole means to define the purpose of a certain element but accompanied by a textual description or icon.

Color palette When choosing the color palette of an application, similar colors should be avoided, with each color being distinct from others to avoid misinterpretation and chosen with a specific role in mind, which consists throughout the entire application. Placing opponent colors on top of each other should also be avoided, as this can impact readability, especially in the case of the text color.

Following these guidelines enhances usability, strengthens visual hierarchy, and creates a more intuitive and cohesive experience across the application [31, 32, 34].

3.2. Application-specific Guidelines

The graphical application created in this thesis is intended for a diverse audience, including beginners with limited knowledge of CUDA kernels and GPU programming, as well as experts in the field. Therefore, presenting information in a way that ensures each user receives the appropriate level of detail and supporting explanations is crucial.

Beyond the previously established guidelines, offering comprehensive explanations of elements and actions is essential, which should be easily accessible through intuitive controls. For instance, when presenting a statistic that the user may not be familiar with, a practical approach would be to include a help icon that, when clicked, opens a popup with a detailed explanation of the statistic, allowing users to gain an understanding of relevant concepts

without having to leave the application. Additionally, providing default values for input fields or preselecting certain options can help users avoid mistakes and simultaneously learn about the topic.

3.3. Related Projects

Even though GPUscout is fairly unique in its ability to statically analyze CUDA source code and its intermediary representations while providing recommendations tied to concrete statements, plenty of tools exist to assist programmers in developing correct and performant HPC applications, with most of these tools focusing on profiling certain aspects of an application using sampling or instrumentation-based techniques. While this does not align with the main focus of GPUscout-GUI, their approaches to visualizing analysis results can still be beneficial and will be presented in the following sections.

TAU

The *TAU* parallel performance system is a profiling and tracing toolkit for the analysis of parallel programs and is organized into three layers: instrumentation, measurement, and analysis. During the compilation and runtime of a kernel, additional instructions or probes are inserted, which allow for the collection of performance statistics by calling the *TAU* measurement system through a provided API responsible for collecting and aggregating this data according to user-defined configuration. In the case of NVIDIA GPUs, the *TAUcuda* measurement library is used to expose an interface for measuring several performance profiles.

Included with the *TAU* system is a separate analysis tool called *ParaProf*, which includes a graphical user interface with a focus on modularity, flexibility, and extensibility. After selecting one or more data sets to investigate or compare against each other, a global profile view displays user-defined metrics for all applications across all threads, several separate views offering more in-depth information on a specific topic [37, 6].

HPCToolkit

Developed by Rice University, *HPCToolkit* allows CPU and GPU performance metric collection using either profiling APIs provided by GPU vendors or custom hooks, with the Cupti API being used in the case of NVIDIA GPUs. In addition to the usage through the command line, a separate tool, *hpcviewer*, implements an interactive user interface, providing an easy way to visualize the collected data organized into two main modes: profile and trace view. In the profile view, the calling context of a kernel is displayed, with pre- or user-defined metric information being provided in a fine-grained manner and three further modes defining how these metrics are associated with each other. The second trace view provides a time-based view of program execution, allowing users to observe detailed insights into the call path evolution of each thread [5, 38].

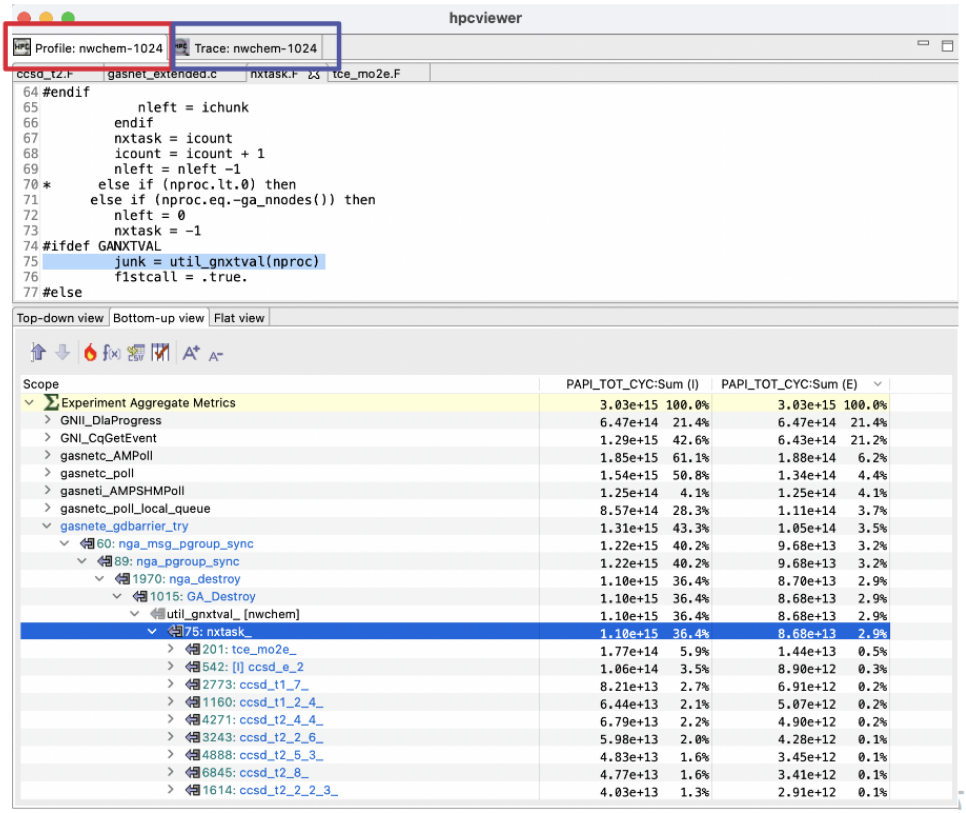


Figure 3.2.: HPCViewer profile view [38]

CudaPAD

CudaPAD is an open-source application enabling users to compare the source code of a CUDA kernel with its intermediate PTX and SASS representations. It provides comprehensive syntax highlighting for all code representations, helping users analyze and understand their code more effectively. In addition to syntax highlighting, *CudaPAD* offers valuable features such as line-by-line mapping between the source code and its PTX and SASS representations, making it easier to track how high-level code translates into lower-level instructions. The application also includes functionality for highlighting custom registers, which aids in identifying register usage and potential optimization opportunities. A key feature of *CudaPAD* is its real-time code modification capability, allowing users to edit the source code directly within the application, with the kernel automatically being recompiled whenever changes are detected and immediate feedback provided by highlighting which PTX and SASS lines have been updated compared to the previous version, facilitating an efficient workflow for iterative code optimization and debugging [39].

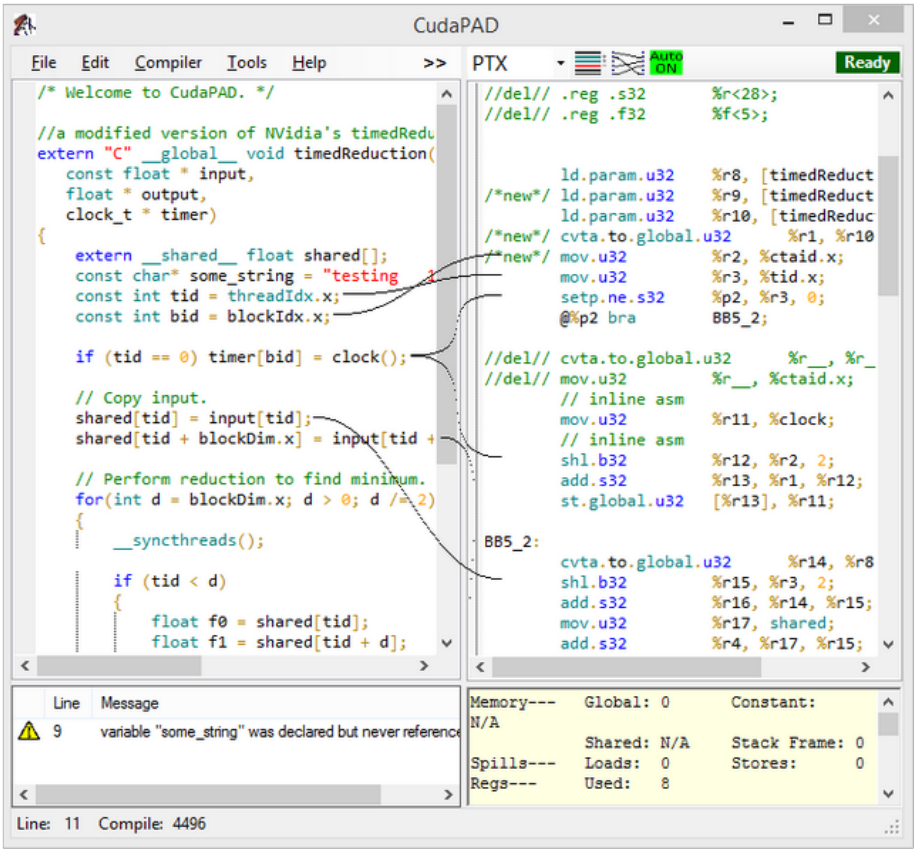


Figure 3.3.: CudaPAD user interface [39]

Nsight Compute

Nsight Compute is an official tool by NVIDIA that provides interactive kernel profiling capabilities for CUDA programs [3] and is available both as a command-line application and through a graphical user interface, allowing for extensive customization and data collection from a comprehensive list of metrics. Data is collected mainly by inserting libraries into the application process and intercepting all communication between the host and the CUDA driver. To be able to provide the best user experience, several UI components provide visualizations and configuration options, including, but not limited to, the following:

Source viewer The source viewer allows users to quickly compare two different source files. This can, for example, be used to compare the CUDA source code with either the intermediary SASS or PTX instructions side-by-side with a mapping indicating which code lines of the two map to each other or to view two different versions of the same kernel. In both cases, each line can be additionally annotated with user-defined information like live registers, stall samples, or instruction counts, allowing for quick identification of potential bottlenecks and problems.

Memory charts and tables Several windows dedicated to visualizing memory transfer and utilization metrics allow fine-grained insight into the memory topology of the used GPU.

Report result The report result page consists of several tables and charts presenting all collected metrics in a single interface. A notable feature is the ability to specify a particular result as a baseline, which all further analyzed kernels will be compared against, allowing for a simple comparison of different kernel versions.

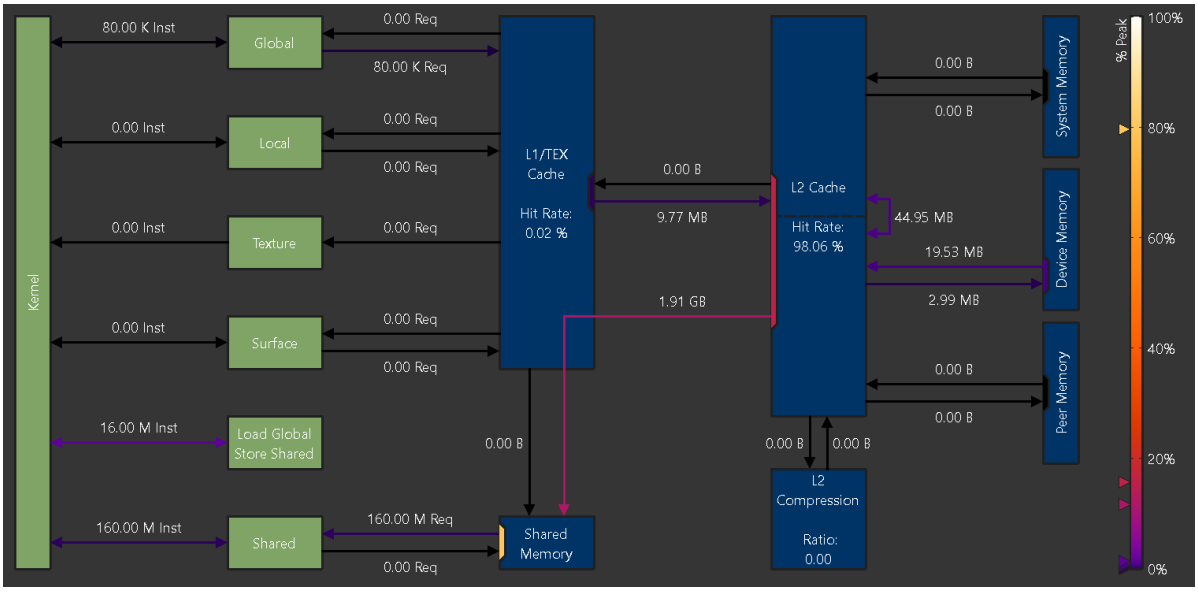


Figure 3.4.: NVIDIA Nsight Compute memory chart [3]

4. Design and Development

In order to create a user-friendly interface for the GPUscout-GUI application that includes all essential features for both experienced CUDA users and those new to GPU programming, several trade-offs and design choices had to be made. This chapter provides insights into how different aspects of the UI were implemented and explains the reasoning behind certain design decisions.

4.1. Program Architecture

The selection of the architecture and technologies used to build applications is essential to the effectiveness of implementing requirements and is a key factor in ensuring long-term maintainability and scalability.

4.1.1. Technologies Used

Platform independence is among the most important requirements the technologies used to create GPUscout-GUI have to meet, as, although Linux is the primary platform for the application, compatibility with all major operating systems is essential to prevent unnecessary limitations and broaden accessibility. Several programming languages were evaluated on their ability to define graphical interfaces and their functionality in a simple and understandable manner. While frameworks enabling the creation of graphical applications exist for all major programming languages, their functionality and ease of use differ significantly, with a combination of traditional web standards like HTML, CSS, and Javascript offering the best useability. In order to be able to use these technologies in the creation of standalone applications, *Electron* has been used. *Electron*[1] is an open-source framework combining the Chromium rendering engine and Node.js runtime to enable seamless integration of web-based user interfaces with native system capabilities, which not only allows development completely independent from the target operating system but also offers the potential to easily implement a web-based version of the application if desired in the future. Another major requirement is a modular and component-based workflow, which is supported by the use of an additional framework, *VueJS*. *VueJS*[2] is an open-source Javascript framework for building highly modularized single-page applications and is designed to be lightweight and easy to integrate into existing projects. Its use significantly improves development workflows by consolidating all code related to a specific component within a single file

[1] `https://www.electronjs.org/`
[2] `https://vuejs.org/`

rather than separating it by language, as well as providing a highly efficient reactivity system automatically updating UI elements in the event an associated property changes.

4.1.2. Code Structure

When working on large projects especially with multiple different frameworks, organizing the source code in a clear and logical way is essential for maintaining readability. In the case of GPUscout-GUI, the main working directory is divided into four distinct subdirectories, each corresponding to a specific application layer.

Due to the use of technologies traditionally used to create web applications, additional configuration is necessary to ensure correct functionality, with the content of the `main` directory acting as the entry point of the application and corresponding to the backend of a webpage. Using the *Electron* framework, the desktop window is created, and interactions with the underlying operating system to handle functionality like selecting system files are implemented. The actual definition of the interface and its individual components is stored in the `renderer` folder, which contains all code associated with *VueJS* and behaves like a traditional web project created with this framework. Due to this, code in the front end does not have direct access to native operating system functions provided by *Electron*, with communication between these two instances being handled by the code in the `preload` directory, which allows the exchange of messages between them. Lastly, configuration files for all three layers are aggregated in the `config` directory, providing a single location for defining all dynamic and interchangeable content that may need to be modified on a regular basis. This separation enables, for example, adjusting the information displayed in the UI or implementing support for new analyses without altering the application's source code.

4.2. Internal Data Representation

To display all desired information, GPUscout-GUI needs considerably more information than the textual output that GPUscout currently provides. This section explains the input data needed by GPUscout-GUI and outlines how this data is processed and represented internally.

4.2.1. Input Data

Inputs from two different data sources can be used with GPUscout-GUI, namely the analysis result produced by GPUscout and an optional specification of the memory topology of the GPU, which was used to run the analysis.

GPUscout Output

In order to generate recommendations, GPUscout aggregates data acquired from multiple sources, among which are the SASS and PTX intermediary representations of the analyzed binary file, as well as a set of metrics and stall samples. As can be seen in listing 4.1, the output of an analysis in GPUscout is formatted in a way that is easy to read for humans and

focuses on the most important information necessary to locate and fix potential performance bottlenecks. For each analysis, individual instances of problems, including information such as the source code line and the name of the relevant register, as well as optionally recorded stall samples in the corresponding SASS line, are listed, followed by kernel-wide metrics relevant in the context of this particular analysis.

```
1 INFO :: Register R4, in line number 10 of your code, is not aliased anywhere in
    ↪  the kernel
2 WARNING :: You can benifit from using __restrict__ for register R4 at line
    ↪ number 10 of your code
3 INFO :: Register R6, in line number 10 of your code, is not aliased anywhere in
    ↪  the kernel
4 WARNING :: You can benifit from using __restrict__ for register R6 at line
    ↪ number 10 of your code
5 INFO :: Total current registers for the SASS instruction: 5
6 Stalls are detected with % of occurence for the SASS instruction
7 stalled_wait (100 %)
8 If using __restrict__ (read-only cache), check IMC miss: 51.46 % per warp
    ↪ active
```

Listing 4.1: GPUscout output: Simple CUDA kernel (Restrict analysis)

As this output is not easily machine-readable, it cannot be used directly as an input for GPUscout-GUI. Additionally, due to a large amount of information necessary in the GUI not being included in the textual output, GPUscout was extended with an optional flag `--json` to generate a JSON-formatted output file containing all the information required, in addition to the human-readable console output. JSON was chosen as a data format for its easy integration with different programming languages and for being both human- and machine-readable.

As mentioned, the generated file not only contains the data already displayed to the user in a different format but contains additional information, which is then used to provide more detailed and easier-to-understand insights into the potential bottlenecks found by GPUscout. The general structure of files generated using this flag can be seen in listing 4.2, which, in addition to the `analysis` attribute containing information about the analyses run and problems found in them, defines five attributes with data not present in the textual output of GPUscout. In the `binary_files` attribute, the three intermediary files generated during the compilation of the CUDA code are passed, namely the PTX and SASS representations, as well as an annotated version of the SASS representation with information about the number of live registers at each address. In the current output of GPUscout, kernels are identified by their mangled names, which, even though still identifiable, can get confusing, especially for kernels with multiple complex parameters or in cases of function overloading. To improve readability and avoid confusion in the user interface, especially for inexperienced users, the actual demangled kernel names with their parameters are passed in the `kernels` attribute, while a list of all metrics that GPUscout either acquired using NVIDIA's NSight Compute CLI or computed based on them is passed in `metrics`. One of the main intended features of GPUscout-GUI is the ability to compare the PTX and SASS intermediary representations

with the relevant source code files while providing mappings between them. Because of this, the content of all source files, which are mentioned in line mappings of either intermediary representation, are passed in the `source_files` attribute. Lastly, the results of the PC sampling are included under `stalls`. More information about the metrics and analysis-specific information included in the result file can be found in appendix A.

```
{
  "analyses": {
    "datatype_conversion": {},
    "global_atomics": {},
    ...
  },
  "binary_files": {
    "ptx": "...",
    "sass": "...",
    "sass_registers": "..."
  },
  "kernels": {
    "_Z4axpyifPfS_": "axpy (int, float, float*, float*)"
  },
  "metrics": {},
  "source_files": {
    "/path/to/source.cu": "..."
  },
  "stalls": {}
}
```

Listing 4.2: GPUscout JSON output structure

Memory Topology

In addition to the output generated from GPUscout, support for another optional data source has been added in MT4G[3], a tool created at the Chair of Computer Architecture and Parallel Systems (CAPS) at TUM, capable of capturing the memory topology of NVIDIA GPUs. If available, the generated file provides additional insights with metrics detailing, for example, the size and latency of certain memory components of a GPU. The tool output is formatted as a CSV file and, as such, can be easily read programmatically, eliminating the need to implement support for another format.

4.2.2. Data Parsing and Storage

Even though the data generated by GPUscout is JSON-formatted and, as such, easily machine-readable, it still needs to be parsed to be usable in the application. After receiving a JSON file

[3] `https://github.com/caps-tum/mt4g`

(as well as optionally a CSV file from MT4G), parsing is done in three steps: first, all general information is parsed and stored into an `GPUscoutResult` object, followed by analysis-specific data, which is split into separate `Analysis` objects. Lastly, each so-called occurrence, which internally describes information related to an individual optimization in the GPUscout output and can be uniquely identified by an analysis and line number in one of the intermediary representations, is stored in instances of the `Occurrence` class. A detailed description of each of these steps can be found in the following sections.

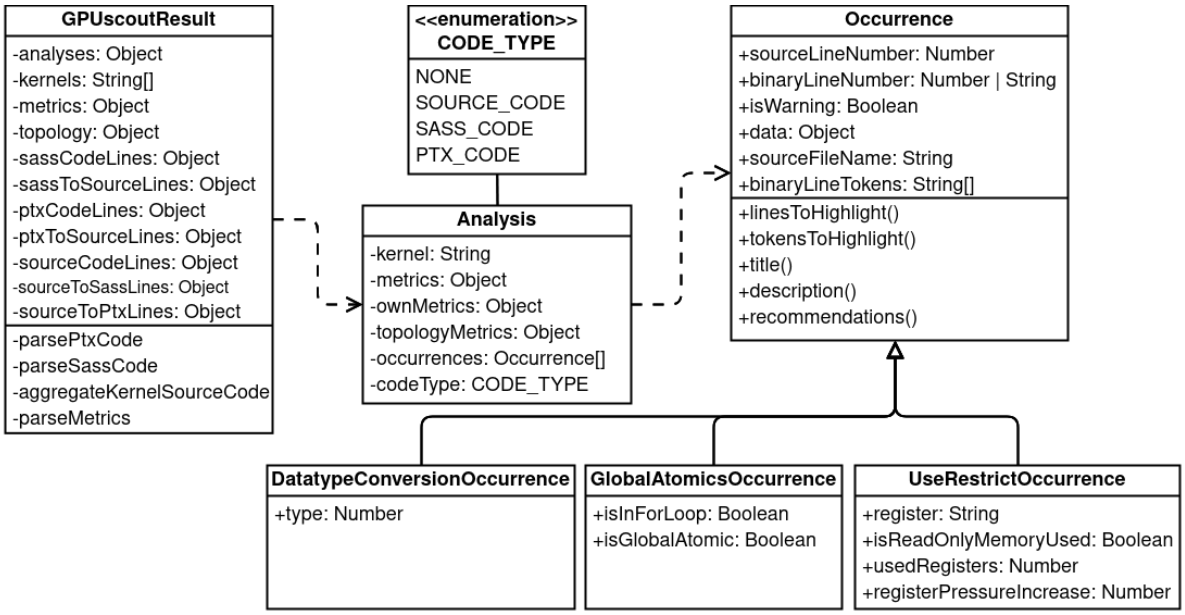


Figure 4.1.: Class diagram: GPUscout-GUI internal data representation

GPUscout Result

The main responsibilities of an instance of the `GPUscoutResult` class are to parse and extract information included in the intermediary PTX and SASS, as well as all source files, in addition to handling the remapping of kernel names. As mentioned before, GPUscout operates with mangled kernel names, which GPUscout-GUI aims to replace with their demangled counterparts using the mapping from mangled to demangled kernel names included in the JSON output of GPUscout. Even though the mangled kernel names could just be replaced with their demangled version, this can lead to problems when comparing two GPUscout results. In the scenario of a user changing the parameters of a kernel in the process of resolving a performance problem found by GPUscout, the mangled kernel names will not align anymore, preventing the kernels from being able to be compared in GPUscout-GUI, as the kernel name is used as their sole identifier. This problem is resolved by removing the parameters from demangled kernel names in cases where this does not load to conflicts and replacing the mangled name with this shorter version, leading to kernel parameters only being visible in the user interface in the case of overloaded kernel names. This ensures

maximum flexibility when comparing different versions of a particular kernel while also reducing the amount of unnecessary information displayed in the UI.

Both the intermediary PTX and SASS files, as well as all source files, contain various pieces of information that need to be extracted, most importantly the mappings between individual code lines, which are a crucial part of the additional value offered by the graphical representation of results. While keeping track of the current kernel, the following information is stored for each line in the SASS file that is either a label or starts with an address:

address : The address of the current SASS line. If the current line corresponds to a label, the address of the following SASS line is used instead. If a label marks the last line of a kernel, its name is used as the address of that line, as no further information is available.

tokens : Instead of storing the entire SASS line as a string, it is split into individual tokens stored in an array. This is required to be able to later highlight individual parts of a line in the UI[4].

liveRegisters : Corresponds to a list containing the number of live registers by category. While this always includes general-purpose (GPR) and predicate registers (PRED), further entries may correspond to register types only available in specific scenarios. One example of this are uniform versions of these registers (UGPR/UPRED), which are only available on newer architectures of NVIDIA GPUs, although not publicly detailed which ones.

stalls : Corresponds to a dictionary mapping warp stall reasons to the number of their occurrences in the given SASS line.

To be able to map each SASS line to a specific line in a source file, specially formatted comments have been inserted into the SASS file by the NVCC compiler, mapping the following SASS lines to the specified file and line until the next occurrence of such a comment. The code responsible for parsing this part of a SASS file can be seen in listing 4.3.

```
if (line.includes('//##')) {
    // //## File "FILE_PATH", line LINE_NUMBER
    // This line maps the following SASS lines to a certain source file and
        ↪ line number -> save both
    const sourceLine = parseInt(line.split(' ').at(-1));
    const file = line.substring(line.indexOf('"') + 1, line.lastIndexOf('"'));

    currentSourceLine = sourceLine;
    currentSourceFile = file;
}
```

Listing 4.3: Application code: Mapping SASS lines to corresponding source file and line

[4] A SASS line is split using the following regex: `/([+-,.:[\]()])/`

The PTX file is parsed in a similar manner, with the exception that no information about either live registers or PC sampling stalls is available. Similarly to the SASS file, a mapping to source file lines is also provided in the form of comments and parsed accordingly. One special case that can occur during the parsing of PTX files is kernels appearing more than once, which can have many reasons, the most prevalent being that the analyzed binary has been compiled with multiple target architectures, resulting in a separate kernel entry for each architecture. In this case, only the PTX code of the first occurrence of each kernel is parsed, and subsequent occurrences are ignored. After all intermediary files have been parsed successfully, the resulting line mappings can be used to identify which source files are relevant for each kernel, with a source file being considered relevant if at least one line in one of the intermediary files maps to a line in this particular file. This avoids the necessity of displaying source files that are not relevant to the current state of the interface, which could confuse users.

Analysis

An instance of the `Analysis` class represents the GPUscout results of a single analysis related to a particular kernel and can thus be identified by the analysis and kernel name. In addition to references to both the metrics included in the JSON file, as well as optionally the topology metrics that have been parsed in the `GPUscoutResult`, each analysis can contain some analysis-specific metrics stored in the `ownMetrics` attribute, which are included in the analysis data rather than the metrics in the JSON result. Because kernel-wide metrics are collected independently from individual analyses in GPUscout, metrics that are dependent on the results of, for example, the SASS parsing of a kernel, need to be stored separately during the relevant analysis to avoid duplication of code. An example of this is the number of shared and global atomics, which is computed during the global atomics analysis and only available in its scope.

Occurrence

For each analysis, the GPUscout output can contain multiple occurrences, which can be either information about code lines that already use a particular optimization (marked by the `INFO` keyword in the textual output) or points to areas where optimization potential has been found (marked by the `WARNING` keyword in the textual output). Even though the command-line output of GPUscout only mentions the relevant line in the source file, each occurrence can be uniquely identified by a line in the intermediary PTX or SASS file. Because occurrences of different analyses require different data points to convey the relevant information, only shared attributes are defined in the base `Occurrence` class, with subclasses for each analysis implementing analysis-specific information. In addition to the relevant source and intermediary line as well as file, the `isWarning` attribute specifies whether this occurrence corresponds to an optimization or is solely informational, with the `data` attribute making the remaining analysis-specific attributes available to the scope of the class.

The `Occurrence` class also implements five methods, specifying how its information is com-

municated to users in the user interface. The methods `linesToHighlight` and `tokensToHighlight` can be used to define which lines and tokens in the intermediary representation are relevant to the current occurrence and should be highlighted, with the remaining methods generating the textual output based on the attributes of the occurrence, which will be displayed to the user upon selecting this occurrence in the UI. More information about how this data is displayed to the user can be found in section 4.5.2.

4.2.3. Configuration

To make the process of modifying and extending the user interface as simple as possible, the configuration of the content displayed to users has been completely decoupled from the source code of the application. Among the configuration files for analyses and metrics, which will be explained in further detail, are also files for general settings like the color scheme used, the texts shown in the UI, as well as the paths of used files.

Analysis

For an analysis included in GPUscout and the resulting JSON file to be visible in GPUscout-GUI, it needs to be configured correctly, ensuring that the information included in the analysis is parsed and interpreted correctly and no misinformation is presented in the user interface. An example of the configuration entry for the register spilling analysis can be seen in listing 4.4, which contains six attributes. The `name` and `display_name` are used to define the internal name of the analysis, which corresponds to the name of the analysis in the JSON file, as well as the name presented in the GUI. Two flags allow specifying which intermediary representation will be displayed for a particular analysis, as well as if live register information should be displayed in the case of SASS code. Even though a default class for the occurrences of an analysis is provided, as mentioned in section 4.2.2, this should only be used as a fallback, with custom subclasses containing all the information of a particular analysis being passed to `occurrence_constructor`. Lastly, the metrics and the form in which they should be displayed in the user interface are defined in the `metrics` attribute, for which two distinct components are provided, allowing arranging them in either a list or graph-like manner, with the value of this attribute accepting a list of any number of them. This allows users to easily add or modify metrics without needing to modify the source code of the application. If no metrics are defined for an analysis, a default metrics list with only those unique to this analysis will be displayed.

```
1 "register_spilling": {
2     "name": "register_spilling",
3     "display_name": "Register Spilling",
4     "use_sass": true,
5     "display_live_registers": true,
6     "occurrence_constructor": (o) => new RegisterSpillingOccurrence(o),
7     "metrics": [...]
```

```
8 }
```

Listing 4.4: Application code: Configuration of register spilling analysis

Metrics

Most metrics in GPUscout are used in multiple analyses, so storing their information in the analysis definition would lead to code duplication. Instead, all metrics used in GPUscout-GUI are defined in a single file and only referenced where needed, with an example configuration of the metric for the total number of instructions executed in a kernel shown in listing 4.5. Similarly to the configuration of analyses, the two attributes `name` and `display_name` correspond to the internal, as well as the display name of the metric, with the `hint` and `help_text` defining additional supporting information for the given metric, made available to users when necessary. Even though most metric values are numerical, simply displaying them to the user without any information about the unit would lead to confusion and possibly wrong assumptions. Because of this, the `format_function` attribute accepts a function, formatting the value of the metric into a string, which, in this example, informs the user that the metric represents a number of instructions. Lastly, in order for the UI to be able to display an indication of whether the value of a particular metric has improved or declined when comparing two kernel versions, the `lower_better` flag can be used to define if a lower metric value is considered an improvement.

```
1 "general_total_instructions": {
2     "name": "general/total_instructions",
3     "display_name": "Total instructions",
4     "hint": "Total number of instructions executed",
5     "format_function": formatInstructions,
6     "help_text": "...",
7     "lower_better": true
8 }
```

Listing 4.5: Application code: Configuration of a metric

4.3. Landing Page

The landing page is the first screen presented to a user upon launching the application and contains functionality for selecting all relevant input files, enabling users to select one or two GPUscout result files, depending on whether they want to view a single result or perform a comparison. To not overwhelm users new to the application with a large number of elements, only one GPUscout result and topology file can be selected initially, with a second identical set of inputs only becoming visible after users confirm they want to compare two results. This reduces the number of elements initially presented in the user interface and, as such, provides a more intuitive experience as users are gradually introduced to the full functionality of the application instead of every control being shown by default.

GPUscout result files can be selected using two different methods: users can manually select files using a system file picker or choose an entire directory. When a directory is selected, all relevant files inside it will be displayed as a list in the UI, allowing users to choose them by clicking on the corresponding entry, with this selection of a directory also persisting between sessions, and when set to, for example, the output directory of GPUscout, automatically displaying new files after they are generated. As performing changes based on recommendations provided by GPUscout-GUI often involves rapid iteration of many versions of the same kernel, the selected folder is also continuously monitored for new files, preventing users from needing to restart the application for new results to appear. Furthermore, a search input allows for the search for a particular result file in case of a large number of options.

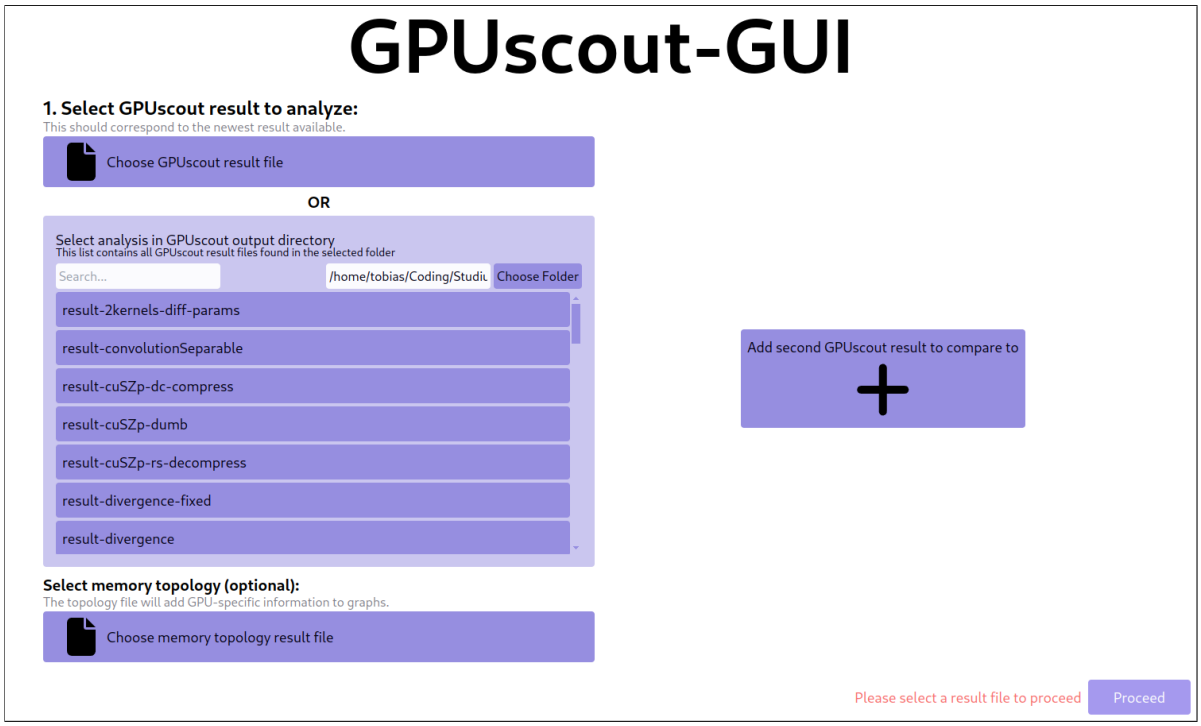


Figure 4.2.: Landing page

Additionally, two inputs are provided, allowing the selection of memory topology files, which are, however, not required to be able to analyze a result file. Even though, in most use cases, the same GPU will be used during the GPUscout analysis to avoid distortion of metrics, the ability to specify two distinct topologies in comparisons allows for special use cases, such as comparing identical kernels among various hardware. If only a single topology is provided, it will be automatically applied to all GPUscout results, eliminating the need for users to select the same file twice, which is clearly communicated in the hint provided for the relevant input option. Restrictions have also been implemented to ensure only files of the correct type can be chosen in all cases and to prevent accidental selection of invalid files that could lead to confusion due to error messages. After selecting all necessary files, users can switch to

the analysis page by clicking the *Proceed* button, which is only interactable after one or two distinct GPUscout result files have been selected. As the processing of GPUscout result files, especially those containing many kernels, can take a short moment, a loading screen informs users of the actions performed in the background.

4.4. Navigation

In any graphical application, the ability to easily navigate between different pages and clearly communicate the current state to the user are some of the most important factors contributing to a good user experience. In the context of GPUscout-GUI, this mainly corresponds to providing an intuitive way of obtaining an overview of all available kernels and their analyses and switching between them. Users should also never be in a state where they do not know which kernel or analysis is currently selected. To achieve this, two navigation components are displayed to the user at all times: a sidebar to the left and a header on top of every page, which are fixed in size and position to provide a consistent interface that users can easily understand. Their general layout and associated elements can be seen in figure 4.4.

While information visible in the header is static, always displaying the names of the current kernel and analysis, the content of the sidebar changes based on the results available for the current selection. Elements in the sidebar are organized to align with the intended workflow of first selecting a kernel, then reviewing all relevant analyses, and finally proceeding to the next kernel. Initially, the first kernel appearing in the GPUscout result file is selected, which can be changed through a drop-down menu. Because larger projects can contain many kernels, some of which GPUscout may not have found any results for, the drop-down menu displays the names of the available kernels along with information about the results found in them, with the number of analyses in which GPUscout identified at least one relevant code line, as well as the number of occurrences that contain potential performance bottlenecks being provided. Lastly, as not every kernel included in the source code of the analyzed binary has to have been executed, kernels with no associated metrics are grouped below all others to be able to easily identify the most relevant kernels. As shown in figure 4.3, this allows users to quickly determine which kernels are worth investigating further and which only contain information about already implemented changes or changes that may not lead to improvements. Kernels are then sorted by the number of relevant analyses, with kernels that yielded no result in any analysis being styled in a darker tone. A text input at the top also allows searching for a particular kernel.

After selecting a kernel, users are presented with a list of all analyses GPUscout found at least one occurrence in. While a simple list of analysis names suffices in the case of visualizing a single GPUscout result, analyses are categorized based on their presence in both results when comparing two different kernel versions. The following categories can occur:

- An analysis occurs in the current kernel version but not in the original one compared to. This indicates changes made between the two versions introduced new problems in this analysis.

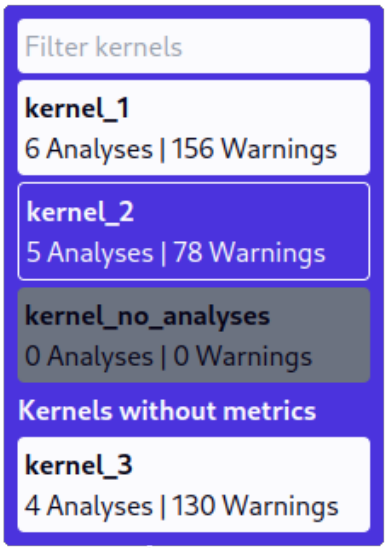


Figure 4.3.: Kernel selector component

- An analysis occurs in the original kernel version but not the current one. This indicates that changes implemented between the two versions fixed all the problems in this analysis.
- An analysis occurs in both the current and original kernel versions. No conclusions can be drawn in this case, as the number of occurrences in the analysis may or may not have changed, with at least one occurrence still found.

Distinguishing between these three categories when comparing two GPUscout results gives users a clear indication of the impact of implementation changes, particularly in relation to analyses where occurrences were initially found. On top of this, analyses that contain only informational occurrences but no optimization potential are marked in a darker color, more clearly indicating which analyses should be examined first. As it is also possible for two compared GPUscout results to differ in the set of available kernels, only the kernels of the current result are available for selection, as kernels only present in the older result are not considered relevant anymore.

Lastly, controls for changing the selected result files on the landing page and closing the application are provided at the bottom of the sidebar.

4.5. Analysis

The analysis page is the main screen of GPUscout-GUI and presents the results of a single analysis for the currently selected kernel. To prevent the need to design a separate interface for each of the nine analyses currently implemented in GPUscout and ensure future maintainability and scalability, a single layout has been developed, which can be configured to

accommodate the specific information of each individual analysis. By emphasizing the use of configuration files to define the structure of each analysis rather than altering the source code directly, the complexity of the application is also greatly reduced while simplifying the process of modifying or implementing analyses.

Figure 4.4 illustrates the general layout of the analysis page in GPUscout-GUI, which, apart from the header and navigation bar, that are always visible, is split into three main components, the content of which is independently and dynamically adjusted based on the currently selected analysis. The topmost section (annotation 1) is reserved for kernel-wide metrics, which are relevant to the current analysis and are especially useful in providing a general overview of the impact of implemented changes when comparing two kernel versions. The remaining components located in the lower part of the interface are responsible for presenting the kernel's code and further information about specific occurrences. Most of the space is reserved for the code view (annotation 2), which displays the source code of a kernel along with one of its intermediary representations while highlighting important lines and tokens. Based on the selection in the code view, further details are provided in the code info component next to it (annotation 3), which can include PC sampling stalls, details about a specific occurrence, or concrete recommendations for changes.

Figure 4.4.: Analysis page structure

The following sections will discuss each of these components, including their implementation, configuration, and the information they provide for each analysis.

4.5.1. Metrics

Regardless of the analysis, metrics are an important aspect in determining the gravity of a performance bottleneck in addition to information specific to an occurrence. Especially when comparing two different versions of the same kernel, the change in metrics is one of the main factors in deciding whether a particular optimization resulted in an improvement. Therefore, presenting them clearly and understandably without distracting from the rest of the user interface is crucial. One of the main challenges in designing the layout of the metrics component is finding a balance between the number of metrics and the level of detail they are displayed in. As the final application is intended to be used by both experienced as well as new users, who may not be familiar with every term or concept mentioned in the UI, a compromise needs to be found between displaying all the information necessary to understand and resolve a particular problem without overwhelming less experienced users, and not providing too much supporting information which may disrupt the workflow of more experienced users. Secondly, metrics should be able to be grouped into different sections, making it easier to distinguish between different classes of metrics, which not only enhances clarity and organization in the layout but also intuitively provides context, reducing the need for alternative explanations. Respecting these requirements, the component should not take up more than a third of the available space in the application, as the user's primary focus should be drawn to the code view component, and as much code as possible should be visible without the need for scrolling.

As a result, an option to collapse the metrics component has been added, allowing the entire window height to be used by the code view component. The decision has also been made to collapse this section by default when viewing a single GPUscout result since the workflow differs from that of comparing two kernel versions. When viewing a single GPUscout result, users are incentivized to investigate specific occurrences, which are highlighted and explained in the code view and info components first, with metrics usually only becoming relevant after this step. In the case of comparing two different versions of a kernel, however, metrics are the main indicator of the effects a particular change had and thus should be the first component users view after opening an analysis.

To easily separate metrics by category, rather than displaying them directly, sub-components are used to group related metrics together and provide a natural sense of association, also enabling choosing varying visualization modes based on the metrics in a category, with two different sub-components currently implemented. As mentioned before, while GPUscout will analyze every CUDA kernel found in the source code of the analyzed binary and thus visible in GPUscout-GUI, some kernels may not have been used during execution. If a kernel is not executed during the analysis, no metrics are collected, and a corresponding warning is displayed instead of the metrics component.

Metrics List

The list sub-component organizes metrics of a common group in a grid with the ability to include supporting information for each individual metric. As the component should take

up as little vertical space as possible, metrics are arranged horizontally by default, with the component only growing in height after the number of metrics displayed exceeds the component's width. Compared to a conventional list, this allows each metric to occupy more space in both directions while taking up less total height in the user interface. Each instance of this component displays a configurable title and hint, informing the user of the kind of information conveyed by the metrics included in them.
As discussed in section 4.2.3, this component is defined during the configuration of the associated analysis, with an example configuration of a simple metrics list being provided in listing 4.6. After specifying the aforementioned title and hint, as well as the type of the component, a list of metrics can be provided, which will then be displayed to the user. Metrics in this list are defined using three different attributes:

name The name of a metric is the only required attribute and corresponds to the unique internal name of the metric as defined in the respective configuration file. This allows their attributes to be inherited and not need to be defined again.

value Even though the value of a metric is automatically fetched based on its `name`, there are cases in which a metric has to be computed either from multiple different metrics or different attributes of relevant analyses. In this case, an additional attribute `value` can be passed to each metric, computing a value based on the current analysis object, which provides access to the required data. This is used, for example, for stall metrics, which are only available as a percentage of the total amount of stalls in the GPUscout result. To calculate the absolute values of these metrics, another metric containing the total number of stalls needs to be used.

secondary_value Keeping with the example of stalls, displaying both the relative and absolute value can be beneficial, as information could get misinterpreted or lost if one of them is omitted. An example of this is high relative stall percentages, which are only an indicator of a performance bottleneck in case the associated absolute values are also moderately high. Because of this, a second value can be specified, which will be displayed in parentheses after the main value of the metric.

```
1 {
2     "type": METRIC_SECTION_TYPE.LIST,
3     "title": "Warp stall analysis",
4     "hint": "Occurrences of relevant warp stalls in the kernel",
5     "metrics": [{
6             "name": "stalls_total"
7         }, {
8             "name": "stalls_short_scoreboard_perc",
9             "value": (analysis) => (analysis.getMetric("
                  ↪ stalls_short_scoreboard_perc") * analysis.getMetric("
                  ↪ stalls_total")) / 100,
```

```
10            "secondary_value": (analysis) => analysis.getMetric("
                  ↪ stalls_short_scoreboard_perc")
11        }, ...
12    ]
13 }
```

Listing 4.6: Application code: Configuration of metric list component

When comparing two kernel versions, the `value` is always used to compare metrics, as the `secondary_value` should only be used for supporting information and not information determining the performance impact of a metric. Because of this, if a metric has both absolute and relative values, the absolute value should always be displayed first. Lastly, a help icon will be displayed for each metric with a defined `help_text`, which, when clicked, will be displayed in a popup for users to read. This is targeted especially for newer users in case they require more information about a particular metric while not cluttering the user interface, which only directly displays relevant information.

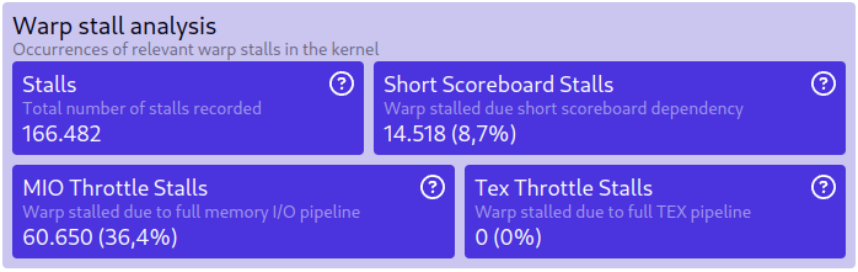


Figure 4.5.: Metrics list component

Memory Graph

In some cases, simply listing all metrics within a category is not ideal, for example, when dealing with a large number of metrics or when the relationship between them should be emphasized. In some analyses, GPUscout already provides users with several memory-related metrics, enabling a basic overview of the data flows between different parts of the GPU memory system during kernel execution. An appropriate mode of display for this kind of information would be a graph-like manner that mirrors the internal memory representation of a GPU, comparable to the memory charts available in NVIDIA's NSight Compute, which are described in section 3.3. An implementation of this in GPUscout-GUI can be seen in figure 4.7, which displays the data flows occurring when loading from global or local memory and is displayed, for example, in the register spilling analysis. Compared to the output of GPUscout, metric values in memory graphs are formatted according to their type: Rather than using bytes as the sole unit for data flow metrics, higher-order units like kilo- and megabytes are used when appropriate, improving readability as values are shorter and easier to compare. Additionally, cache hit rates are displayed instead of miss rates, as they provide a more intuitive way to trace changes in data streams between caches. These graphs also

serve as a place to display additional information provided by a GPU topology result, with additional information for memory components like the L1 and L2 cache or DRAM shown on the appropriate nodes.

For a memory graph to be displayed in an analysis, a corresponding configuration entry pointing to an instance of the `MemoryGraph` class needs to be made. To enable the desired modular creation of graphs, multiple classes are utilized, as shown in figure 4.6.

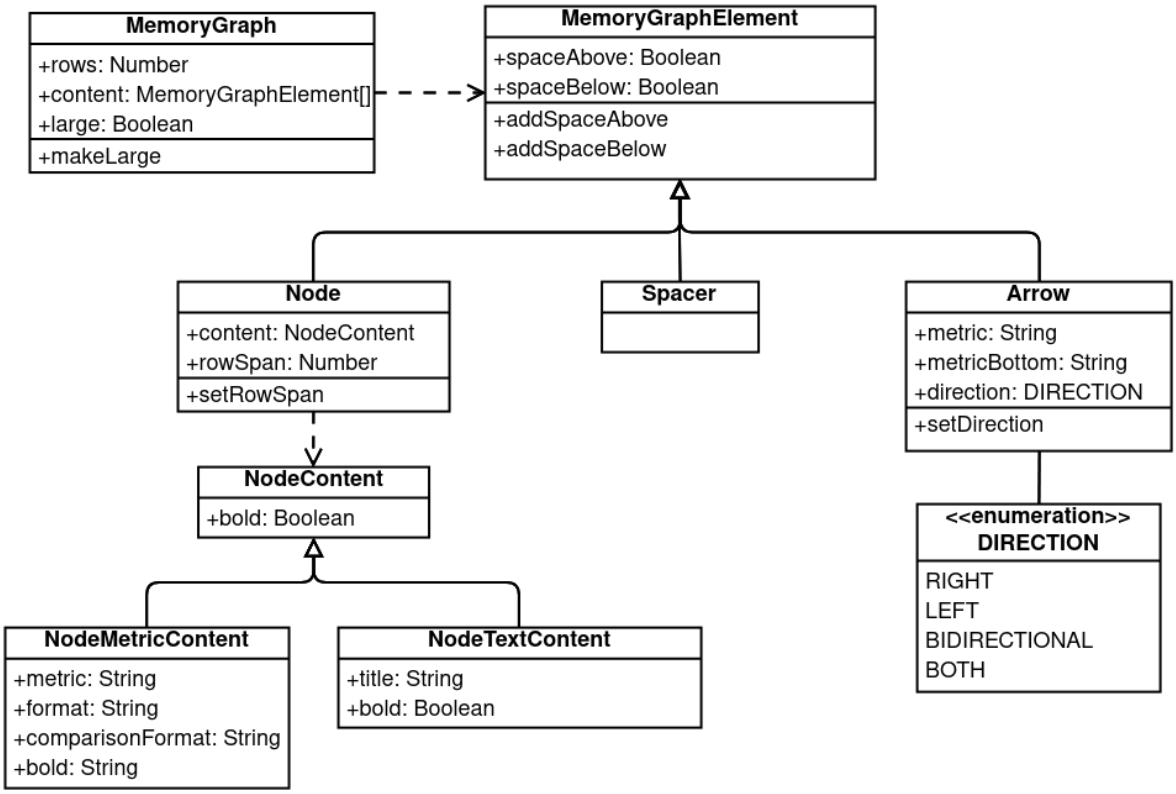


Figure 4.6.: Class diagram: Memory graph

When creating a new `MemoryGraph`, at least two arguments must be provided, the first of which defines the number of rows this graph will span, including vertical space for nodes and arrows, as well as gaps between them. The remaining arguments consist of arrays defining the individual columns that make up the graph, which are ordered from left to right. Each column contains instances of type `MemoryGraphElement` representing its respective items, with the example above configured to have three rows and nine columns. The following elements can be used in the layout of a row in the graph:

Node Nodes, as their name suggests, represent graph nodes that display the various memory components and contain one or more labels inside them. To make the kind and amount of data displayed on each node as customizable as possible, rather than passing the text directly, instances of `NodeContent` are used to define the content inside a node,

which can vary based on the implementation, with two different subclasses currently available. To display static text, an instance of `NodeTextContent` can be used, whereas `NodeMetricContent` implements functionality to display the value of a metric in a predefined format. Additionally, the text of each element can be displayed in bold to emphasize headers or important information. If more than one instance of `NodeContent` is passed to a node, they get listed from top to bottom, with the vertical space of the parent node being split equally among them.

Arrow Arrows can be used to connect two nodes while displaying one or two metrics alongside them, the first metric provided being displayed above the arrow, with the second one, if present, shown below it. This is important to recognize, as the position of the text and the direction of the arrow can implicitly be used to convey information about the type of data stream the arrow represents. Four different styles of arrows allow the modeling of different kinds of data streams: one-directional arrows pointing to the left or right should be used to indicate data part of a load or store operation, with bidirectional arrows being used when the metric is an aggregate of the two. In the case of two metrics shown on top of and below a single arrow, a direction of `BOTH` will render two individual arrows pointing in separate directions, allowing the display of both load and store information on a single arrow.

Spacer Although each `MemoryGraphElement` implements two functions `addSpaceAbove` and `addSpaceBelow` for adding a single space above or below the element, in some cases, more space is needed. For this purpose, an instance of `Spacer` can be used to add intentional blank elements to the graph.

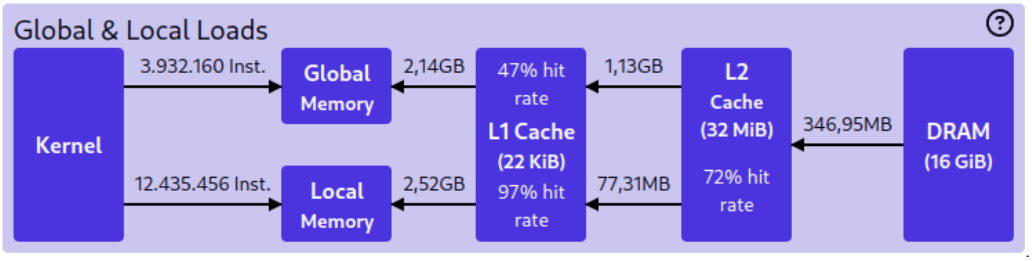
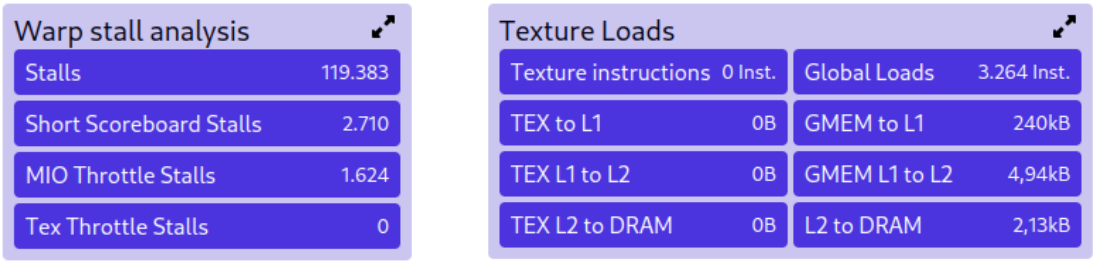


Figure 4.7.: Memory graph component

In addition to the individual memory graphs associated with certain analyses, which usually focus on data movements of a particular part of the memory system of a GPU, a larger graph displaying the data flows between all major memory components has been created and can be accessed in a popup from every analysis. This not only enables detailed insights into the functionality of a particular kernel but provides the opportunity for inexperienced users to gain a better understanding of how certain memory components function, as in-depth explanations for every node of the graph are provided. Clicking on the help icon of any memory graph in any analysis will also present this popup.

Metrics Components Arrangement

Although displaying metrics in the provided sub-components solves the problem of offering the appropriate amount of information in an intuitive and easy-to-understand way for all target groups, especially in analyses with multiple sub-components positioned next to each other, a lot of horizontal space is needed, which leads to users needing to scroll to be able to view all metrics. To reduce the amount of space taken up while not compromising on the amount or readability of the information presented, reduced versions of each sub-component have been implemented, which can be seen in figures 4.8a and 4.8b. If an analysis contains more than one sub-component for metrics, only a single one is displayed in full size and detail at a time, with the remaining components being collapsed into their more compact versions. Compared to their full-sized form, additional information like secondary metric values and hints are omitted while preserving the most important information in the metric name and value. Clicking on the expand icon on the top right of a collapsed list promotes this component to its full version, in turn collapsing the currently expanded one, allowing for maximum flexibility and space efficiency while not compromising on the amount of information presented in the user interface.



(a) Metrics list

(b) Memory graph

Figure 4.8.: Collapsed metrics components

Metrics in Comparisons

As previously mentioned, metrics are an important factor of evaluating implementation changes between two kernel versions, making intuitively communicating the differences in the values of metrics in both sub-components crucial. In the case of the metrics list component, the value of each metric is accompanied by the value of the same metric in the compared GPUscout result. Following the natural reading direction of left to right, the value of the comparison kernel, which in most cases represents an older state, is displayed on the left, with the current value on the right. Between them, a third value indicates the difference between the two metrics, displayed in either red, green or the regular text color accompanied by an arrow pointing up or down to be able to quickly identify the metrics that improved or declined. For metrics with secondary values (as explained in section 4.5.1), the first value is always used for comparison as it is deemed more important and otherwise could lead to

wrong conclusions. An example of this would be one of the compared kernels having lower absolute but higher relative values in long scoreboard stalls. This can happen if the total number of stalls declined by a large margin, while the number of long scoreboard stalls only decreased by a small amount. If the relative value were to be used for comparison, the change would be considered bad, while the number of both total and long scoreboard stalls declined.

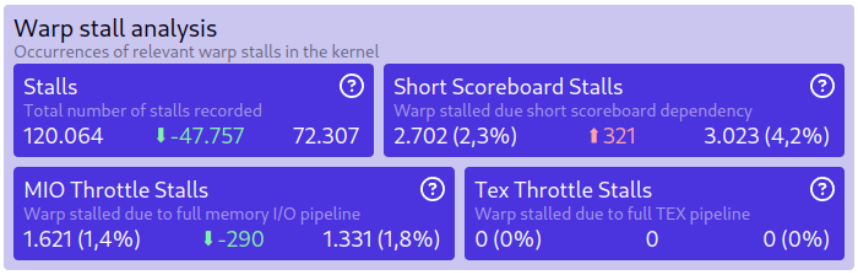


Figure 4.9.: Metrics list component during comparison

In the case of memory graphs, each metric value displayed on a graph node is displayed similarly to the metrics in the metrics list, with the exception that no additional value indicating the difference between both values is displayed, as this would take up too much space. Instead, the metric value corresponding to the current kernel version is colored in either red or green in case of varying values, helping users identify the current metric more easily and indicating changes even in cases where the loss of precision due to rounding resulted in otherwise identical values. For metrics displayed alongside arrows, two cases need to be handled: if only a single metric is displayed alongside an arrow, the older metric value from the comparison GPUscout result is displayed on top, with the current value on the bottom of the arrow, again respecting the natural reading direction from top to bottom. In case two metrics are already present on an arrow, however, the different values are displayed next to each other in the format *old value vs. new value* instead. Similarly to other components, the newer value is always colored according to the change, allowing users to quickly identify which value corresponds to the current kernel version.

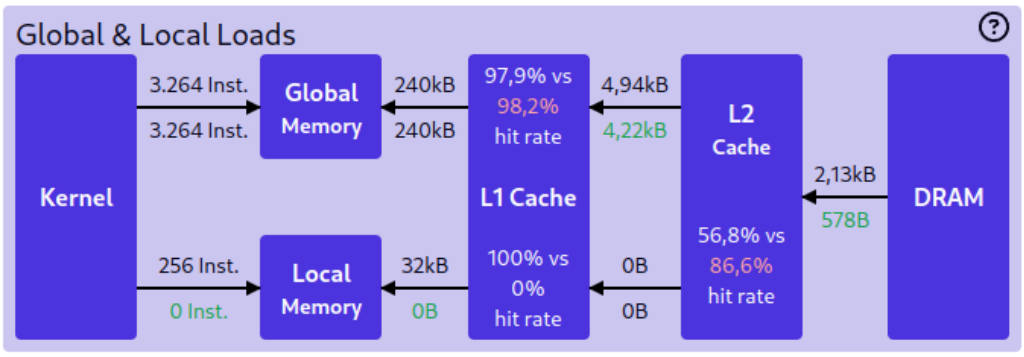


Figure 4.10.: Memory graph component during comparison

Lastly, in the collapsed versions of both sub-components, values are formatted similarly to graph arrows in the format *old value vs. new value* and colored based on their change. Coloring is especially important in this case, as the dense display of a large amount of data in a small area can make it difficult to otherwise determine changes, for example in the case of large metric values only differing slightly between kernel versions. Figure 4.11 shows the collapsed memory graph in the *Use texture* analysis, which, even though in its more compact collapsed version, still effectively communicates the change in global and texture memory data flows.

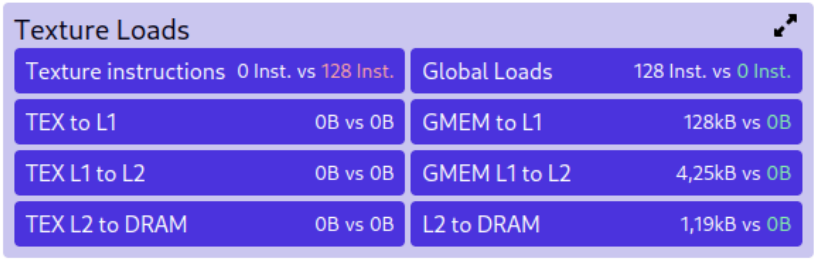


Figure 4.11.: Collapsed Memory graph component during comparison

4.5.2. Code View

The code view represents the central and most important component of the analysis page, presenting the kernel code along with extensive supporting information. While the kernel's source code is always displayed, the intermediate representation it is compared to is dynamically determined based on whether the selected analysis relies on SASS or PTX code.

Colors and Highlighting

Important information and the current state of the code view are primarily conveyed through variations in styles and colors applied to specific parts of the code. By using distinct colors and styles, users can quickly identify related elements and distinguish different purposes, all while preserving code readability in a non-intrusive way. The choice of colors in the code view also played a key role in determining the overall color scheme of the user interface, ensuring sufficient contrast in all areas of the application for optimal readability. Two shades of purple were chosen as the primary colors due to their strong contrast with red and green, the most important supporting colors in the user interface, which are heavily used not only in the code view but also in the metrics section during comparisons. Additionally, purple offers a warmer alternative to gray, another widely used color scheme that was specifically avoided as a background color due to its poor contrast with black, designated as the primary text color. All colors used in highlighting were carefully selected and can be configured to ensure strong contrast with both the background and text.

Three distinct styles are utilized in the code view to communicate different types of information about the meaning of a line or specific parts of it: Borders around entire

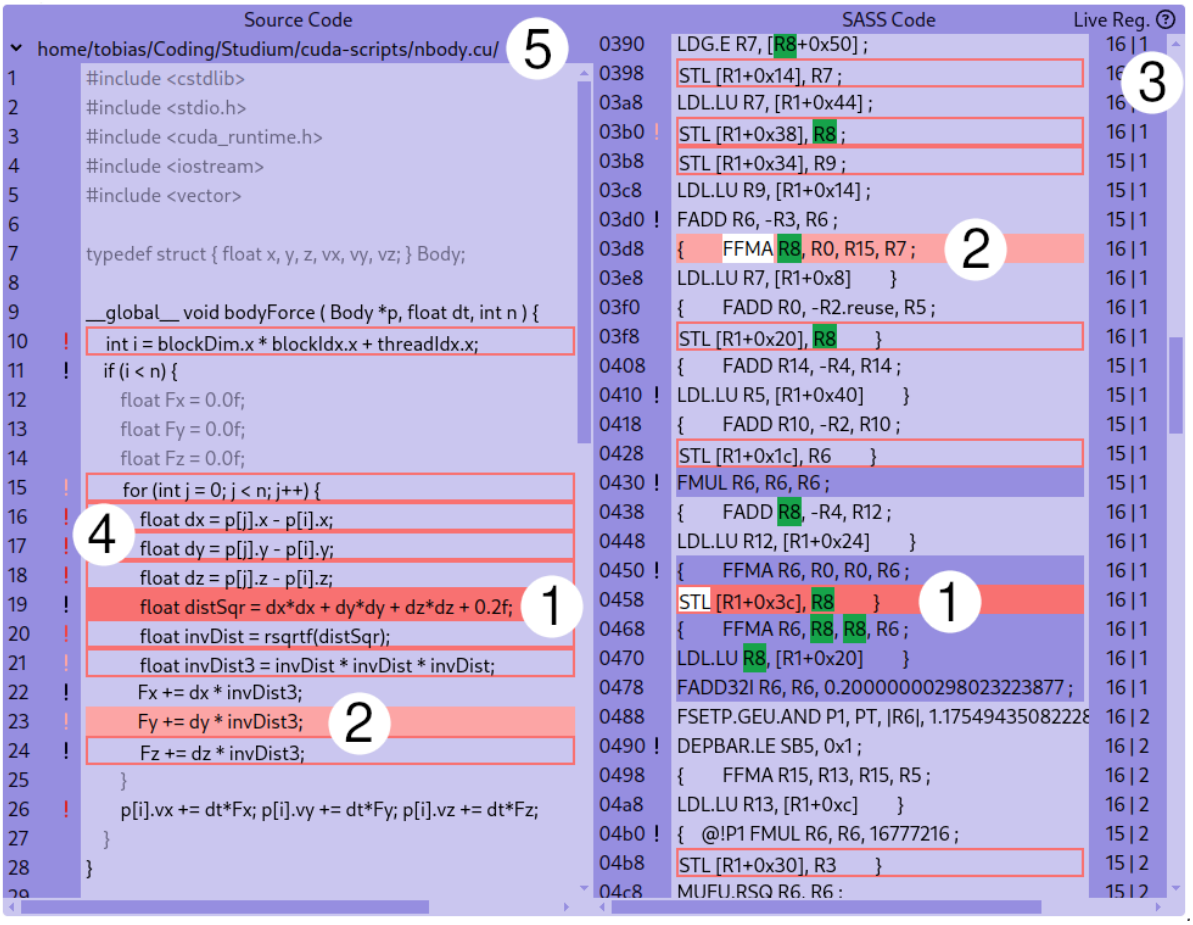


Figure 4.12.: Code view component

lines, as well as differently colored backgrounds for either code lines or a subset of their tokens, with borders in the source and intermediary code pointing to lines related to specific occurrences identified by GPUscout being the only highlights visible upon opening an analysis. Depending on their color, users can quickly distinguish between occurrences indicating potential performance bottlenecks and those solely providing informational context, with warnings being styled in red color, while information occurrences are colored blue. In contrast to other forms of highlighting, which are only active when a particular code line has been selected, these borders are always displayed to the user, as the location of other bottlenecks can be an important piece of information even when currently investigating a different occurrence.

The second highlighting style modifies the background color of a specific line in either the source or intermediary SASS and PTX code and is used to emphasize the currently selected line(s), as well as other important code lines related to any selected occurrences. Depending on the information available and the type of a particular code line, the following steps are taken:

Lines that are not occurrences When selecting a code line that is not marked as an occurrence using a border style, the corresponding line(s) in the other code type should also be highlighted. In the case of selecting a line in one of the intermediary representations, this always translates to a single specific line in the source code, but one source code line can translate to multiple lines of PTX and SASS code or no line at all.
As intermediary code often contains significantly more lines than the associated source code, users may have difficulties finding newly highlighted lines within the code component's limited height. To address this, the container automatically scrolls to the first highlighted line upon selection, making it easier for users to locate the mapped code lines. This enables users to get a better understanding of the syntax of intermediary representations, as well as how certain functionality is executed on the device, which can be useful even in the case no bottlenecks have been found in a particular code section.

Occurrence lines Upon selecting a code line that corresponds to an occurrence and is thus marked by either a blue or red border, the underlying occurrence(s) get selected. If this selection was made in one of the intermediary representations, the selected line corresponds to a unique occurrence, and further highlighting can be made based on it. Even though each occurrence can be uniquely identified using a single SASS or PTX line, depending on the analysis, additional lines of the intermediary representation can be relevant to understanding the gravity of a particular performance bottleneck. An example of this is the register spilling analysis, where, in addition to the main SASS line a spill occurs in, the line containing the previous compute instruction of the relevant register can also provide important context for understanding the cause and more easily perform optimizations for the particular bottleneck. Because of this, this additional line is also highlighted in a lighter tone than the primary occurrence line. Both of these highlightings are also translated to the source code of the kernel to be able to identify the relevant code sections more easily.
In the case of selecting an occurrence line in the source code, two different possibilities need to be handled: if the particular source code line maps to exactly one intermediary line that corresponds to an occurrence, highlighting is performed as if this SASS or PTX line was selected, with the addition of also highlighting all mapped code lines as described above. However, a selected source line may correspond to more than one code line in the intermediary representation that contains an occurrence. In this case, only the primary line of each occurrence gets highlighted in order to provide an overview of all relevant occurrences and not confuse users with an excessive amount of highlighting, with further highlights becoming visible only after selecting one of those occurrences in the SASS or PTX code.
A practical example of this can be seen in figure 4.12, where a source code line corresponding to a single occurrence has been selected. In addition to the mapped lines, annotations 1 and 2 correspond to the primary and secondary lines highlighted in both code representations.

Lastly, in addition to entire lines of code, individual tokens can be highlighted to provide

further insights into the specific circumstances of an occurrence. This can be particularly useful for highlighting a relevant register, not only in the line of an occurrence but also across all other code lines, allowing users to easily trace where problematic registers are used and helping them identify the origin of an issue more easily. Similar to the procedure described for entire lines, these highlights are only visible if a single occurrence is selected by the user and currently only implemented for the intermediary code representations, as the level of detail these tokens grant cannot be easily translated to source code. In addition to the name of the instruction, which is highlighted for each relevant line, an overview of the highlighting provided for each individual occurrence based on the selected analysis can be seen in the following table:

Analysis	Highlighted Lines	Highlighted Tokens
Datatype Conversion	-	Both registers involved in the conversion
Global Atomics	-	All registers involved in the atomic operation
Register Spilling	SASS line with previous compute instruction of spilled register	The register that spilled
Use Restrict	-	The relevant register
Use Shared	SASS lines with global load instructions or computation instructions involving the relevant register	The relevant register
Use Texture	SASS lines with reads from the same base address, if spatial locality is given	The written register, as well as the register and unroll of the load address
Vectorization	SASS lines with reads from the same base address	The written register, as well as the register and unroll of the load address
Warp Divergence	-	The target branch name

Table 4.1.: Code view: Configured highlighting per analysis

For every analysis, relevant registers of the instruction executed in the primary SASS or PTX code line are highlighted among the entire kernel, allowing users to trace their usage before and after the problematic code lines. While the number of highlighted tokens is kept to a minimum to avoid compromising the readability of the code, in some analyses, additional tokens are crucial and therefore highlighted as well: In the *Warp Divergence* analysis, the name of the target branch is highlighted, with both the *Vectorization* and *Use Texture* analyses emphasizing the unroll added to a register to compute the a load address. As previously mentioned, in the *Register Spilling* analysis, the SASS line corresponding to the previous compute instruction of the spilled register is highlighted, if available. Additionally, the *Use Shared* analysis highlights all lines where the relevant register is involved in global loads or computation instructions, as this plays a key role in determining the effectiveness of proposed

changes. In the *Use Texture* and *Vectorization* analyses, all lines corresponding to reads from the same base address as in the primary line of the occurrence are emphasized, but in the case of *Use Texture* only when spatial locality has been found.

To further enhance the user experience, source lines that do not map to any line in the currently relevant intermediary representation are colored in a lighter shade, helping users more easily identify relevant source lines and preventing confusion when selecting a line with no corresponding mapping. Additionally, this can provide insights into certain optimizations performed by the compiler, such as when certain variables are inlined or code is reorganized for improved performance, helping users understand the impact of these optimizations on the overall code structure.

Additional Information

While the highlighting already provides significant insights into the results of GPUscout, some additional information that is available and depends on specific code lines should also be integrated into the interface to offer a more comprehensive view. In analyses based on the SASS intermediary representation, each code line can be annotated with the number of live registers used at that point in the execution of the kernel, providing valuable insights, especially in analyses where register usage is important. An example of this can be seen in figure 4.12, which displays the register spilling analysis in a kernel limited to 16 registers. To the right of every SASS line (annotation 3), the number of live registers can be seen, which is split into two categories, namely the number of used general-purpose registers followed by predicate registers.

Apart from kernel-wide metrics, stall samples collected during PC sampling are essential for determining the actual impact of a particular bottleneck, making it crucial to clearly communicate this information in the code view. As can be seen in annotation 4 of the figure above, exclamation mark icons of different colors are used to indicate the presence of stall samples in a particular code line. To be able to provide users with an intuitive way of assessing the impact a specific source code line has on the performance of a kernel, stall samples, which are collected for every SASS line, are translated to source code lines by aggregating the samples of all SASS lines by the source code line they map to. As especially in large kernels, locating which sections of the source or SASS code are most problematic can be difficult, these icons are colored differently based on the amount of stall samples in a particular line. While icons in lines with less than 2.5% of all recorded samples retain the normal text color, shades of red are used to indicate the presence of more than 2.5% of all collected samples, with darker variants of the color being used in increments of 2.5% up to 10%. This approach enables users to quickly identify hotspots in a kernel's source and SASS code without manually searching through all the indicated lines.

Comparison

Even though the change in metrics and the number of problems found by GPUscout are the most important factors to consider when comparing two different kernel versions, changes

in the source and intermediary code can also provide helpful information. Because of this, an option allowing users to toggle between displaying the current and older version of the kernel code is provided. Upon selecting the older kernel, all highlighting and otherwise displayed information in both the code view and code info components are adapted to match the old kernel, allowing a detailed comparison of the generated code, as well as changes in the location and details of certain occurrences, the number of live registers, or stall samples.

Handling multiple Source Files

As described in section 4.2.2, in contrast to the intermediary PTX and SASS code, which is aggregated in a single file for all kernels, multiple source files can be involved in generating mappings of each of their code lines to actual source code lines. During the development of GPUscout-GUI, two different techniques for integrating this information in the code info component were considered: while the first option would simply concatenate all available source files to form a single large file, the second option presents a single source file at a time, with users needing to manually switch between files if desired. After evaluating both, the second option has been chosen and implemented, providing a more user-friendly and intuitive experience. Especially in larger projects, it is not uncommon for individual CUDA source files to span more than 1000 lines of code, making finding a specific part of a certain file very tedious if multiple files were concatenated. As can be seen in annotation 5 of figure 4.12, a dropdown menu has been implemented, which informs users of the full path of the file currently visible and allows switching between them. Upon selecting a line in an intermediary representation that maps to a line in a source file other than the currently visible one, the relevant file is automatically displayed and the relevant line scrolled to, with no manual intervention required by the user.

Even though displaying only a single file at a time reduces the number of code lines visible in the UI, viewing kernels with large SASS, PTX, or source files still significantly impacts the application's rendering performance. Because of this, a technique called list virtualization is used to render only the visible part of the code of a file, with the corresponding code snipped displayed in figure 4.7.

```
1 <div
2     v-for="index in Math.min(150, relevantLines.length - firstVisibleLine)"
3     :key="relevantLines[index + firstVisibleLine - 1]"
4     class="absolute w-full"
5     :style="{ top: (index - 1 + firstVisibleLine) * 1.5 + 'rem' }"
6 >
7     ...
8 </div>
```

Listing 4.7: Application code: List virtualization

The `div` HTML element represents a single code line in one of the two parts of the code view. Instead of iterating over all code lines of a file, which are stored in `relevantLines`, only a maximum of 150 lines of code are rendered at a time, depending on the scroll position

inside the container. To always display the correct section of the code, the `firstVisibleLine` attribute gets updated whenever a scroll event occurs within the code view container, corresponding to the index of the code line of the topmost visible code line, with a minimum padding maintained to prevent users from noticing abrupt changes during fast scrolling. This greatly reduces the performance impact of large files on the UI while not negatively impacting user experience in any way.

4.5.3. Code Info

The code info component displays additional information based on the current selection in the code view. As shown in figure 4.13, the component is split into multiple sections to increase readability and distinguish different kinds of information. If no line in the code view is selected or no information is available for the current selection, a hint is displayed to inform users of this and provide basic guidance on the purpose of differently styled code lines, ensuring that users are always informed about the current state of the selection and the steps needed to be taken to select relevant lines of code. The content displayed when one or more relevant code lines have been selected can be split into two categories: results from PC sampling and further information about currently selected occurrences. Sampling information is always displayed at the top of the component, followed by the remaining sections, as the number and type of stall samples in a particular line are important pieces of information that should be considered for every occurrence and thus be read first. Two controls are also included at the bottom, which, when clicked, will automatically scroll the code view to the previous or next occurrence in the currently selected code type, avoiding the necessity of manually searching through the code to find each occurrence identified by GPUscout. As these buttons are only active if a previous or next occurrence exists, they also provide helpful information in determining if the code before or after a particular occurrence contains further findings.

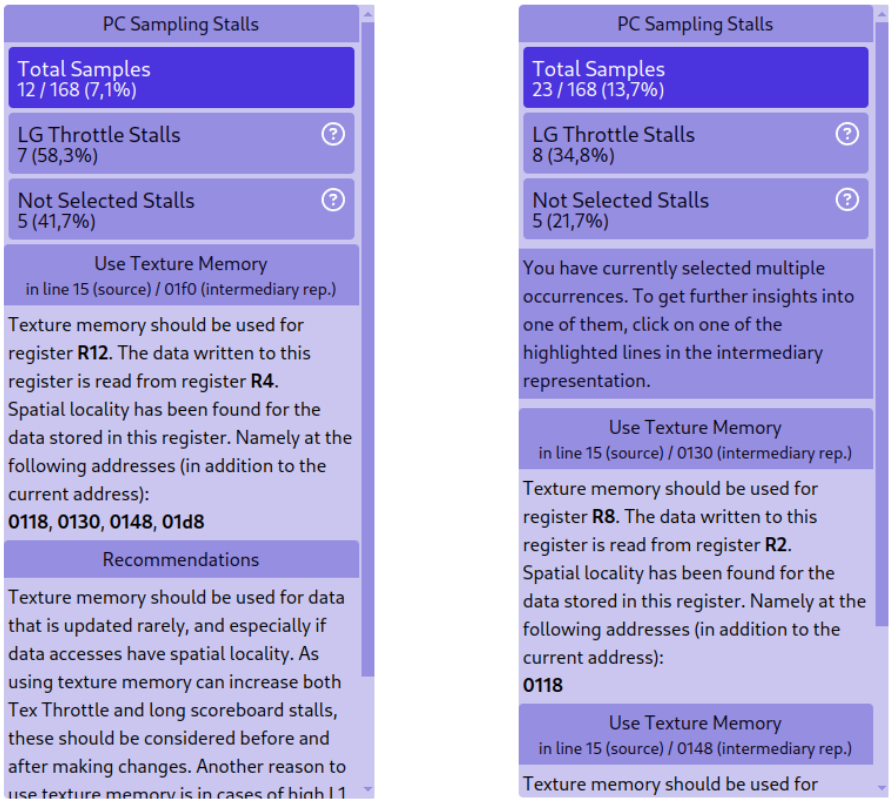
Occurrence Info

If the current selection in the code view includes one or more occurrences, further information about them, which can be configured for each analysis separately, is displayed to the user. The title and description of this section provide a summary of the nature of the occurrence, including, but not limited to, the information currently provided in GPUscout, as well as its exact location in both the source and intermediary code. Additionally, recommendations to help evaluate the gravity of the problem, guidance for the next steps, and important metrics are provided in a separate section below. As these recommendations can be extensive, they are only displayed if a single occurrence has been selected and hidden otherwise to avoid overwhelming users with too much text. By splitting this content into two distinct sections, users are presented with the most important details first, with more information about the cause of a problem and potential fixes displayed separately, allowing more in-depth information to be displayed than currently available in GPUscout while also presenting it in a more readable and user-friendly manner. An example of this can be seen in figure 4.13a,

where on top of general information about using texture memory instead of global memory, like relevant registers and addresses in the SASS code, a more detailed explanation of when to use texture memory, as well as the expected impact of the change on certain metrics is provided.

Stall Samples

In contrast to the metrics displayed in the upper section of the user interface, which are relevant in the context of the entire kernel, PC sampling stalls are specific to a single line of SASS code. When a line marked to contain stalls is selected, a list of all recorded stalls is provided in the code info component, with each entry specifying both the number of stalls recorded as well as the percentage relative to the total amount. Additionally, the percentage of stalls occurring in this line in relation to the total number of samples collected in the kernel is provided. Upon selecting a line in the source code, which maps to more than one line in the intermediary SASS code with PC sampling stalls, the aggregated number of stalls for all mapped SASS lines is presented instead.



(a) Single occurrence

(b) Multiple occurrences

Figure 4.13.: Code info component

4.6. Summary

As it is not guaranteed that GPUscout will find performance bottlenecks in every analyzed binary, the case of viewing a result file with no analyses that contain any occurrences to display needs to be handled. Due to the increased amount of information provided, some users may also want to use GPUscout-GUI to analyze the code and metrics of their kernels, even if no concrete evidence for a particular bottleneck is available. The most common case, however, occurs when analyzing a GPUscout result with multiple kernels, some of which do not have any analysis results. In these circumstances, a special summary page is displayed, which allows users to toggle between and explore the source code, as well as both intermediary representations, the full memory graph, and a predefined set of metrics. Namely, the total and individual number of stalls occurring in the kernel, as well as the occupancy and total number of instructions executed, are displayed, as especially the occurrence of stalls can be very informative even in cases of no concrete optimization potential. The ability to view and inspect code lines in which PC sampling stalls have been recorded can also provide valuable information in these cases. Similar to the regular analysis page, users can also switch between two kernel versions in comparisons, in case both kernels have no analysis results, and this page is displayed.

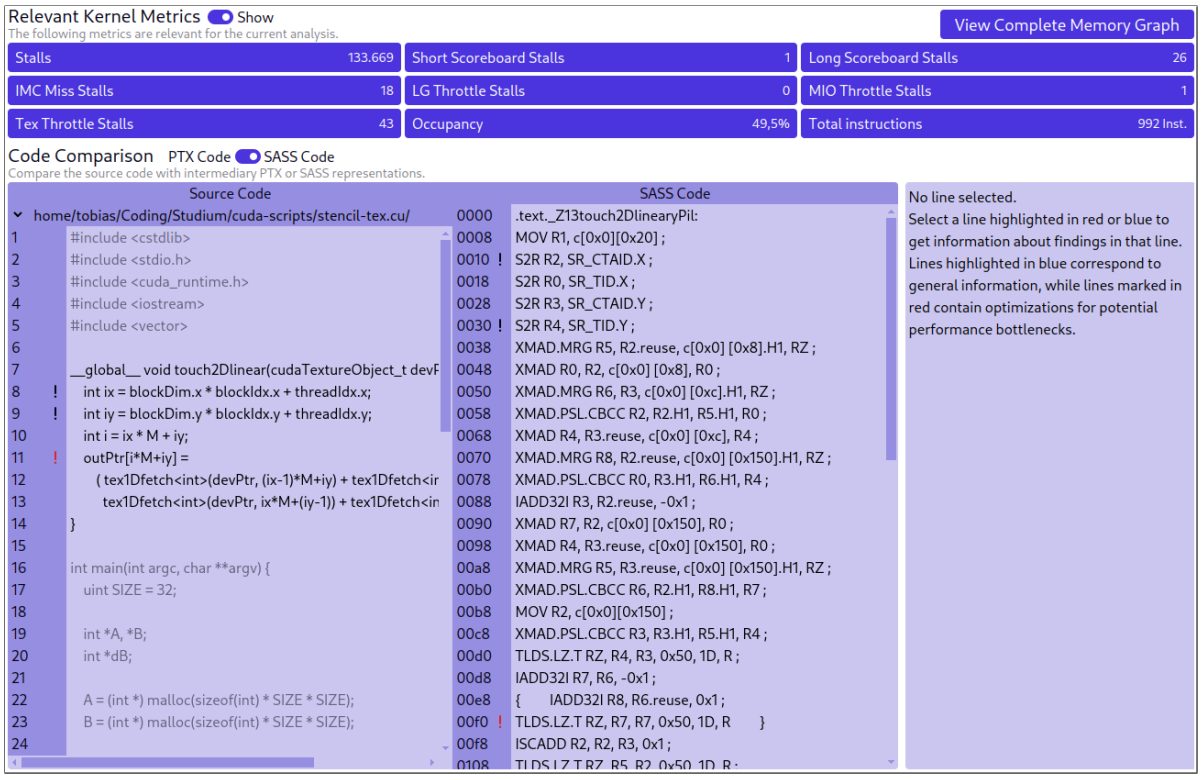


Figure 4.14.: Summary page

5. Evaluation

To assess the accuracy of the results displayed in GPUscout-GUI and highlight the benefits of visualizing data graphically, three different CUDA projects were analyzed using GPUscout, with the results subsequently visualized in GPUscout-GUI. This chapter examines the findings and explores the role of GPUscout-GUI in presenting these results, as well as its impact on decision-making compared to the output provided by GPUscout. Rather than selecting projects based on the optimization potential identified by GPUscout, three projects with varying complexity and structure were chosen, emphasizing how both tools process and present their results to users rather than focusing solely on the performance improvements gained.

5.1. Hardware Used

All following projects were analyzed using two different GPUs, namely NVIDIA A100[1] and RTX 4080[2], with the NVIDIA A100 being accessed using the CAPS cloud [3]. CAPS cloud is a set of servers made available for research purposes, with the *zen1* server being used to access the NVIDIA A100 GPU. A summary of their specifications, as well as the versions of relevant tools, can be seen in the following table:

Property	NVIDIA A100	NVIDIA RTX 4080
Architecture	Ampere	Ada Lovelace
Compute capability	8.0	8.9
CUDA Version	12.5	12.5
Streaming multiprocessors	108	76
CUDA Cores	6912	9728
Global memory	40GB	16GB
Max. shared memory per block	48kB	48kB
Peak FP32 Performance (non-Tensor)	19.5 TFLOPS	48.7 TFLOPS
Peak FP64 Performance (non-Tensor)	9.7 TFLOPS	0.95 TFLOPS

Table 5.1.: NVIDIA A100 and RTX 4080 properties

While the evaluations of all tested kernels were performed mainly using the NVIDIA A100 GPU, as this GPU is more commonly used and optimized for heavy data-center workloads,

[1] https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[2] https://images.nvidia.com/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf

[3] https://www.ce.cit.tum.de/caps/hw/caps-cloud/

assessing the results using a second GPU provides valuable insight into how GPUscout and GPUscout-GUI handle the analysis and comparison of kernels between different hardware. Evaluating the performance impact of the implemented changes on both GPUs also enables insights into their effectiveness across multiple architectures and varying GPU performance levels, with all performance measurements and visualizations in GPUscout-GUI mentioned in the following sections having been created using the NVIDIA A100 GPU, unless stated otherwise.

Changes made to GPUscout

While performing GPUscout analyses on both mentioned graphics cards, some inconsistencies between the actual and expected results, along with occasional crashes, were observed. The causes of these issues and their potential mitigations will now be discussed.

Since the initial development of the GPUscout application between the second half of 2022 and the first half of 2023, NVIDIA released several new versions of their CUDA toolkit. During at least one of these updates, the format of the CSV output generated by NVIDIA NSight Compute CLI changed, which resulted in GPUscout being unable to correctly parse the metrics extracted by the tool as the indices of the relevant columns were no longer valid. To resolve this issue, the code responsible for parsing the metrics file has been altered to dynamically compute the indices of the desired columns instead of relying on hardcoded values.

```
1 int metric_name_index = 19;
2 int metric_value_index = 23;
3 ...
4 while (std::getline(header, word, '"'))
5 {
6     if (word == "Metric Name") {
7         metric_name_index = word_index;
8     } else if (word == "Metric Value") {
9         metric_value_index = word_index;
10    }
11    word_index++;
12 }
```

Listing 5.1: GPUscout code change: Metrics file parsing

In the code snippet shown, the header line of the generated CSV file gets split into individual columns, with the indices of the metric name and value column being stored.

As apparent from the data presented in table 5.1, the GPUs used to evaluate kernels in this thesis differ in their number of streaming multiprocessors, which is significant because the SM count of the GPU used during the GPUscout analysis influences the calculation of certain metrics, such as the percentage of L2 queries issued due to local memory accesses, which is used in the register spilling analysis. This leads to incorrect metric values if the SM count of the used GPU does not correspond to the value used in GPUscout. Instead of relying

on a fixed parameter embedded in the source code, a new parameter `--sm_count` has been implemented, which allows specifying the number of SMs of the currently used GPU and avoids incorrectly calculating some metrics.

Lastly, another issue emerged when accessing the NVIDIA A100 GPU from the CAPS cloud, specifically during the parsing of the CSV metrics file generated by NVIDIA's NSight Compute CLI, caused by differences in the configured system locale. Compared to the system the NVIDIA RTX 4080 GPU was used on, the locale of the *zen1* server is set to American English instead of German, which affects the formatting of numbers in the CSV file containing the metrics. While numbers formatted according to the German locale use dots as the thousand separator and commas as the marker for decimals, their roles are reversed in the American English locale. This leads to GPUscout, which expects the German number format, parsing the value of metrics incorrectly, interpreting, for example, the number *1.000,75* as *1,00075*, which causes significant differences in how certain metrics are interpreted. At the time of writing, this problem has only been mitigated locally, with a more flexible solution needing to be implemented in GPUscout.

5.2. CuSZp

GPUs are highly efficient at executing repetitive arithmetic operations across large datasets in parallel, utilizing thousands of cores to achieve exceptional computational throughput. While they are traditionally used in fields like machine learning and scientific simulations, there is a growing effort to harness their superior performance for other applications as well. One such application is data compression, where frameworks like *cuSZp* take advantage of GPU acceleration to efficiently compress floating-point data arrays. Since compression and decompression involve performing numerous independent operations on large datasets, GPUs can process multiple data chunks simultaneously, leading to significant speedups compared to conventional CPU-based compression tools. By leveraging the massive parallelism and high memory bandwidth of GPUs, *cuSZp* provides a powerful and scalable solution for high-performance lossy data compression.

CuSZp supports both single- (float32) and double-precision (float64) data types, as well as two different encoding modes in plain and outlier mode. Using plain encoding mode, the entire dataset is directly compressed without handling extreme values differently, resulting in generally higher performance and is best suited for datasets with smooth data distributions or many zero values. If the data contains large variations, the outlier encoding can be used, which first separates extreme values from the dataset and then encodes them separately to preserve accuracy. The remaining data is then compressed more aggressively to limit the performance impact of special handling, with this mode being particularly useful in cases where localized precision is crucial while coming at the cost of slightly lower performance.

The compression algorithm implemented by *cuSZp* can be split into the following basic steps: the floating-point input data is first converted to integers (quantization), before applying a predictor function to estimate the value of each datapoint based on its neighbors. The resulting values are then further compressed using fixed-length encoding. Lastly, bit

manipulation operations are used to align the resulting data in a suitable format while including information such as the dimensions of certain distinct data blocks. Decompression is implemented by reversing the order of these steps, decoding the compressed bitstream into quantization residuals before converting them back to approximate floating-point values [40].

For the purpose of this evaluation, only the kernels related to double-precision data will be considered. As the kernels for both encodings are very similar in their structure, most bottlenecks identified in one of the encodings, as well as changes made, will translate to the other with exceptions being specifically mentioned. To be able to measure not only the changes in the analyses and metrics presented by GPUscout, but also the impact on actual performance, the `cuSZp_test_f64` binary, which is provided by the project, is evaluated after each change, with both the average throughput and standard deviation of 100 executions being compared. An overview of the performance before making any changes to the code can be seen in the following table:.

NVIDIA A100		
Encoding	Compression	Decompression
Plain	547.5 ($\pm$ 0.7) GB/s	422.9 ($\pm$ 1.8) GB/s
Outlier	447.0 ($\pm$ 4.1) GB/s	277.2 ($\pm$ 3.4) GB/s
NVIDIA RTX 4080		
Encoding	Compression	Decompression
Plain	219.8 ($\pm$ 4.8) GB/s	461.3 ($\pm$ 9.0) GB/s
Outlier	218.5 ($\pm$ 4.6) GB/s	452.6 ($\pm$ 9.2) GB/s

Table 5.2.: CuSZp_test_f64 throughput: regular implementation

In compressions using both encodings, the NVIDIA A100 GPU outperforms the NVIDIA RTX 4080 significantly, which can be explained by the superior FP64 performance shown earlier. While decompression throughput is substantially higher in contrast to compression for the NVIDIA RTX 4080 GPU, this is not the case for the NVIDIA A100, especially when using the outlier encoding, as the majority of compute-intensive FP64 operations, which this GPU is optimized for, are performed in the compression phase.

After analysis with GPUscout, several potential bottlenecks have been identified, which will now be discussed. In accordance with the expected workflow of GPUscout-GUI, rather than implementing changes analysis by analysis for all kernels, the results of the next kernel will only be considered after discussing the changes for each analysis of the current one. As previously mentioned, plain and outlier kernels for compression and decompression resulted in very similar results, so optimizations considered in the following sections apply to both encodings, unless otherwise stated. Even before comparing specific results of the legacy text output of GPUscout with the data presented in GPUscout-GUI, the necessity of displaying results in a different format becomes apparent, as the textual output generated spans more than 3200 lines, making it impossible to easily extract useful information out of. All changes described in the following sections are made based on the latest version 2.0.1 of the project[4].

[4]The source code can be found at `https://github.com/szcompressor/cuSZp/releases/tag/cuSZp-V2.0.1`

5.2.1. Compression Kernels

GPUscout identified relevant results in five different analyses, namely *Datatype Conversion*, *Register Spilling*, *Use Texture*, *Use Shared*, and *Warp Divergence*. Investigating analyses in the order they are presented in in GPUsout-GUI, *Datatype Conversion* is the first analysis to consider.

Datatype Conversion

The output of GPUscout is identical for both compression kernels and can be seen in listing 5.2, with slight differences only in the values of provided metrics. Users are informed of a number of float-to-integer conversions in line number 9 of the source file and given an overview of relevant kernel-wide stall metrics. In addition to the concrete values of those metrics, information about which metric is relevant depending on the type of datatype conversion is also provided.

```
1 INFO :: No F2F conversions found
2 INFO :: No I2F conversions found
3 WARNING :: There are 43 F2I conversions found at line numbers: 9, 9, 9, 9, 9,
    ↪ 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9,
    ↪ 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9,
4 For F2F (32 to 64 bit) conversions, check Tex throttle: 0 %
5 For I2F and F2F (32 bit only) conversions, check MIO throttle: 0 %
6 For I2F and F2F (32 bit only) conversions, check Short Scoreboard: 1 %
```

Listing 5.2: GPUscout output: cuSZp compression kernels (Datatype conversion analysis)

Based on the information shown, users might not see the need for any changes, as the values of all provided stall reasons are at or below one percent. While not much additional information in terms of metrics is provided in GPUscout-GUI apart from an overview of global loads and their cache hit rates, investigating the specific code lines in the GUI reveals information not provided in the textual output, which leads to a different interpretation of the results.

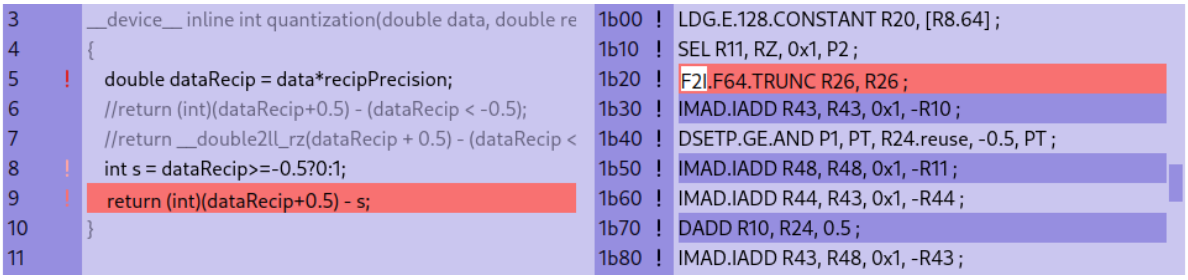


Figure 5.1.: Code view: Datatype conversion occurrence (cuSZp compression kernel)

All occurrences of datatype conversions map to the same source code line in both compression kernels, with the relevant code section being displayed in figure 5.1. As indicated by the

red exclamation marks next to the line numbers, a substantial amount of PC sampling stalls has been recorded inside the `quantization` function, which is not apparent in the textual output of GPUscout. After further investigation, about 29.3% of the stalls recorded in the plain compression kernel and 14.7% of the stalls recorded in the outlier compression kernel occur inside this single function, with the majority of stalls corresponding to short and long scoreboard stalls, indicating optimization potential.

Even though these conversions are not entirely avoidable due to their crucial role in the compression algorithm used by *cuSZp*, improvements can still be made. The main purpose of this function is to perform rounding to the nearest integer for the result of the multiplication result stored in `dataRecip`. Because the rounding mode round-towards-zero is used when casting non-integer values to integers by prefixing the value with `(int)`, a padding of *0.5* needs to be used to ensure rounding to the nearest integer instead. The subtraction of variable `s` is further necessary to correct for negative values that are rounded incorrectly due to truncation. This process can be simplified by using specialized functions that directly perform rounding to the nearest integer while handling negative values correctly, eliminating the need for manual adjustments while also offering better performance. In this case, either the `llrint` function or the CUDA-specific type-casting intrinsic `__double2int_rn` can be used, with `llrint` offering a better performance in the case of the NVIDIA A100 GPU and `__double2int_rn` being used for the NVIDIA RTX 4080, resulting in the following statement replacing the function body:

```
1 return __double2int_rn(data * recipPrecision);
```

After implementing this change and analyzing the resulting binary with GPUscout once more, the positive effect can be seen immediately when compared to the unmodified kernel version.

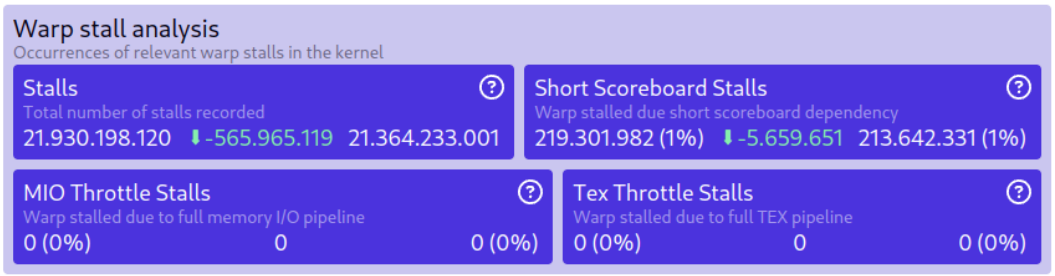


Figure 5.2.: Metrics component: Comparison after changes in cuSZp compression kernels (Datatype conversion analysis)

While the kernel-wide total and short scoreboard stalls have decreased slightly, the main impact of the change can be seen in the amount of PC sampling stalls recorded in the modified function. Both the total amount of stall samples, as well as the portion of samples occurring in the optimized `quantization` function decreased significantly, with samples in the `quantization` function decreasing from 29.3% to 9.8% and from 14.7% to 5.0% in the plain and outlier kernels, marking a significant reduction and validating the positive effect of the changes made.

	NVIDIA A100		NVIDIA RTX 4080	
Encoding	Compression	Change	Compression	Change
Plain	553.9 (± 0.7) GB/s	+1.3%	227.0 (± 5.2) GB/s	+ 3.2%
Outlier	460.2 (± 1.4) GB/s	+3.0%	229.1 (± 4.9) GB/s	+ 4.8%

Table 5.3.: CuSZp_test_f64 compression throughput: Datatype conversion changes

The actual performance impact can be seen in figure 5.3. In both encoding modes, a measurable impact of the changes can be seen, with compression throughput increasing between 1.3% and 4.8% between both encoding modes and GPUs, with greater improvements in the outlier encoding in both cases. These changes were based on information only available in GPUscout-GUI, emphasizing the effect the visualizations provided.

Register Spilling

Multiple instances of register spilling were identified in both compression kernels. The kernel-wide metrics, as well as information provided by the textual output of GPUscout for a single occurrence, can be seen in listing 5.3. For each occurrence, the relevant source line, as well as the name of the relevant register and operation of the spill, are given. Additionally, the previous compute instruction of this register and the number of currently live registers and PC stall samples recorded in the current SASS line are also displayed. Metrics provide users with an overview of the global memory data flow, as well as values of long scoreboard and LG throttle stalls, along with the percentage of L2 queries issued due to local memory accesses.

```
1 WARNING :: Spill detected in line number 157 of your code. Base register number
      ↪  R11 spilled in STORE operation
2 The previous compute instruction of register: R11 before spilling was IMAD at
      ↪ line number 130 of your code
3 INFO :: Total current registers for the SASS instruction: 13
4 Stalls are detected with % of occurence for the SASS instruction
5 stalled_wait (4.592825 %)
6 stalled_mio_throttle (4.670912 %)
7 stalled_selected (12.1201 %)
8 stalled_lg_throttle (78.616159 %)
9 INFO :: Data flow in memory for load operations
10 Kernel ---- request load data ----> Global Memory 1.10095e+07 instructions
11 Global memory ---- request load data ----> L1 cache 8.60418e+09 bytes
12 L1 Cache miss % (due to global memory load request) 54
13 L1 cache ---- request load data ----> L2 cache (due to global memory load
      ↪ request) 4.64626e+09 bytes
14 Local memory used in case of register spilling . . .
15 Local memory ---- request load data ----> L1 cache 9.84084e+09 bytes
16 L1 Cache miss % (due to local memory load request) 40
```

```
17 L1 cache ---- request load data ----> L2 cache (due to local memory load
       ↪ request) 3.93634e+09 bytes
18 L2 Cache miss % (due to L1 load data request) 68
19 L2 cache ---- request load data ----> DRAM 5.83616e+09 bytes
20 For register spilling, check Long Scoreboard stalls: 63 % per warp active
21 For register spilling, check LG Throttle stalls: 2 % per warp active
22 2.21042e+10 - 5.83891e+08
23 Percentage of total L2 queries due to LMEM: 37.8567 %
24 WARNING :: If the above percentage is high, it means the memory traffic between
       ↪  the SMs and L2 cache is mostly due to LMEM (need to contain register
       ↪ spills)
```

Listing 5.3: GPUscout output: cuSZp compression kernels (Register spilling analysis)

Although the textual output contains most of the information needed to determine that changes are necessary, such as high kernel-wide long scoreboard stalls, significant LG throttle stalls in the relevant SASS lines, and a high percentage of L2 queries due to local memory, this data is not presented in a clear and easily interpretable manner, one example being the poor formatting of large numerical values in data streams. After visualizing the results in GPUscout-GUI, it becomes apparent that the specific instances of spills are very similar in both compression kernels, as the source code lines these spills map to correspond to usages of three thread-local arrays in both cases. However, as the use of these arrays cannot be circumvented trivially, other techniques for reducing the maximum amount of simultaneously live registers need to be applied. Apart from smaller changes like reducing the scope of local variables, avoiding or reducing the use of loop unrolling resulted in small improvements. Loop unrolling refers to expanding the body of a loop multiple times, which can improve performance due to less loop control overhead but also impacts the number of live registers during execution. Loop unrolling is explicitly requested for three different loops in both kernels, including the loop performing quantization for individual blocks of data, in which GPUscout identified several occurrences of register spilling. After reducing the number of

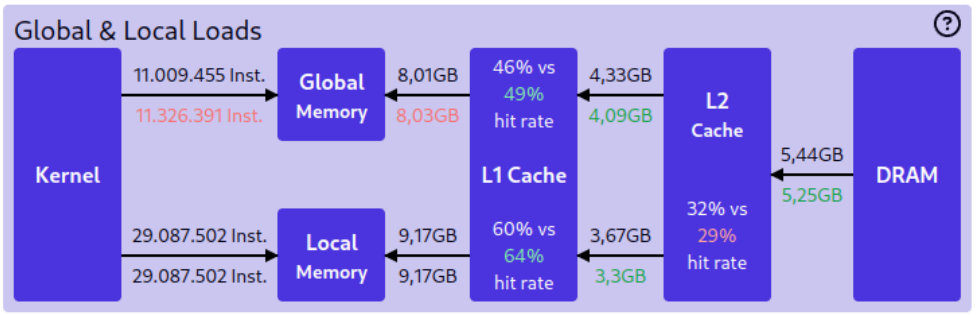


Figure 5.3.: Memory graph: Comparison after changes in cuSZp compression kernels (Register spilling analysis)

expansions that are unrolled at a time from 8 to 2, performance gains could be seen mainly

in the kernel implementing the outlier encoding, which will be focused on from now on. After comparing both kernel versions, although register spilling could not be completely eliminated, the number of individual occurrences of spilling reduced from 37 to only 10 in the relevant loop. Using the live register counts displayed in GPUscout-GUI, a reduction of the maximum number of live registers in the loop from 45 to 38 could also be identified, which can be be compared in a much more user-friendly manner than by using the textual outputs of both kernels. The impact of the changes on global and local data streams can be seen in the relevant memory graph, which is shown in figure 5.3. Apart from slightly higher global loads, the number of local stores decreased, with the most significant change being an increase in L1 cache hit rates and, thus, considerable reductions in the amount of data being requested from the L2 cache and DRAM. Small reductions in total and long scoreboard stalls can also be seen in the metrics presented in the user interface.

	NVIDIA A100		NVIDIA RTX 4080	
Mode	Compression	Change	Compression	Change
Plain	553.7 ($\pm$ 0.5) GB/s	< 1%	246.0 ($\pm$ 5.9) GB/s	+ 8.3%
Outlier	470.5 ($\pm$ 2.1) GB/s	+2.2%	242.2 ($\pm$ 5.6) GB/s	+ 5.7%

Table 5.4.: CuSZp_test_f64 compression performance: Register spilling changes

The impact these changes had on the performance of the binary can be seen in table 5.4. While no measurable change in throughput was visible for the plain compression kernel, an increase of about 2.2% could be seen in the outlier kernel for the NVIDIA A100, with improvements between 5.7% and 8.3% in both compression kernels in the case of the RTX 4080. This difference in impact the changes made between the GPUs is mostly due to the significantly lower FP64 performance of the RTX 4080 GPU and emphasizes the importance of optimizations especially in cases of hardware limitations.

Remaining Analyses

After implementing changes in both analyses, only occurrences in the *Use Shared*, *Use Texture* and *Warp Divergence* analysis are remaining. As individual elements in the pointers recommended to use texture or shared memory are only used once in all cases, with the metrics also not supporting any implementation changes, no modifications were made in these cases. While some occurrences of warp divergence were found in the associated analysis, which also showed a significant number of stall samples, due to the complexity of changes in this area and the low branch divergence percentage calculated by GPUscout, no changes were also made based on information in this analysis.

5.2.2. Decompression Kernels

Similarly to the compression kernels, GPUscout identified results in the *Datatype Conversion*, *Register Spilling*, and *Warp Divergence* analyses in both encoding modes for decompression, with the kernel implementing the outlier encoding also displaying results in the *Use Shared*

analysis. Similarly to the previous kernels, results are discussed in the order as displayed in GPUscout-GUI.

Datatype Conversion

In both decompression kernels, over 50 occurrences of integer-to-float datatype conversions have been identified by GPUscout, with the textual output providing their location in the source code, as well as values for the relevant kernel-wide metrics being displayed in figure 5.4 (duplicates of the relevant line numbers have been omitted for brevity).

```
1 INFO :: No F2F conversions found
2 WARNING :: There are 97 I2F conversions found at line numbers: 803, 834, 1197,
      ↪ 1203, 1209, 1222, ...
3 INFO :: No F2I conversions found
4 For F2F (32 to 64 bit) conversions, check Tex throttle: 0 %
5 For I2F and F2F (32 bit only) conversions, check MIO throttle: 7 %
6 For I2F and F2F (32 bit only) conversions, check Short Scoreboard: 5 %
```

Listing 5.4: GPUscout output: cuSZp decompression kernels (Datatype conversion analysis)

The main differences compared to the results provided for the compression kernels are the locations of the conversions, as well as slightly higher stall values. While these higher stall values might indicate a greater potential for performance improvements, after viewing the results in GPUscout-GUI, it becomes apparent that none of the relevant source code lines are associated with a significant number of PC sampling stalls, leading to the assumption that none of these datatype conversions are responsible for considerable performance bottlenecks. Each of the relevant source code lines is of the form

```
1 ... = currQuant * eb * 2;
```

with `currQuant` having the integer type and being converted to floating-point, as the pointer the resulting value is assigned to, as well as the variable `eb` are of this type. Similarly to the case of compression, these conversions cannot be avoided, as they are part of the algorithm used to implement core functionality. Using CUDA-native typecasting functions instead of relying on implicit conversion also did not increase performance in this case, further confirming the information displayed in GPUscout-GUI. Even though the results did not lead to performance improvements, this case emphasizes the effect of the additional information displayed in GPUscout-GUI, which leads to a considerably better and faster understanding of the expected effect of changes, for example, due to the immediate visibility of relevant source code lines without having to manually search for them.

Register Spilling

Register spilling occurs in the decompression using both encoding modes, with the output of a single occurrence, as well as the relevant metrics being provided in the following listing:

```
1 WARNING :: Spill detected in line number 672 of your code. Base register number
      ↪  R13 spilled in STORE operation
2 The previous compute instruction of register: R13 before spilling was IADD3 at
      ↪ line number 1188 of your code
3 INFO :: Total current registers for the SASS instruction: 47
4 Stalls are detected with % of occurence for the SASS instruction
5 stalled_wait (0.686213 %)
6 stalled_selected (0.155958 %)
7 stalled_not_selected (0.655022 %)
8 stalled_mio_throttle (38.6775 %)
9 stalled_lg_throttle (59.8253 %)
10 INFO :: Data flow in memory for load operations
11 Kernel ---- request load data ----> Global Memory 1.45836e+07 instructions
12 Global memory ---- request load data ----> L1 cache 2.35725e+09 bytes
13 L1 Cache miss % (due to global memory load request) 54
14 L1 cache ---- request load data ----> L2 cache (due to global memory load
      ↪ request) 1.27291e+09 bytes
15 Local memory used in case of register spilling . . .
16 Local memory ---- request load data ----> L1 cache 6.50117e+07 bytes
17 L1 Cache miss % (due to local memory load request) 94
18 L1 cache ---- request load data ----> L2 cache (due to local memory load
      ↪ request) 6.1111e+07 bytes
19 L2 Cache miss % (due to L1 load data request) 25
20 L2 cache ---- request load data ----> DRAM 3.33506e+08 bytes
21 For register spilling, check Long Scoreboard stalls: 54 % per warp active
22 For register spilling, check LG Throttle stalls: 9 % per warp active
23 Percentage of total L2 queries due to LMEM: 1.02943 %
24 WARNING :: If the above percentage is high, it means the memory traffic between
      ↪  the SMs and L2 cache is mostly due to LMEM (need to contain register
      ↪ spills)
```

Listing 5.5: GPUscout output: cuSZp decompression kernels (Register Spilling analysis)

Even though the information provided is sufficient for identifying the particular spill in the source code, the relevance of most metrics is not communicated clearly, leaving inexperienced users especially unaware of their role in determining the gravity of the spill. Information like the number of live registers is also only useful with the necessary context that is missing in this case, as it may not be apparent for all users if the provided number is high in the context of the kernel. After viewing the results in GPUscout-GUI, not only are metrics displayed in a clearer and more understandable manner, as already discussed in section 5.2.1, but seeing the occurrences directly in the code view instead of manually having to look for all relevant code lines significantly improves the workflow of locating and identifying a certain bottleneck, with a part of the code view of the outlier kernel shown in figure 5.4. Apart from the time savings mentioned, the ability to track the number of live registers across the entire kernel

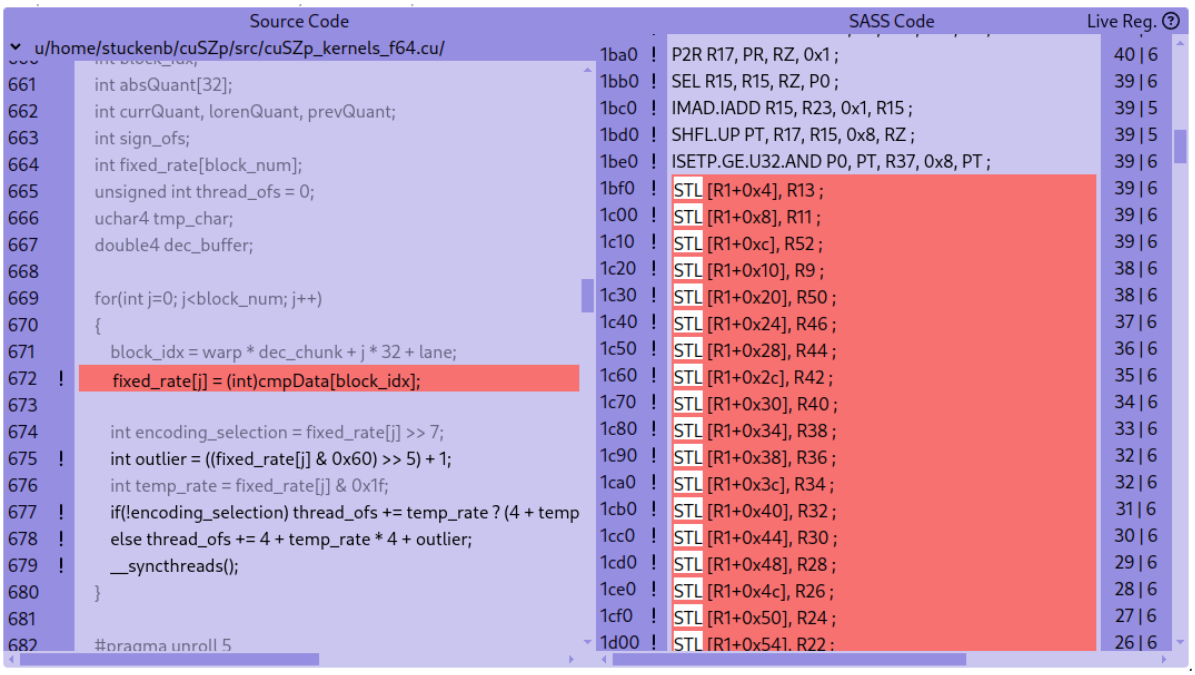


Figure 5.4.: Code View: Register spilling occurrence (cuSZp decompression kernel)

instead of only at the SASS lines of a particular spill significantly improves the ability of users to see the effect spill instructions have on the number of live registers, as well as allows to more easily determine the amount of registers available to the kernel. As each SASS line a spill occurs in displays high amounts of LG throttle stalls, with long scoreboard stalls being high kernel-wide, users are advised to try to contain the spill. After examining all occurrences of register spilling, it becomes apparent that the local `fixed_rate` array used in both decompression kernels is mostly responsible for the spill, the use of which can be avoided due to the way it is interacted with. As visible in the figure above, the values of each entry of `fixed_rate` are loaded from the pointer `cmpData` containing the compressed data in a for-loop at the start of the kernel and then only used to calculate another value, `outlier`. Later in the kernel, values of `fixed_rate` are then used in a second for-loop, whose header is identical to the first one, which allows avoiding the use of this array by initializing a local variable in each loop iteration instead, whose value is initialized from `cmpData` in both cases. This is possible, as the values inside `cmpData` are only read and not modified during the execution of the kernel, with the modifications made to the second loop identical to the initialization in the first one:

```
1 for(int j=0; j<block_num; j++)
2 {
3     block_idx = warp * dec_chunk + j * 32 + lane;
4     int fixed_rate = (int)cmpData[block_idx];
```

```
5       ...
```

Listing 5.6: cuSZp: Loop header after register spilling changes

After modifying the source code of both kernels in this manner and analyzing the resulting binary again, the result is compared to the original version of the kernels. Upon opening the comparison in GPUscout-GUI, it is immediately apparent that the change worked, as no entry for the *Register Spilling* analysis is present in the sidebar anymore, indicating that in the new version of the kernel, all occurrences of register spilling are eliminated.

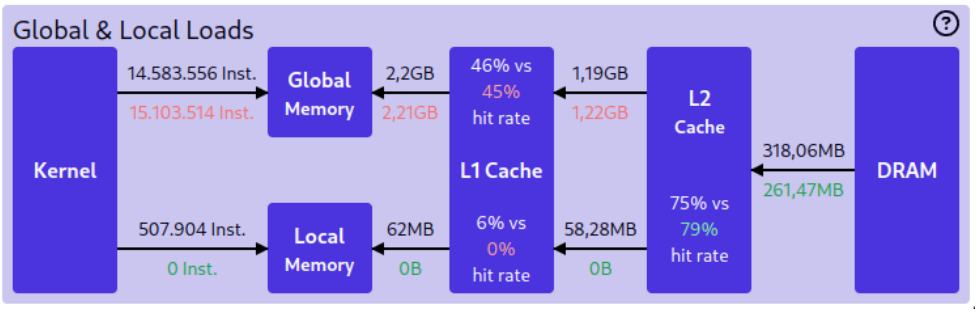


Figure 5.5.: Memory graph: Comparison after changes in cuSZp decompression kernels (Register spilling analysis)

This can be confirmed by inspecting the relevant memory graph in GPUscout, which can be seen in figure 5.5, displaying that no local loads occur in the kernel anymore, with global loads slightly increasing due to the additional load from the `cmpData` pointer. The positive effect can be seen not only in the reduced amount of data loaded from DRAM but also by other kernel-wide metrics like LG throttle stalls, which decreased by more than 10%. To verify this information, the throughput on both GPUs was measured once more with the results visible in table 5.5.

	NVIDIA A100		NVIDIA RTX 4080	
Mode	Decompression	Change	Decompression	Change
Plain	423.2 (± 1.6) GB/s	< 1%	475.9 GB/s (± 15.4)	+ 3.1%
Outlier	282.5 (± 0.7) GB/s	+2.0%	479.7 GB/s (± 17.5)	+ 5.9%

Table 5.5.: CuSZp_test_f64 decompression throughput: Register spilling changes

In the case of the NVIDIA A100 GPU, no measurable effect can be seen in the plain encoding mode, with an improvement of 2% in the outlier encoding. Improvements in both encodings can be seen for the NVIDIA RTX 4080, with the larger impact on the outlier mode in both GPUs being consistent with expectations, as the relevant local array was used much more heavily in this kernel than in the kernel implementing the plain encoding.

Remaining Analyses

After implementing changes in both analyses, only occurrences in the *Warp Divergence* analysis remain. While some occurrences are associated with a significant number of stall samples, due to the complexity of changes in this area and the low branch divergence percentage calculated by GPUscout, no changes were made here.

5.2.3. Performance Evaluation

A summary of the impact of all implemented changes on the compression and decompression throughput using both encodings on the used GPUs can be seen in the following table, which shows the relative change in the average throughput in addition to the absolute change in standard deviation of individual results:

NVIDIA A100				
Encoding	Compression	Change	Decompression	Change
Plain	553.7 (± 0.5) GB/s	+ 1.1% (- 0.2)	423.2 (± 1.6) GB/s	< 1% (- 0.2)
Outlier	470.5 (± 2.1) GB/s	+ 5.3% (- 2.0)	282.5 (± 0.7) GB/s	+ 2.0% (- 2.7)
NVIDIA RTX 4080				
Encoding	Compression	Change	Decompression	Change
Plain	246.0 (± 5.9) GB/s	+ 11.9% (+ 1.1)	475.9 (± 15.4) GB/s	+ 3.1% (+ 6.4)
Outlier	242.2 (± 5.6) GB/s	+ 10.8% (+ 1.0)	479.7 (± 17.5) GB/s	+ 5.9% (+ 8.3)

Table 5.6.: CuSZp_test_f64 throughput: summary

In the case of the NVIDIA A100 GPU, improvements can be seen mainly in kernels using the outlier encoding, with improvements in throughput of up to 5.3%. This is expected, as the plain encoding is significantly less compute-intensive and thus does not benefit as much from the implemented changes, although a slight increase in throughput can be measured in both associated kernels. In all cases, the standard deviation of individual results aggregated to form the presented metrics declined, further confirming the positive nature of the changes.

The NVIDIA RTX 4080 GPU shows improvements primarily in both compression kernels, achieving up to 11.9% more throughput compared to the unmodified kernel versions. These results were anticipated due to the heavy reliance on FP64 computations of the compression algorithm and the considerably lower FP64 performance of this GPU compared to the NVIDIA A100. Additionally, the decompression kernels also exhibited strong throughput improvements between 3.1% and 5.9%. These larger performance gains come at a cost, however, as the standard deviation across individual results increased in all tested kernels.

5.3. Deal.II / Step-64

Deal.II[5] (Differential Equations Analysis Library) is an open-source C++ library initially developed at the University of Heidelberg in Germany and is widely used in the computational

[5] `https://www.dealii.org/`

solution of partial differential equations using finite element methods. Being capable of handling a variety of finite elements in up to three dimensions, as well as offering a visual representation of computation results, its use enables a significant reduction of both the workload and complexity of programs depending on these computations. While *deal.II* provides MPI-based parallelism for distributed-memory systems, it lacks built-in abstractions for shared-memory parallelism across heterogeneous architectures. This gap is filled by a second library, *kokkos*[6], which is bundled with *deal.II* and provides a portable parallel programming abstraction for HPC applications. In the context of *deal.II*, this enables offloading the handling of memory and computation of defined data structures to *kokkos* kernels, removing the need to write hardware-specific code while avoiding performance loss due to missing optimizations. By analyzing one of the provided *deal.II* example projects, valuable insights into how both GPUscout-GUI, as well as GPUscout handle larger projects using multiple complex dependencies and libraries can be gained.

Deal.II comes with several prepared example kernels, which can be built independently and are designed to gradually introduce users to the different mathematical concepts covered by the library. For this evaluation, the *step-64* kernel[7] has been selected, which implements a matrix-free method for solving the Helmholtz equation with variable coefficients on a hypercube. The Helmholtz equation is a fundamental partial differential equation that arises in various fields, such as acoustics and fluid dynamics and can be used to describe wave propagation and steady-state oscillatory behavior. The implemented form of the equation is:

$$\begin{aligned} -\nabla \cdot \nabla + a(x)u &= 1 \\ u &= 0 \text{ on } \partial\Omega \end{aligned}$$

$$\text{where } \Omega = [0,1]^3 \text{ and the variable coefficient } a(x) = \frac{10}{0.05+2||x||^2}$$

Instead of performing traditional matrix multiplications, the equation is solved by applying the action of a matrix on a vector directly, improving memory efficiency and allowing for more efficient parallel computation. The resulting linear system is then solved iteratively using the conjugate gradient method. Upon analyzing the resulting binary with GPUscout and opening the generated JSON file in GPUscout-GUI, users are presented with results in 22 kernels, with GPUscout identifying 260 occurrences of potential performance bottlenecks across a total of 75 analyses, indicating significant optimization potential in the project.

Differences in Kernel Names

The legacy textual output generated by GPUscout spans more than 1000 lines, which makes extracting useful information difficult. After visualizing the results in GPUscout-GUI, some differences arise between the expected and actual data, starting with the analyzed kernels. The

[6] `https://kokkos.org/`

[7] `https://dealii.org/current/doxygen/deal.II/step_64.html`

source code of the step-64 *deal.II* example contains three kernels, implementing the `operator` function for each of the classes `VaryingCoefficientFunctor`, `HelmholtzOperatorQuad` and `LocalHelmholtzOperator`, however, a total of 22 kernels are available for analysis in GPUscout-GUI, which may lead to confusion, especially as their names do not resemble any of the previously mentioned kernel or class names. An example of this is provided in listing 5.7, which corresponds to the name of a kernel as displayed in the UI.

The reason for these diverging kernel names is the use of the *kokkos* library and its abstraction model. Kernel functions in the step-64 source file are not directly marked as CUDA kernels with the `__global__` keyword but use a macro defined in the `deal.II` library, which translates to the `KOKKOS_FUNCTION` macro of the *kokkos* library. The use of this macro corresponds to the `__host__` or `__device__` markup in CUDA and allows a function to be executed in a parallelized manner in *kokkos*[8].

```
1 void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::ParallelFor<
    ↪ void dealii::Portable::MatrixFree<(int)3, double>::evaluate_coefficients
    ↪ <VaryingCoefficientFunctor<(int)3, (int)4>>(T1) const::[lambda(int, int)
    ↪  (instance 1)], Kokkos::MDRangePolicy<Kokkos::Cuda, Kokkos::Rank<(
    ↪ unsigned int)2, (Kokkos::Iterate)0, (Kokkos::Iterate)0>>, Kokkos::Cuda
    ↪ >>(T1)
```

Listing 5.7: Step-64: Demangled kernel name

Even though this has a significant impact on the usability of the results, users with knowledge of the *kokkos* library are still able to identify which code section a particular kernel is associated with, mainly by analyzing its parameters. The use of demangled kernel names in the GUI compared to the textual output of GPUscout eases this process significantly and highlights one of the advantages of GPUscout-GUI. Additionally, the code line mappings between intermediary and source code can be used to more easily locate relevant source code lines for a specific kernel. Users are also able to quickly identify which kernels are relevant for analysis due to the fact that no metrics have been collected for the majority of kernels, which is clearly communicated in the UI, but missing from the textual output.

Analysis Results

While intuitive results are provided in some cases, the intermediary code lines of most highlighted occurrences map to source files other than the one the actual kernels are defined in. The only exception of this can be seen when analyzing the kernel mentioned in figure 5.7, in which GPUscout identified potential performance bottlenecks in three analyses. In the *Use Texture* analysis, a single instance of optimization potential has been found, the relevant code section of which can be seen in figure 5.6. Using the mapping to a concrete source lines in one of the defined kernels, users are able to implement changes based on this recommendation. An overview of the kernel-wide stalls, as well as usage of texture and global memory, also provides the context and guidance necessary to determine the expected impact of these

[8] `https://kokkos.org/kokkos-core-wiki/index.html`

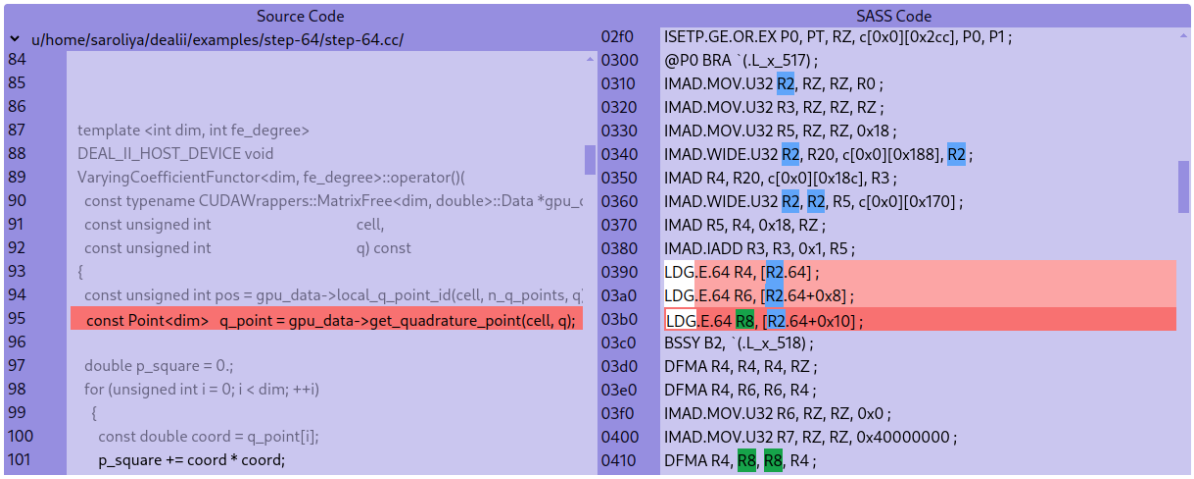


Figure 5.6.: Code view: Use Texture occurrence (Step-64 kernel)

changes. Due to the complex nature of the kernel, as well as the significant changes necessary, no modifications were made at this point.

In the *Datatype Conversion* analysis, however, the primary SASS lines of all occurrences map to source code in files other than the one all step-64 kernels are defined in, as is the case in most analyses and kernels. As can be seen in figure 5.7, seven different source files are involved in the line mappings of the current kernel, corresponding to internals of either the *kokkos* or *deal.II* library, which does not lead to any optimizations relevant to the actual analyzed kernel.

As described in section 4.2.2, this is not unexpected as the intermediary code of kernels can not only map to the actual source file but also include libraries or other linked files. However, only a single of the 260 occurrences found by GPUscout across all kernels maps to a line in the source file of the step-64 kernels, significantly reducing the usability of the results. This is emphasized by the fact that mappings to an intermediary code lines exist for only four relevant source lines, as indicated by the color of the source code lines in the code view of GPUscout-GUI. In total, 20 distinct source files are involved in code line mappings, six of which are related to the *deal.II* library, with 13 related to *kokkos*. In summary, the use of wrappers for kernel functions in the *kokkos* library leads to the NVIDIA tools *nvdisasm* and *cuobjdump* not being able to extract as fine-grained code-line mappings as if these would not be used.

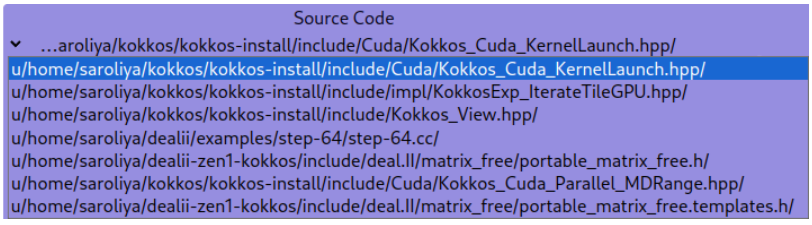


Figure 5.7.: Source file selector (Step-64 kernel)

Usability of Results

While most bottlenecks found by GPUscout cannot easily be translated to actual source code lines in the defined kernels, helpful information is still provided by GPUscout-GUI in the case of some analyses and kernels. For example, in the case of the *Register Spilling* analysis, in which multiple occurrences have been identified, conveying that a spill took place, along with providing the live register counts in the SASS code, can still be valuable when assessing the overall performance of the binary or when comparing different kernel versions after a certain change has been implemented. Another example is the *Use Shared* analysis, where the indication of bank conflicts alone can lead to users locating bottlenecks. Kernel-wide metrics are also still useful tools in evaluating the general performance of kernels, with especially stall values and the complete memory graph providing additional value. In case of the kernel mentioned in listing 5.7, for example, a high percentage of long scoreboard stalls is displayed in GPUscout-GUI, indicating optimization potential. Lastly, the presence of stall samples in certain parts of the intermediary code can also be used to identify certain parts of the computations performed, which could be optimized.

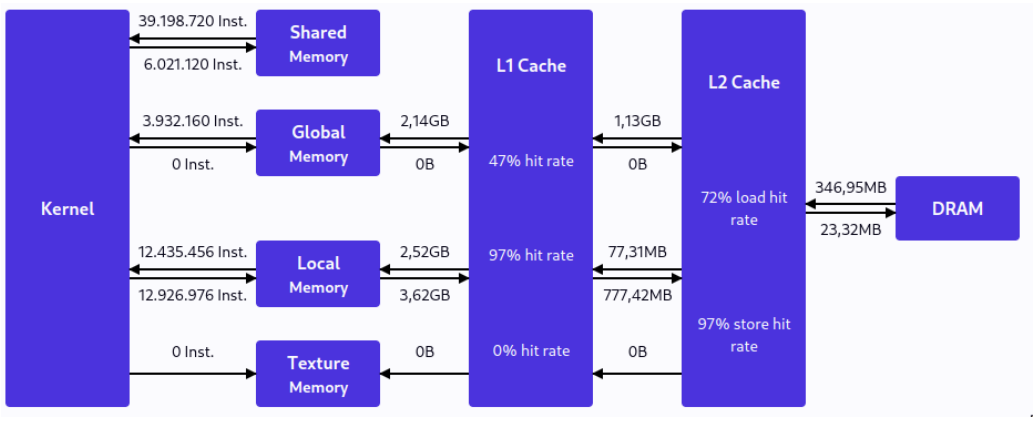


Figure 5.8.: Memory graph: Step-64 kernel

Figure 5.8 displays the complete memory graph for one of the analyzed and executed kernels, which provides a strong overview of the data streams in the kernel. While several instances of register spilling occurred, the high L1 cache hit rate of 97% indicates most spills are contained by the L1 cache, with improvement potential mostly in the significantly lower cache hit rate of global memory. While most of the information presented is also part of the textual representation of results, presenting the data visually significantly improves user experience as users may not be able to easily locate relevant information in the large output generated by GPUscout.

Advantage of GPUscout-GUI regarding Code Line Mappings

Even though analyzing this project with GPUscout and GPUscout-GUI did not lead to the implementation of changes, which in turn lead to performance improvements, it can be used

to illustrate one of the main advantages the graphical user interface provides to users, which otherwise may receive misleading information from GPUscout. Listing 5.8 shows a small portion of the textual output presented to the user, informing of an optimization found in line number 1281, which should use the `__restrict__` keyword. As the name of the file in which the change should be made is not mentioned, users are led to assume that this refers to the actual source file the analyzed kernels are defined in. When looking at the occurrence in GPUscout-GUI, however, it becomes clear that the relevant SASS line maps to a different file.

```
1 ---------------------- Use of __restrict__ analysis for kernel:
    ↪ _ZN6Kokkos4Impl33cuda_parallel_launch_local_memoryINS0_11ParallelForIN6deal
    ↪  ii8Portable8internal11ApplyKernelILi3Ed22LocalHelmholtzOperatorILi3ELi4
    ↪  EEEENS_10TeamPolicyIJNS_4CudaEEEESB_EEEEvT_ ----------------------
2 INFO :: Register R234, in line number 1281 of your code, is not aliased
    ↪ anywhere in the kernel
3 WARNING :: You can benifit from using __restrict__ for register R234 at line
    ↪ number 1281 of your code
```

Listing 5.8: GPUscout output: Step-64 (Use restrict analysis)

This phenomenon is not unique to this project but can occur in each analyzed binary compiled using multiple source files. Clearly displaying the full path of the source file a SASS or PTX line maps to ensures users are able to easily locate the relevant parts of the source code implementation changes are recommended in, without having to manually search for the source file containing the definition of a particular kernel.

5.4. Jacobi Iteration

One of the biggest advantages of performing computations on the GPU instead of the CPU is the ability to parallelize large amounts of frequently used operations. The Jacobi method is an algorithm for approximating the solution of a system of linear equations and, due to its iterative nature, can benefit greatly from the parallelization potential offered by modern GPUs. Jacobi Iteration solves linear equations of the form $A \cdot x = b$, where the coefficient matrix A and the result vector b are already known, by iteratively updating the value of the vector x using the result from the previous iteration. After an initial guess for the solution of x, it is updated iteratively using the following formula:

$$x_i^{(k+1)} = \frac{1}{A_{ii}}(b_i - \sum_{j \neq i} A_{ij} x_j^{(k)})$$

This computation can be effectively parallelized by assigning each thread a row in the coefficient matrix A and thus computing a single element of the result of the current iteration.

```
1 __global__ void jacobi(const float *A, const float *b, float *x, float *xNew,
    ↪ int N) {
2     int row = blockIdx.x * blockDim.x + threadIdx.x;
```

```
3
4     float sum = 0.0f;
5     for (int col = 0; col < N; ++col) {
6         if (col != row) {
7             sum += A[row * N + col] * x[col];
8         }
9     }
10    xNew[row] = (b[row] - sum) / A[row * N + row];
11 }
```

Listing 5.9: Jacobi Iteration: Naive implementation

The kernel shown in listing 5.9 shows the naive CUDA implementation of the algorithm described above. Each thread is responsible for computing a single value of the new approximation for x, the index corresponding to the thread's index. The kernel first iterates over all elements of a row of the matrix A to compute the sum of $A[row, col] \cdot [col]$ for all columns except the diagonal element, before updating relevant element of `xNew` using the Jacobi iteration formula. The kernel assumes `N` threads to be available, with each thread handling one row of the matrix in parallel. `N` was defined as 128 for all following kernels, with the iteration concluding after either a maximum of 10.000 iterations or an error of less than 1e-5 between two elements of consecutive result vectors. To ensure convergence, the input matrix `A` has been generated to be diagonally dominant, which corresponds to the following condition:

$$|A_{ii}| > \sum_{j \neq i} |A_{ij}|$$

The CUDA code is then compiled and analyzed with GPUscout before visualizing the results using GPUscout-GUI. After opening the relevant result file in GPUscout-GUI, users are presented with findings in five different analyses: *Use Restrict*, *Use Shared*, *Use Texture*, *Vectorization*, and *Warp Divergence*, which will now be examined in more detail. To be able to measure the impact of implemented changes on the runtime of the kernel, each kernel was executed 100 times to evaluate the average runtime and standard deviation after each modification. As the size of the input data is a considerable factor to the runtime, matrices and vectors of three different sizes have been evaluated, with results from GPUscout and GPUscout-GUI presented in the following sections assuming a matrix of size 128x128. The runtimes of the unmodified kernel shown above can be seen in the following table:

	NVIDIA A100	NVIDIA RTX 4080
N	Runtime	Runtime
128	18700ms (± 1010ms)	7840ms (± 873ms)
256	65140ms (± 1709ms)	37902ms (± 526ms)
512	384983ms (± 1194ms)	202312ms (± 1592ms)

Table 5.7.: Jacobi iteration runtime: Naive implementation

As expected, the runtime scales with increases in the size of the input data, with the NVIDIA RTX 4080 outperforming the NVIDIA A100 significantly due to the use of 32-bit floating-points numbers in the kernel and its superior FP32 performance illustrated in section 5.1.

5.4.1. Restrict Analysis

Two occurrences of pointers, where the usage of the `__restrict__` keyword is recommended, have been identified by GPUscout, corresponding to the matrix `A` and vector `x`. As can be seen in listing 5.10, only the name of the relevant register in the SASS code, as well as the corresponding source line number and eventual to PC stall samples are presented to users in the textual output of GPUscout, in addition to kernel-wide IMC miss stalls.

```
1 INFO :: Register R31, in line number 12 of your code, is not aliased anywhere
    ↪ in the kernel
2 WARNING :: You can benifit from using __restrict__ for register R31 at line
    ↪ number 12 of your code
3 If using __restrict__ (read-only cache), check IMC miss: 2 % per warp active
```

Listing 5.10: GPUscout output: Jacobi iteration (Use restrict analysis)

In GPUscout-GUI, this information is extended by several additional metrics like the achieved occupancy, long scoreboard stalls and the number of instructions related to global memory, which might be relevant in the context of this analysis. Live register counts for individual SASS lines are also provided, as they are crucial for determining the expected performance impact of this analysis, as using the `__restrict__` keyword can increase register usage. In contrast to the textual output, an explanation of this is given in the code info upon selecting one of the occurrences.

After implementing these changes for both mentioned pointers, the new kernel is compared to the unmodified version. Although a decrease of long scoreboard stalls of 7%, as well as an increased L2 cache hit rate can be seen in the UI, which is not included in the relevant textual output, no measurable effect on the runtime of the kernel can is observed.

5.4.2. Use Shared Analysis

Shared memory can significantly improve performance when used for frequently accessed data. In the case of the Jacobi kernel, GPUscout identified four instances where using shared instead of global memory could be beneficial.

GPUscout Output

The output produced by GPUscout can be seen in listing 5.11 and has been truncated to display only the result relevant to a single occurrence. For this analysis, users are provided with several pieces of information: In addition to the name of the relevant register in the SASS code and the corresponding source code line, the textual output includes the number of global loads and computation instructions involving this register, and whether the register is

utilized within a for-loop. Based on this data, the use of shared memory is recommended for the given register. The outputs for the remaining occurrences are very similar, all of which refer to the same source line performing the multiplication of a single matrix and vector element.

```
1 Register number R2 at line number 14 of your code has 1 total global load
    ↪ counts and 4 computation instruction counts
2 This register seems to be in a for loop and hence will perform multiple load
    ↪ operations
3 WARNING :: Since the data at register number R2 is accessed multiple times, you
    ↪  can benifit from using shared memory instead of global memory.
4 INFO :: Check data flow in shared memory, if you modify your code to use shared
    ↪  memory
5 Kernel ---- request load data ----> Shared Memory 0 instructions
6 If using shared memory, check Long Scoreboard: 86.04 %
7 If using shared memory, check MIO throttle: 0 %
8 INFO :: Check bank conflict in shared memory, if you modify your code to use
    ↪ shared memory.
9 Shared memory efficiency for load operations: 0 %
10 No shared memory data request made
```

Listing 5.11: GPUscout output: Jacobi iteration (Use shared analysis)

Following occurrence-specific details, users are also provided with a set of relevant metrics, mainly made up of kernel-wide values for long scoreboard and MIO throttle stalls, as well as insights into the current usage of shared memory. While the relevance of some metrics is explained in the output, such as the need to check for bank conflicts after implementing shared memory, especially in the case of long scoreboard and MIO stalls, it is not clearly indicated whether these metrics should decrease after the optimization or if they should already be low beforehand, as they might increase instead. This ambiguity can make it challenging for users to determine the expected effects of their changes and accurately assess performance improvements.

GPUscout-GUI Interface

Displaying the results of GPUscout in GPUscout-GUI not only enables the presentation of more information but also influences how users interpret those results. One of the main challenges when relying solely on the textual output of GPUscout is a user's ability to accurately map the relevant register in the SASS code of a given occurrence to the corresponding pointer in the source code, with it not being immediately clear, whether a specific occurrence pertains to the pointer `A` or `x` in the relevant source code in the current case. Users can more easily establish these connections by utilizing the GUI, improving clarity and efficiency in the optimization process. This can be illustrated by figure 5.9, which displays a part of the code view in GPUscout-GUI after selecting the occurrence corresponding to the output shown above. In addition to the SASS line containing the initial global load at address `0a68`, all lines

corresponding to further global loads and computation instructions the register is used in are highlighted. Focusing on the line with address `0ad0`, the `FFMA` instruction, which corresponds to a fused multiply-add, can be seen, which, after further inspection of the involved registers, can be identified as one of the operations performed in the relevant source code line. Even though it is not clear if the highlighted register `R2` relates to pointer `A` or `x`, the fact that another occurrence visible in the figure shown (at address `0a78`) corresponds to the same line but with register `R5` can lead to the conclusion that both pointers in the source code line are found to be relevant, with a more detailed analysis of the SASS code being necessary to identify which is which. Although tedious, especially in more complex kernels, this shows one of the main advantages of GPUscout-GUI compared to GPUscout, where making this distinction is not easily possible.

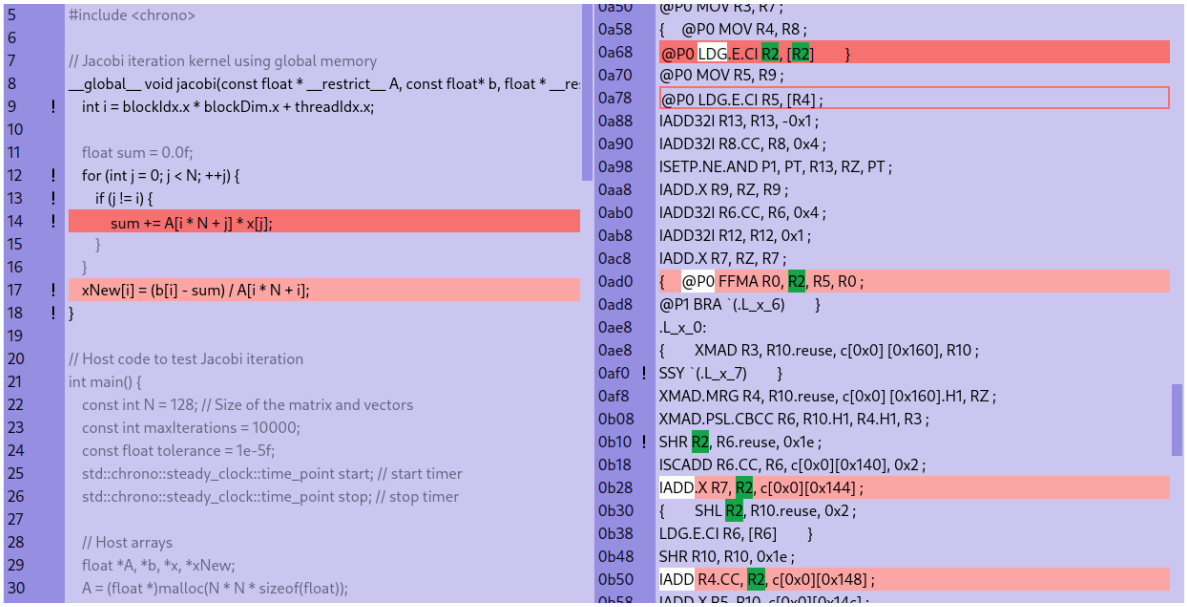


Figure 5.9.: Code View: Use shared occurrence (Jacobi iteration kernel)

Examining the source code of the Jacobi kernel reveals that implementing shared memory is beneficial for the pointer corresponding to the vector `x` but not the matrix `A`. While all but one element of `x` is read per row iteration of the matrix, each element of `A` is only accessed once, making it unsuitable for shared memory optimization. Although experienced users are able to easily make this distinction, the current textual output of GPUscout does not clearly convey this fact, potentially leading inexperienced users to implement changes degrading instead of improving performance. To address this issue, GPUscout-GUI includes detailed recommendations in the code info section, outlining when shared memory should and should not be used. This additional guidance, alongside a more detailed explanation of the expected effects on metrics like stalls, aims to prevent this kind of misinterpretation.

Comparison

After modifying the code of the Jacobi kernel to use shared memory for the vector `x`, the resulting binary is analyzed with GPUscout once more, the result of which is then compared to the original code in GPUscout-GUI. After selecting the *Use Shared* analysis, the number of occurrences indicating potential bottlenecks reduced to one, which corresponds to the pointer `A`. The difference in metrics between the two kernel versions, as it is presented in the application, can be seen in figure 5.10, which displays a significant reduction in total and long scoreboard stalls, aligning with the desired effect mentioned in the recommendations. The use of shared memory instead of only relying on global memory can also be seen in the increased number of shared loads and shared load efficiency. Instead of having to compare the textual output of the two analyses run in GPUscout, users can now easily and quickly see that the implemented change worked.

Figure 5.10.: Metrics list: Comparison after changes in Jacobi iteration kernel (Use shared analysis)

After evaluating the runtime of the kernel with varying sizes of the matrix `A`, improvements can be seen in all cases on both GPUs, with improvements generally scaling with increasing input sizes. Matrices of size 512x512 on the NVIDIA A100 GPU mark an exception to this pattern, whose performance improved by a smaller margin than other kernels.

	NVIDIA A100		NVIDIA RTX 4080	
N	Runtime	Change	Runtime	Change
128	18308ms (± 1215ms)	- 2.1%	7670ms (± 409ms)	- 2.2%
256	62854ms (± 1107ms)	- 3.6%	37269ms (± 1258ms)	- 3.0%
512	380371ms (± 2625ms)	- 1.2%	192477ms (± 3450ms)	- 5.1%

Table 5.8.: Jacobi iteration runtime: Shared memory implementation

All occurrences related to the *Use Texture* analysis also disappeared after implementing these changes, as they referred to the same registers in the SASS and pointers in the source code.

5.4.3. Vectorization Analysis

After addressing all bottlenecks identified in the *Use Shared* analysis, the next analysis GPUscout found occurrences in is *Vectorization*. This optimization is particularly important when load operations are performed inside a loop. By replacing these individual load operations with vectorized loads, the number of load operations can be reduced, which in turn can lead to significant performance improvements.

GPUscout Output

The relevant output of GPUscout is shown in listing 5.12, which includes the name of the relevant register and the corresponding source code line, along with the number of adjacent memory accesses the register is involved in. Additionally, the output provides insight into the number of live registers at this point in execution, as well as eventual PC sampling stalls. Kernel-wide values for long scoreboard stalls and the achieved occupancy are also provided.

```
1 WARNING :: Use vectorized load for register R4, in line number 18 of your code
2 Register R4 in line number 18 of your code has 19 adjacent memory accesses
3 INFO :: Total current registers for the SASS instruction: 28
4 Stalls detected with % of occurence for the SASS instruction
5 stalled_wait (61.5385 %)
6 stalled_selected (38.4615 %)
7 If you are using non-vectorized load/store, check Long Scoreboard: 85.72 % per
    ↪ warp active
8 INFO :: Using vectorized load increases the register pressure and hence might
    ↪ affect occupancy. Occupancy achieved: 8.23 %
```

Listing 5.12: GPUscout output: Jacobi iteration (Vectorization analysis)

In comparison to the output of the previously mentioned *Use Shared* analysis, the textual output already informs users about the semantics of all relevant metrics. Using this information, as well as the occurrence-specific data, users are able to perform the desired optimization easily.

GPUscout-GUI Interface

Although the textual output of GPUscout already provides all necessary information, inspecting the results in GPUscout-GUI offers some additional insights that enable inexperienced users to more easily grasp the expected performance impact changes in these analysis promise. As can be seen in figure 5.11, after selecting one of the relevant occurrences in the source code, all following lines corresponding to load instructions using the same base register are also highlighted, visualizing the amount of load instructions relevant to the current optimization. All data included in the textual output is additionally provided in a more verbose manner, as already explained in previous analyses, with additional metrics like an overview of the number of live registers per SASS line adding additional insights.

Comparison

After implementing the change to use vectorized loads to access elements of both the matrix `A` and vector `x` in the for-loop of the kernel, it is compared to the previous version implementing shared memory in GPUscout-GUI. Upon opening the application, users can immediately recognize that the change was implemented correctly, as the entry of the vectorization analysis in the sidebar is greyed out, indicating no occurrences regarding optimizations have been

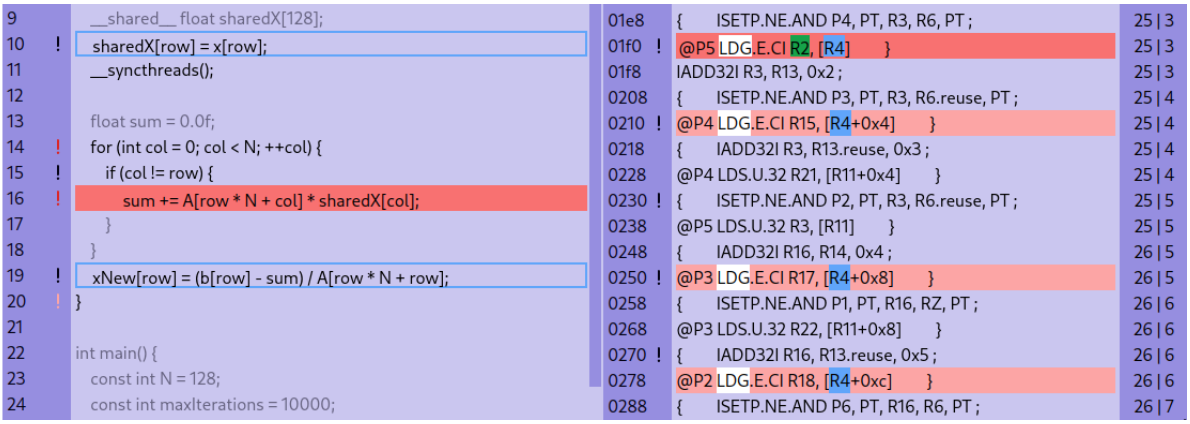


Figure 5.11.: Code View: Vectorization occurrence (Jacobi iteration kernel)

found. This is emphasized by the change of relevant metrics, with the total amount of stalls having decreased by more than 50%, mostly due to lower long scoreboard stalls. The total number of global loads also decreased from 540 to 140, indicating a significant performance improvement. The change can also clearly be seen in the memory graph, with the number of

Figure 5.12.: Metrics list: Comparison after changes in Jacobi iteration kernel (Vectorization analysis)

load instructions and data requested from shared and global memory significantly reduced. While the graph allows users to easily reconstruct the change in data flow, two values stand out upon closer inspection: The L2 cache hit rate, which increased to more than 100%, and the negative amount of data stored to DRAM. The L2 cache hit rate is taken directly from the output generated by NVIDIA's NSight Compute CLI, with the semantic meaning of values higher than 100% not known at this time. Due to the way in which this value is handled in computations in GPUscout, this results in a negative value for the stores in DRAM and will have to be adapted in the future.

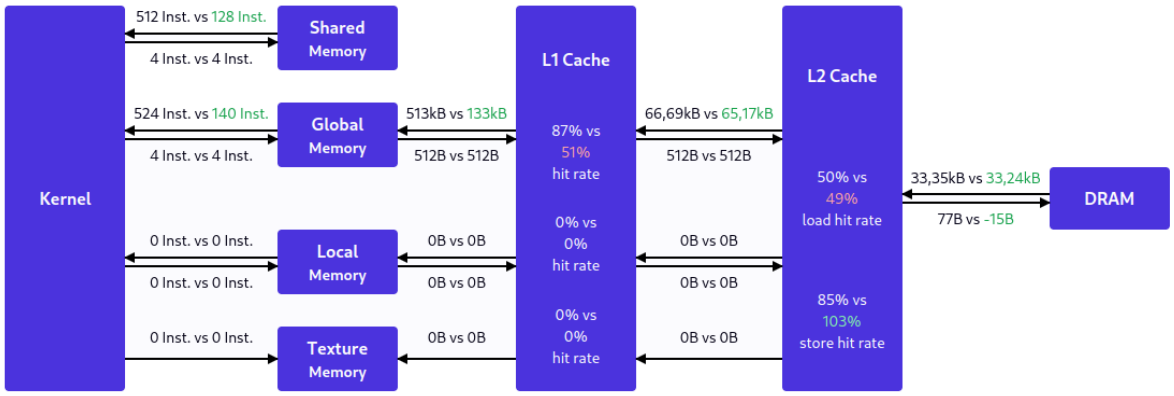


Figure 5.13.: Memory graph: Comparison after changes in Jacobi iteration kernel (Vectorization analysis)

The effect of these changes on the runtime of the kernel can be seen in the following table, which show the most significant performance improvements among all implemented changes. Expectedly, the benefit of using vectorized loads increases with the amount of data used in the kernels, with the NVIDIA A100 GPU showing the largest reduction in runtime of more than 160%. Due to the high FP32 performance of the NVIDIA 4080 GPU, no measurable improvement could be seen in the case of 128x128 matrices, with gains only seen in larger input sizes.

	NVIDIA A100		NVIDIA RTX 4080	
N	Runtime	Change	Runtime	Change
128	15792ms (± 752ms)	- 15.9%	7615ms (± 421ms)	< 1%
256	38520ms (± 2721ms)	- 63.2%	16255ms (± 1396ms)	- 129.3%
512	145868ms (± 3893ms)	- 160.7%	76381ms (± 2427ms)	- 151.9%

Table 5.9.: Jacobi iteration runtime: Vectorization implementation

After these changes, only occurrences in the *Warp Divergence* analysis remain, with a warp divergence percentage near 0% indicating no optimization potential, with no implementation changes performed.

6. Future Work

The graphical interface created in this thesis significantly enhances the functionality of GPUscout by introducing new features, visualizations, and supplementary information. Although this greatly improves usability, particularly for inexperienced users, there are still areas where further enhancements could be made. Apart from smaller improvements like more detailed help texts and better integration of memory topology metrics, the following areas offer the greatest potential for additional functionality:

Opinionated Results

Even though both GPUscout and GPUscout-GUI provide extensive descriptions of results, as well as recommendations for the next steps, most of the recommendations are defined in a very broad manner. An example of this is the *Use Texture* analysis, optimizations in which should only be performed in case of already low TEX and long scoreboard stalls. While respecting this information is crucial in ensuring users only perform changes that are expected to improve performance, assessing if a certain percentage of stalls is considered low can be difficult especially for inexperienced users. One approach to improve the user experience for this user group would be to define concrete thresholds that can be used as an orientation instead. In the example of texture memory, this could, for example, correspond to a mentioning a rough maximum percentage for TEX and long scoreboard stalls instead.

Global Statistics

When analyzing an individual kernel, GPUscout-GUI provides a strong overview of metrics and relevant information, depending on the analysis. However, especially in larger projects with tens or even hundreds of kernels, the initial step of reviewing the results of each individual kernel and assessing their optimization potential is very tedious. While the kernel selector provides a basic overview of the number of relevant analyses and occurrences for each kernel, additional value could be found in a separate page displayed upon opening a result file in the GUI, which provides a more detailed overview of the analyses and kernels optimizations were found in. Aggregated metrics from all kernels could also benefit users by being able to quickly grasp the general performance of a project, as well as the project-wide impact a particular change made instead of being limited to a kernel-specific scope.

Advanced Highlighting

The display of the source code of a kernel, along with its relevant intermediary representation enables powerful insights into the reasoning behind proposed optimizations, especially using

highlighting. While highlighting already provides significant value for users, it is mostly limited to the SASS and PTX intermediary representations and basic tokens like the relevant registers used in an instruction. Additional work could include trying to map the highlighting of certain tokens to concrete variables in the source code, which would dramatically improve the user experience for new and inexperienced users.

Additionally, providing the ability to highlight custom tokens could also provide benefits, for example for tracing multiple instances of registers spilling at the same time.

7. Conclusion

This thesis focused on creating a user interface for the GPUscout application, with the goal of displaying results in a more understandable and accessible manner, as well as providing several visualizations and data points not present in the original tool, further improving usability. Using the final application, both experienced users and those new to GPU and CUDA programming can now analyze kernels using GPUscout and view the findings and recommendations in a more user-friendly format compared to raw text. The application has also been implemented in a modular manner, enabling easy maintenance and modification to support further analyses and visualizations without the need of understanding the entire source code.

After introducing the relevant background in both GPU architecture and programming, as well as user interface design, the implementation of GPUscout-GUI is documented. In addition to a detailed explanation of the various interfaces, each component is thoroughly described, including information about certain design choices and the way data from GPUscout is translated and integrated to GPUscout-GUI. The focus on providing a satisfactory user experience for all target groups in new and experienced users is also emphasized.

The application was then evaluated using three projects of different complexity and structure, with the goal of verifying the correctness of the information displayed with respect to the results produced by GPUscout, as well demonstrating its potential to ease the process of understanding the presented information and taking the correct actions based on them.

The first project, *cuSZp*, a lossy compression tool specifically for floating-point data, which includes four kernels implementing compression and decompression using two separate encoding modes, was able to be optimized findings in two analyses leading to performance improvements between 3% and 12% depending on the configuration and hardware used. In this process, the additional visualizations in GPUscout-GUI provided a significantly better user experience than solely relying on the textual output of GPUscout.

A second project, *step-64* computes a solution of the Helmholtz equation using two frameworks, showed that the tools are not effective for every kernel and project written in CUDA, especially when using layers of abstraction. Even though the information generated by GPUscout was displayed correctly in GPUscout-GUI and provided significantly better insights into the results than the textual output, no concrete optimizations could be made due to difficulties with the project structure.

Lastly, an implementation of the Jacobi iteration in CUDA was considered, which GPUscout and GPUscout-GUI helped optimize to more than 150% in performance gains in certain cases. More importantly, this project showed the massive value in the graphical interface, especially in regard to the additional information provided, aiding users to more easily evaluate if a certain optimization is expected to bring improvements and thus avoid mistakes otherwise

not as apparent.

In summary, GPUscout-GUI adds extensive additional features to GPUscout, making it more accessible for certain user groups, as well as enhancing usability and intuitiveness significantly compared to the current textual output. As the data is presented in a more detailed manner and additional explanations are provided, both new and experienced users are be able to more easily find and optimize performance bottlenecks in their CUDA kernels.

Appendix

A. GPUscout JSON Result File Options

In this section, some parts of the JSON file generated by GPUscout are described in more detail.

A.1. Analysis

This section provides an overview of the analysis-specific data present in the JSON output of GPUscout. For each analysis, data points can either belong to a certain occurrence or be part of the analysis-specific metrics available to all occurrences. The following attributes are part of the occurrences of every analysis:

line_number occurrence integer	The relevant line number in the source code.
pc_offset occurrence string	The relevant address in the intermediary SASS code if the analysis is based on SASS.
line_number_raw occurrence string	The relevant address in the intermediary PTX code if the analysis is based on PTX.
severity occurrence string	The severity of the occurrence according to GPUscout. It can be set either to "WARNING" (this occurrence is related to some performance bottleneck) or "INFO" (This occurrence is related to an already implemented solution).

Table A.2.: GPUscout JSON output: Analysis-specific metrics (All analyses)

Datatype Conversion

type occurrence string	Specifies the type of the datatype conversion. It can be set either to "I2F" (Integer to Float), "F2I" (Float to Integer) or "F2F" (Float to Float).

Table A.4.: GPUscout JSON output: Analysis-specific metrics (Datatype conversion)

Deadlock Detection

deadlock_detect_flag metric boolean	Set to true if a deadlock has been detected in the kernel.

Table A.6.: GPUscout JSON output: Analysis-specific metrics (Deadlock detection)

Global Atomics

in_for_loop occurrence boolean	If the relevant atomic operation is executed in a for-loop.
is_global occurrence boolean	If the relevant atomic is a global atomic.

Table A.8.: GPUscout JSON output: Analysis-specific metrics (Global atomics)

Register Spilling

operation occurrence string	The operation executed in the SASS line of the spill.
register occurrence string	The name of the register that spilled.
register_pressure_increase occurrence integer	The register pressure increase since the last SASS instruction.
used_register_count occurrence integer	The number of live registers at the time of the spill.
previous_compute_instruction occurrence object	

instruction occurrence string	The previous compute instruction the spilled register was used in.
line_number occurrence integer	The relevant line number in the source code of this compute instruction.
pc_offset occurrence string	The relevant line number in the SASS code of this compute instruction.

Table A.10.: GPUscout JSON output: Analysis-specific metrics (Register spilling)

Use Restrict

read_only_memory_used occurrence boolean	If the relevant SASS line already uses read-only memory.
register occurrence string	The name of the relevant register.
register_pressure_increase occurrence integer	The register pressure increase since the last SASS instruction.
used_register_count occurrence integer	The number of live registers at the time of the spill.

Table A.12.: GPUscout JSON output: Analysis-specific metrics (Use restrict)

Use Shared

computation_instruction_count occurrence integer	The number of computation instructions the relevant register was used in.
computation_instruction_pc_offsets occurrence array	The SASS addresses of these computation instructions.

global_load_count occurrence integer	The number of global load instructions the relevant register was used in.
global_load_pc_offsets occurrence array	The SASS addresses of these global load instructions.
in_for_loop occurrence boolean	If the shared memory instruction was executed inside a for-loop.
register occurrence string	The name of the relevant register.
uses_shared_memory occurrence boolean	If the relevant SASS line already uses shared memory.

Table A.14.: GPUscout JSON output: Analysis-specific metrics (Use shared)

Use Texture

read_register occurrence string	The name of the register that data was read from.
spatial_locality occurrence boolean	If spatial locality among the read addresses has been found.
unroll_pc_offsets occurrence array	The unrolls of the different reads.
written_register occurrence string	The name of the register that data was written to.

Table A.16.: GPUscout JSON output: Analysis-specific metrics (Use texture)

Vectorization

register occurrence string	The name of the relevant register.
register_load_type occurrence integer	The type of load operation the register was involved in. It can be set to 0 (32-bit load), 1 (64-bit load), or 2 (128-bit load)
unroll_pc_offsets occurrence array	The unrolls of the different reads.
adjacent_memory_accesses occurrence number	The number of adjacent memory accesses found.

Table A.18.: GPUscout JSON output: Analysis-specific metrics (Vectorization)

Warp Divergence

target_branch occurrence string	The name of the target branch.
target_branch_start_line_number occurrence integer	The source code line number of the first line of the target branch.
branch_divergence_perc metric integer	The percentage of branches found that diverge. METRIK

Table A.20.: GPUscout JSON output: Analysis-specific metrics (Warp divergence)

A.2. Metrics

The following metrics are available globally in the JSON output of GPUscout:

Key	Description
l2_cache_hit_perc	L2 cache hit percentage
l2_queries	Number of L2 cache queries
loads_l2_cache_hit_perc	L2 cache hit percentage for loads
loads_l2_to_dram_bytes	Data transferred from L2 to DRAM in bytes for loads
stores_l2_cache_hit_perc	L2 cache hit percentage for stores
stores_l2_to_dram_bytes	Data transferred from L2 to DRAM in bytes for stores
total_instructions	Total number of instructions

Table A.21.: GPUscout JSON output: Metrics (General)

Key	Description
atomic_l1_cache_hit_perc	L1 cache hit percentage for atomics
atomic_to_l1_bytes	Bytes transferred from atomic operations to L1
atomics_l1_to_l2_bytes	Bytes transferred from L1 to L2 for atomics
atomics_l2_cache_hit_perc	L2 cache hit percentage for atomics
atomics_l2_to_dram_bytes	Bytes transferred from L2 to DRAM for atomics
bytes_per_instruction	Average bytes per instruction
instructions	Total number of global instructions
loads_instructions	Total number of load instructions
loads_l1_cache_hit_perc	L1 cache hit percentage for loads
loads_l1_to_l2_bytes	Bytes transferred from L1 to L2 for loads
loads_to_l1_bytes	Bytes transferred from DRAM to L1 for loads
stores_instructions	Total number of store instructions
stores_l1_cache_hit_perc	L1 cache hit percentage for stores
stores_l1_to_l2_bytes	Bytes transferred from L1 to L2 for stores
stores_to_l1_bytes	Bytes transferred from DRAM to L1 for stores

Table A.22.: GPUscout JSON output: Metrics (Global memory & atomics)

Key	Description
instructions	Total number of local instructions
l2_queries_perc	L2 queries percentage for local memory
loads_instructions	Total number of load instructions in local memory
loads_l1_cache_hit_perc	L1 cache hit percentage for local loads
loads_l1_to_l2_bytes	Bytes transferred from L1 to L2 for local loads
loads_to_l1_bytes	Bytes transferred from DRAM to L1 for local loads
stores_instructions	Total number of store instructions in local memory
stores_l1_cache_hit_perc	L1 cache hit percentage for local stores
stores_l1_to_l2_bytes	Bytes transferred from L1 to L2 for local stores
stores_to_l1_bytes	Bytes transferred from DRAM to L1 for local stores

Table A.23.: GPUscout JSON output: Metrics (Local memory)

Key	Description
sm__warps_active	Number of active warps on SM
barrier_per_warp_active[1]	Warp stalls due to barrier issues per active warp
imc_miss_per_warp_active[1]	Warp stalls due to IMC misses per active warp
lg_throttle_per_warp_active[1]	Warp stalls due to large thread throttle per active warp
long_scoreboard_per_warp_active[1]	Warp stalls due to long scoreboard per active warp
membar_per_warp_active[1]	Warp stalls due to memory barrier per active warp
mio_throttle_per_warp_active[1]	Warp stalls due to memory I/O throttle per active warp
short_scoreboard_per_warp_active[1]	Warp stalls due to short scoreboard per active warp
tex_throttle_per_warp_active[1]	Warp stalls due to texture throttle per active warp
wait_per_warp_active[1]	Warp stalls due to wait states per active warp

Table A.24.: GPUscout JSON output: Metrics (Stalls)

[1]The prefix of smsp__warp_issue_stalled_ has been truncated for brevity

Key	Description
instructions	Total number of shared memory instructions
ldgsts_instructions	Total number of load-store instructions in shared memory
loads_bank_conflict	Bank conflict for shared memory loads
loads_efficiency_perc	Efficiency percentage for shared memory loads
loads_instructions	Total number of load instructions in shared memory
stores_instructions	Total number of store instructions in shared memory

Table A.25.: GPUscout JSON output: Metrics (Shared memory)

Key	Description
instructions	Total number of texture memory instructions
loads_l1_cache_hit_perc	L1 cache hit percentage for texture loads
loads_l1_to_l2_bytes	Bytes transferred from L1 to L2 for texture loads
loads_l2_to_dram_bytes	Bytes transferred from L2 to DRAM for texture loads
loads_to_l1_bytes	Bytes transferred from DRAM to L1 for texture loads

Table A.26.: GPUscout JSON output: Metrics (Texture memory)

B. Implementation Changes during Evaluation

This chapter documents the detailed changes made after each optimization step of the projects analyzed in the evaluation chapter.

B.1. Jacobi Iteration

Use Restrict Analysis

Based on findings in the *Use Restrict* analysis, the matrix `A` and vector `x` have been marked with the `__restrict__` keyword.

```
1 __global__ void jacobi(const float * __restrict__ A, const float *b, float *
    ↪ __restrict__ x, float *xNew, int N) {
2     int row = blockIdx.x * blockDim.x + threadIdx.x;
3
4     float sum = 0.0f;
5     for (int col = 0; col < N; ++col) {
6         if (col != row) {
7             sum += A[row * N + col] * x[col];
8         }
9     }
10    xNew[row] = (b[row] - sum) / A[row * N + row];
11 }
```

Listing B.1: Jacobi Iteration: Kernel after Use Restrict

Use Shared Analysis

Based on findings in the *Use Shared* analysis, shared memory has been implemented to be used for the vector `x`.

```
1 __global__ void jacobi(const float * __restrict__ A, const float* b, float *
    ↪ __restrict__ x, float* xNew, int N) {
2     int row = blockIdx.x * blockDim.x + threadIdx.x;
3
4     extern __shared__ float sharedX[];
5     sharedX[row] = x[row];
6     __syncthreads();
7
```

```
8     float sum = 0.0f;
9     for (int col = 0; col < N; ++col) {
10        if (col != row) {
11            sum += A[row * N + col] * sharedX[col];
12        }
13    }
14    xNew[row] = (b[row] - sum) / A[row * N + row];
15 }
```

Listing B.2: Jacobi Iteration: Kernel after Use Shared

Vectorization Analysis

Based on findings in the *Vectorization* analysis, vectorized loads have been implemented to be used during the iteration over rows of the matrix `A`.

```
1  __global__ void jacobi(const float * __restrict__ A, const float* b, float *
       ↪ __restrict__ x, float* xNew, int N) {
2     int row = blockIdx.x * blockDim.x + threadIdx.x;
3
4     __shared__ float sharedX[512];
5     sharedX[row] = x[row];
6     __syncthreads();
7
8     float sum = 0.0f;
9     for (int col = 0; col < N; col += 4) {
10        float4 xVec = *reinterpret_cast<const float4 *>(&sharedX[col]);
11        float4 AVec = *reinterpret_cast<const float4 *>(&A[row * N + col]);
12        sum += (col != row) ? AVec.x * xVec.x : 0.0f;
13        sum += (col + 1 != row) ? AVec.y * xVec.y : 0.0f;
14        sum += (col + 2 != row) ? AVec.z * xVec.z : 0.0f;
15        sum += (col + 3 != row) ? AVec.w * xVec.w : 0.0f;
16    }
17    xNew[row] = (b[row] - sum) / A[row * N + row];
18 }
```

Listing B.3: Jacobi Iteration: Kernel after Vectorization

B.2. CuSZp

B.2.1. Compression kernels

Datatype Conversion Analysis

Based on findings in the *Datatype Conversion* analysis for both compression kernels, the `quantization` function has been modified to use native functions to perform the datatype conversion while rounding correctly.

```
*** 3,12 ****
  __device__ inline int quantization(double data, double recipPrecision)
  {
      double dataRecip = data*recipPrecision;
! int s = dataRecip>=-0.5?0:1;
! return (int)(dataRecip+0.5) - s;
  }
--- 3,12 ----
  __device__ inline int quantization(double data, double recipPrecision)
  {
      double dataRecip = data*recipPrecision;
! return __double2int_rn(dataRecip);
  }
```

Listing B.4: cuSZp: Compression kernel changes after Datatype Conversion

Register Spilling Analysis

Based on findings in the *Register Spilling* analysis for both compression kernels, the use of loop unrolling for the main for-loop in both kernels has been limited to unrolling 2 instead of 8 executions at a time.

```
*************** __global__ void cuSZp_compress_kernel_outlier_f64
*** 51,57 ****
          base_block_start_idx = base_start_idx + j * 1024 + lane * 32;
          base_block_end_idx = base_block_start_idx + 32;
          sign_flag[j] = 0;
- fixed_rate[j] = 0;
          block_idx = base_block_start_idx/32;
          prevQuant = 0;
          int maxQuant = 0;
*** 60,66 ****

          if(base_block_end_idx < nbEle)
          {
```

```
14 ! #pragma unroll 8
15              for(int i=base_block_start_idx; i<base_block_end_idx; i+=4)
16              {
17                  tmp_buffer = reinterpret_cast<const double4*>(oriData)[i/4];
18 --- 60,66 ----
19
20          if(base_block_end_idx < nbEle)
21          {
22 ! #pragma unroll 2
23              for(int i=base_block_start_idx; i<base_block_end_idx; i+=4)
24              {
25                  tmp_buffer = reinterpret_cast<const double4*>(oriData)[i/4];
26
27 *************** __global__ void cuSZp_compress_kernel_plain_f64
28 *** 1270,1276 ****
29
30          if(base_block_end_idx < nbEle)
31          {
32 ! #pragma unroll 8
33              for(int i=base_block_start_idx; i<base_block_end_idx; i+=4)
34              {
35                  tmp_buffer = reinterpret_cast<const double4*>(oriData)[i/4];
36 --- 1272,1278 ----
37
38          if(base_block_end_idx < nbEle)
39          {
40 ! #pragma unroll 2
41              for(int i=base_block_start_idx; i<base_block_end_idx; i+=4)
42              {
43                  tmp_buffer = reinterpret_cast<const double4*>(oriData)[i/4];
```

Listing B.5: cuSZp: Compression kernel changes after Register Spilling

B.2.2. Decompress Kernels

Datatype Conversion Analysis

No changes in decompression kernels were performed based on the *Datatype Conversion* analysis

Register Spilling Analysis

Based on findings in the *Register Spilling* analysis for both compression kernels, the `fixed_rate` array has been replaced by local variables initialized in each loop iteration in both kernels.

```
1 *************** __global__ void cuSZp_decompress_kernel_outlier_f64
2 *** 659,665 ****
3       int absQuant[32];
4       int currQuant, lorenQuant, prevQuant;
5       int sign_ofs;
6 - int fixed_rate[block_num];
7       unsigned int thread_ofs = 0;
8       uchar4 tmp_char;
9       double4 dec_buffer;
10 *** 667,677 ****
11      for(int j=0; j<block_num; j++)
12      {
13          block_idx = warp * dec_chunk + j * 32 + lane;
14 ! fixed_rate[j] = (int)cmpData[block_idx];
15
16 ! int encoding_selection = fixed_rate[j] >> 7;
17 ! int outlier = ((fixed_rate[j] & 0x60) >> 5) + 1;
18 ! int temp_rate = fixed_rate[j] & 0x1f;
19          if(!encoding_selection) thread_ofs += temp_rate ? (4 + temp_rate * 4)
                ↪ : 0;
20          else thread_ofs += 4 + temp_rate * 4 + outlier;
21          __syncthreads();
22 --- 643,653 ----
23      for(int j=0; j<block_num; j++)
24      {
25          block_idx = warp * dec_chunk + j * 32 + lane;
26 ! int fixed_rate = (int)cmpData[block_idx];
27
28 ! int encoding_selection = fixed_rate >> 7;
29 ! int outlier = ((fixed_rate & 0x60) >> 5) + 1;
30 ! int temp_rate = fixed_rate & 0x1f;
31          if(!encoding_selection) thread_ofs += temp_rate ? (4 + temp_rate * 4)
                ↪ : 0;
32          else thread_ofs += 4 + temp_rate * 4 + outlier;
33          __syncthreads();
34 *** 751,766 ****
35      base_start_idx = warp * dec_chunk * 32;
36      for(int j=0; j<block_num; j++)
37      {
38 ! int encoding_selection = fixed_rate[j] >> 7;
39 ! int outlier_byte_num = ((fixed_rate[j] & 0x60) >> 5) + 1;
40 ! fixed_rate[j] &= 0x1f;
```

```
41          int outlier_buffer = 0;
42          base_block_start_idx = base_start_idx + j * 1024 + lane * 32;
43          base_block_end_idx = base_block_start_idx + 32;
44          unsigned int sign_flag = 0;
45
46 ! if(!encoding_selection) tmp_byte_ofs = (fixed_rate[j]) ? (4+fixed_rate[j]*4)
       ↪ : 0;
47 ! else tmp_byte_ofs = 4 + outlier_byte_num + fixed_rate[j] * 4;
48          #pragma unroll 5
49          for(int i=1; i<32; i<<=1)
50          {
51 --- 727,745 ----
52       base_start_idx = warp * dec_chunk * 32;
53       for(int j=0; j<block_num; j++)
54       {
55 ! block_idx = warp * dec_chunk + j * 32 + lane;
56 ! int fixed_rate = (int)cmpData[block_idx];
57 !
58 ! int encoding_selection = fixed_rate >> 7;
59 ! int outlier_byte_num = ((fixed_rate & 0x60) >> 5) + 1;
60 ! fixed_rate &= 0x1f;
61          int outlier_buffer = 0;
62          base_block_start_idx = base_start_idx + j * 1024 + lane * 32;
63          base_block_end_idx = base_block_start_idx + 32;
64          unsigned int sign_flag = 0;
65
66 ! if(!encoding_selection) tmp_byte_ofs = (fixed_rate) ? (4+fixed_rate*4) : 0;
67 ! else tmp_byte_ofs = 4 + outlier_byte_num + fixed_rate * 4;
68          #pragma unroll 5
69          for(int i=1; i<32; i<<=1)
70          {
71 *************** __global__ void cuSZp_decompress_kernel_plain_f64
72 *** 1523,1529 ****
73       int absQuant[32];
74       int currQuant, lorenQuant, prevQuant;
75       int sign_ofs;
76 - int fixed_rate[block_num];
77       unsigned int thread_ofs = 0;
78       uchar4 tmp_char;
79       double4 dec_buffer;
80 *** 1531,1538 ****
81       for(int j=0; j<block_num; j++)
```

```
82      {
83          block_idx = warp * dec_chunk + j * 32 + lane;
84 ! fixed_rate[j] = (int)cmpData[block_idx];
85 ! thread_ofs += (fixed_rate[j]) ? (4+fixed_rate[j]*4) : 0;
86          __syncthreads();
87      }
88
89 --- 1502,1509 ----
90      for(int j=0; j<block_num; j++)
91      {
92          block_idx = warp * dec_chunk + j * 32 + lane;
93 ! int fixed_rate = (int)cmpData[block_idx];
94 ! thread_ofs += (fixed_rate) ? (4+fixed_rate*4) : 0;
95          __syncthreads();
96      }
97  *** 1610,1620 ****
98      base_start_idx = warp * dec_chunk * 32;
99      for(int j=0; j<block_num; j++)
100     {
101         base_block_start_idx = base_start_idx + j * 1024 + lane * 32;
102         base_block_end_idx = base_block_start_idx + 32;
103         unsigned int sign_flag = 0;
104
105 ! tmp_byte_ofs = (fixed_rate[j]) ? (4+fixed_rate[j]*4) : 0;
106         #pragma unroll 5
107         for(int i=1; i<32; i<<=1)
108         {
109 --- 1581,1594 ----
110     base_start_idx = warp * dec_chunk * 32;
111     for(int j=0; j<block_num; j++)
112     {
113 + block_idx = warp * dec_chunk + j * 32 + lane;
114 + int fixed_rate = (int)cmpData[block_idx];
115 +
116         base_block_start_idx = base_start_idx + j * 1024 + lane * 32;
117         base_block_end_idx = base_block_start_idx + 32;
118         unsigned int sign_flag = 0;
119
120 ! tmp_byte_ofs = (fixed_rate) ? (4+fixed_rate*4) : 0;
121         #pragma unroll 5
122         for(int i=1; i<32; i<<=1)
```

```
123             {
```

Listing B.6: cuSZp: Decompression kernel changes after Register Spilling

Following these changes, all further accesses of `fixed_rate[i]` were replaced by usages of the local `fixed_rate` variable in the respective kernels.

List of Figures

List of Tables

List of Listings

Acronyms

AI Artificial Intelligence.

CAPS Chair of Computer Architecture and Parallel Systems.

CLI Command-Line Interface.

CPU Central Processing Unit.

CTA Cooperative Thread Array.

CUDA Compute Unified Device Architecture.

DRAM Dynamic Random Access Memory.

GPU Graphics Processing Unit.

GUI Graphical User Interface.

HPC High-Performance Computing.

ISA Instruction Set Architecture.

JIT Just-in-time.

LLM Large Language Model.

MPI Message-Passing Interface.

PC Program Counter.

PTX Parallel Thread Execution.

SASS Streaming Assembler.

SIMT Single Instruction, Multiple Threads.

SM Streaming Multiprocessor.

UI User Interface.

UX User Experience.

Bibliography

[1] NVIDIA. *CUDA C++ Programming Guide*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html# (visited on 12/09/2024).

[2] D. Youvan. *Parallel Precision: The Role of GPUs in the Acceleration of Artificial Intelligence*. 2023. DOI: 10.13140/RG.2.2.21937.76641. Pre-published.

[3] NVIDIA. *Nsight Compute, Release 12.6*. URL: https://docs.nvidia.com/nsight-compute/NsightCompute/index.html#standalone-source-viewer (visited on 08/13/2024).

[4] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr. "The Scalasca Performance Toolset Architecture". In: *Concurrency and Computation: Practice and Experience* 22.6 (2010), pp. 702–719. ISSN: 1532-0634. DOI: 10.1002/cpe.1556. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1556 (visited on 07/11/2024).

[5] K. Zhou, L. Adhianto, J. Anderson, A. Cherian, D. Grubisic, M. Krentel, Y. Liu, X. Meng, and J. Mellor-Crummey. "Measurement and Analysis of GPU-accelerated Applications with HPCToolkit". In: *Parallel Computing* 108 (Dec. 1, 2021), p. 102837. ISSN: 0167-8191. DOI: 10.1016/j.parco.2021.102837. URL: https://www.sciencedirect.com/science/article/pii/S0167819121000843 (visited on 07/11/2024).

[6] S. S. Shende and A. D. Malony. "The Tau Parallel Performance System". In: *The International Journal of High Performance Computing Applications* 20.2 (May 1, 2006), pp. 287–311. ISSN: 1094-3420. DOI: 10.1177/1094342006064482. URL: https://doi.org/10.1177/1094342006064482 (visited on 07/11/2024).

[7] S. Sen, S. Vanecek, and M. Schulz. "GPUscout: Locating Data Movement-related Bottlenecks on GPUs". In: *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. SC-W '23. New York, NY, USA: Association for Computing Machinery, Nov. 12, 2023, pp. 1392–1402. ISBN: 9798400707858. DOI: 10.1145/3624062.3624208. URL: https://doi.org/10.1145/3624062.3624208 (visited on 07/11/2024).

[8] NVIDIA. *Nvidia Ampere GA102 GPU Architecture*. URL: https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf (visited on 07/31/2024).

[9] NVIDIA. *Nvidia GP100 Pascal Whitepaper*. URL: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf (visited on 08/13/2024).

[10] NVIDIA. *Nvidia H100 Tensor Core GPU Architecture*. URL: https://resources.nvidia.com/en-us-tensor-core (visited on 07/31/2024).

[11] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. "NVIDIA Tesla: A Unified Graphics and Computing Architecture". In: *IEEE Micro* 28.2 (Mar. 2008), pp. 39–55. ISSN: 1937-4143. DOI: 10.1109/MM.2008.31. URL: https://ieeexplore.ieee.org/abstract/document/4523358 (visited on 07/31/2024).

[12] NVIDIA. *Nvidia Turing GPU Architecture*. URL: https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf (visited on 07/31/2024).

[13] NVIDIA. *Nvidia's next Generation Cuda Compute Architecture: Fermi*. URL: https://www.nvidia.de/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf (visited on 07/31/2024).

[14] NVIDIA. *CUDA C++ Best Practices Guide*. URL: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/ (visited on 08/13/2024).

[15] NVIDIA. *Kernel Profiling Guide, Release 12.6*. URL: https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html (visited on 08/13/2024).

[16] X. Chen, L.-W. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W.-M. Hwu. "Adaptive Cache Management for Energy-Efficient GPU Computing". In: *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture. Dec. 2014, pp. 343–355. DOI: 10.1109/MICRO.2014.11. URL: https://ieeexplore.ieee.org/document/7011400 (visited on 08/05/2024).

[17] NVIDIA. *GPU Performance Background User's Guide*. NVIDIA Docs. URL: https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html (visited on 08/13/2024).

[18] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA". In: *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP08: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. Salt Lake City UT USA: ACM, Feb. 20, 2008, pp. 73–82. ISBN: 978-1-59593-795-7. DOI: 10.1145/1345206.1345220. URL: https://dl.acm.org/doi/10.1145/1345206.1345220 (visited on 07/23/2024).

[19] M. Sadrosadati, A. Mirhosseini, B. Ehsani, H. Sarbazi-Azad, M. Drumond, B. Falsafi, R. Ausavarungnirun, and O. Mutlu. "LTRF: Enabling High-Capacity Register Files for GPUs via Hardware/Software Cooperative Register Prefetching". In: *ACM SIGPLAN Notices* 53 (Mar. 19, 2018), pp. 489–502. ISSN: 978-1-4503-4911-6. DOI: 10.1145/3296957.3173211.

[20] M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally. "Unifying Primary Cache, Scratch, and Register File Memories in a Throughput Processor". In: *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture. Dec. 2012, pp. 96–106. DOI: 10.1109/MICRO.2012.18. URL: https://ieeexplore.ieee.org/document/6493611 (visited on 08/02/2024).

[21] NVIDIA. *Hopper Tuning Guide, Release 12.6*. URL: https://docs.nvidia.com/cuda/hopper-tuning-guide/index.html (visited on 08/13/2024).

[22] M. A. Al-Mouhamed, A. H. Khan, and N. Mohammad. "A Review of CUDA Optimization Techniques and Tools for Structured Grid Computing". In: *Computing* 102.4 (Apr. 1, 2020), pp. 977–1003. ISSN: 1436-5057. DOI: 10.1007/s00607-019-00744-1. URL: https://doi.org/10.1007/s00607-019-00744-1 (visited on 07/16/2024).

[23] S. Franey and M. Lipasti. "Accelerating Atomic Operations on GPGPUs". In: *2013 Seventh IEEE/ACM International Symposium on Networks-on-Chip (NoCS)*. 2013 Seventh IEEE/ACM International Symposium on Networks-on-Chip (NoCS). Tempe, AZ, USA: IEEE, Apr. 2013, pp. 1–8. ISBN: 978-1-4673-6493-5 978-1-4673-6491-1 978-1-4673-6492-8. DOI: 10.1109/NoCS.2013.6558404. URL: http://ieeexplore.ieee.org/document/6558404/ (visited on 08/05/2024).

[24] NVIDIA. *Parallel Thread Execution ISA Version 8.5*. URL: https://docs.nvidia.com/cuda/parallel-thread-execution/ (visited on 08/13/2024).

[25] NVIDIA. *NVIDIA CUDA Compiler Driver NVCC*. URL: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/ (visited on 08/13/2024).

[26] NVIDIA. *Nvidia Tesla V100 GPU Architecture*. 2017. URL: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf (visited on 01/04/2025).

[27] NVIDIA. *CUDA Binary Utilities, Release 12.6*. URL: https://docs.nvidia.com/cuda/cuda-binary-utilities/ (visited on 08/13/2024).

[28] NVIDIA. *CUDA C Best Practices Guide*. URL: https://docs.nvidia.com/cuda/archive/9.0/pdf/CUDA_C_Best_Practices_Guide.pdf (visited on 08/13/2024).

[29] S. Sen. "Discovering Data Movement-Related Bottlenecks on NVIDIA GPUs". In: *Computational Science and Engineering* (Apr. 18, 2023).

[30] P. Ralph and Y. Wand. "A Proposal for a Formal Definition of the Design Concept". In: *Design Requirements Engineering: A Ten-Year Perspective*. Ed. by K. Lyytinen, P. Loucopoulos, J. Mylopoulos, and B. Robinson. Red. by W. Van Der Aalst, J. Mylopoulos, N. M. Sadeh, M. J. Shaw, and C. Szyperski. Vol. 14. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 103–136. ISBN: 978-3-540-92965-9 978-3-540-92966-6. DOI: 10.1007/978-3-540-92966-6_6. URL: http://link.springer.com/10.1007/978-3-540-92966-6_6 (visited on 07/30/2024).

[31] E. McKay. *UI Is Communication*. O'Reilly Media, May 2013. ISBN: 978-0-12-396980-4. URL: https://learning.oreilly.com/library/view/ui-is-communication/9780123969804/ (visited on 07/16/2024).

[32] J. Johnson. *Designing with the Mind in Mind: Simple Guide to Understanding User Interface Design Rules*. Amsterdam ; Boston: Morgan Kaufmann Publishers/Elsevier, 2010. 186 pp. ISBN: 978-0-12-375030-3.

[33] J. Nielsen. "Enhancing the Explanatory Power of Usability Heuristics". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '94. New York, NY, USA: Association for Computing Machinery, Apr. 24, 1994, pp. 152–158. ISBN: 978-0-89791-650-9. DOI: 10.1145/191666.191729. URL: https://dl.acm.org/doi/10.1145/191666.191729 (visited on 07/30/2024).

[34] A. S. Badashian, M. Mahdavi, A. Pourshirmohammadi, and M. M. nejad. "Fundamental Usability Guidelines for User Interface Design". In: *2008 International Conference on Computational Sciences and Its Applications*. 2008 International Conference on Computational Sciences and Its Applications. June 2008, pp. 106–113. DOI: 10.1109/ICCSA.2008.45. URL: https://ieeexplore.ieee.org/abstract/document/4561210 (visited on 07/16/2024).

[35] R. Nacheva. "Principles of User Interface Design: Important Rules That Every Designer Should Follow". In: Oct. 30, 2015. DOI: 10.13140/RG.2.1.5148.8083.

[36] J. Cabrera. *Modular Design Frameworks: A Projects-Based Guide for UI/UX Designers*. Berkeley, CA, UNITED STATES: Apress L. P., 2017. ISBN: 978-1-4842-1688-0. URL: http://ebookcentral.proquest.com/lib/munchentech/detail.action?docID=4944166 (visited on 07/16/2024).

[37] R. Bell, A. D. Malony, and S. Shende. "ParaProf: A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis". In: *Euro-Par 2003 Parallel Processing*. Ed. by H. Kosch, L. Böszörményi, and H. Hellwagner. Vol. 2790. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 17–26. ISBN: 978-3-540-40788-1 978-3-540-45209-6. DOI: 10.1007/978-3-540-45209-6_7. URL: http://link.springer.com/10.1007/978-3-540-45209-6_7 (visited on 08/13/2024).

[38] L. Adhianto. *HPCToolkit Graphical User Interface*. URL: https://www.nersc.gov/assets/Uploads/HPCToolkit-GUI.pdf (visited on 07/16/2024).

[39] R. S. White. *CudaPAD*. CodeProject. 6/14/2015 1:13:00 PM. URL: https://www.codeproject.com/Articles/999744/CudaPAD (visited on 12/09/2024).

[40] Y. Huang, S. Di, X. Yu, G. Li, and F. Cappello. "cuSZp: An Ultra-fast GPU Error-bounded Lossy Compression Framework with Optimized End-to-End Performance". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '23. New York, NY, USA: Association for Computing Machinery, Nov. 11, 2023, pp. 1–13. ISBN: 9798400701092. DOI: 10.1145/3581784.3607048. URL: https://dl.acm.org/doi/10.1145/3581784.3607048 (visited on 02/08/2025).