SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY

TECHNICAL UNIVERSITY MUNICH

Master's Thesis in Informatics

Extending GPUscout analysis to AMD GPUs

Benedikt Deike

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY

TECHNICAL UNIVERSITY MUNICH

Master's Thesis in Informatics

Extending GPUscout analysis to AMD GPUs

Author:	Benedikt Deike
Examiner:	Prof. Dr. rer. nat. Martin Schulz
Advisor:	M.Sc. Stepan Vanecek
Submission Date:	14th February 2025

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

Munich, 14th February 2025 Benedikt Deike

Abstract

Programming for GPUs is significantly more complex than programming for CPUs, as developers must independently address various performance challenges due to the lack of a mature ecosystem for identifying and resolving bottlenecks in GPU-accelerated programs. To enrich this ecosystem, GPUscout has been developed as a tool for detecting memory system bottlenecks, which are among the most common performance issues in GPU programs. Focusing on the memory system and hierarchy is a strategic choice, as many modern GPU workloads are constrained by memory bandwidth and latency rather than raw computing power. To date, GPUscout has only been available for the NVIDIA ecosystem, a limitation that this thesis aims to overcome. This work presents an effort to make GPUscout compatible with AMD hardware, providing a detailed account of the translation process from the NVIDIA to the AMD architecture. The thesis systematically describes how two of the three key components of GPUscout, assembly parsing and performance metrics collection, were successfully ported to the AMD ecosystem. Although the third component, PC stalling information, could not be implemented due to tool limitations, a roadmap for future implementation is provided. The resulting utility, AMD GPUscout, is presented through several practical examples that demonstrate its ability to identify memory related performance issues on AMD GPUs. This work not only extends the capabilities of GPUscout to a new hardware ecosystem, but also contributes to the broader effort of building a more comprehensive GPU performance analysis toolset to help developers effectively optimize their applications for AMD GPUs.

Contents

1 Introduction

Before delving into this thesis, it is advisable to first read Soumya Sen's thesis titled "Discovering Data Movement-Related Bottlenecks on NVIDIA GPUs" [1]. In this thesis, Sen provides a comprehensive explanation of the development of GPUscout, detailing its creation and the design choices underlying its implementation. Building on this work, a subsequent paper by Soumya Sen, Stepan Vanecek, and Martin Schulz titled "GPUscout: Locating Data Movement-Related Bottlenecks on GPUs" [2], was published. This paper serves as a condensed version of the research presented in Sen's thesis. Readers are strongly encouraged to familiarize themselves with these two publications before proceeding with this thesis.

1.1 Motivation

GPUscout is a powerful profiling tool designed to help developers optimize the performance of GPU-accelerated programs by identifying memory and cache related bottlenecks. Initially, GPUscout was built exclusively for NVIDIA GPUs, a logical choice given NVIDIA's dominance in the GPU market. Most developers optimizing their programs with third-party tools are likely using NVIDIA hardware, making it a natural starting point.

However, AMD has been making significant strides in the high-performance computing (HPC) sector, positioning itself as a serious competitor. Through active open-source development, AMD has introduced robust auxiliary software like the ROCm Compute Profiler and ROCprofiler-SDK, which provide valuable insights into GPU performance. These advancements make AMD an increasingly viable platform for GPUscout, paving the way for its expansion beyond the NVIDIA ecosystem.

Beyond AMD's ambitions, extending GPUscout's compatibility to additional hardware platforms benefits the project itself. While NVIDIA remains the industry leader, a subset of developers, such as those using the BEAST cluster at LRZ, work with AMD GPUs. Broadening GPUscout's support to include AMD hardware ensures that more developers can leverage its profiling capabilities, ultimately making it a more versatile and impactful tool.

1.2 Parallel Computing

Parallel computing aims to significantly speed up the execution of a program by dividing the task among multiple processors. However, the potential speedup of a parallel program is fundamentally limited by the fraction of the program that can be parallelized. This limitation is described by Amdahl's law. It states that if a program consists of a parallelizable fraction

(f) and a serial fraction (1 - f), the maximum speedup achievable using N processors is:

$$\text{Speedup} = \frac{1}{(1-f) + \frac{f}{N}}$$

As the number of processors approaches infinity, the speedup is constrained by the serial portion of the program, meaning that even with infinite computing resources, the execution time cannot be reduced beyond the time taken by the non-parallelizable part. This insight highlights that effective parallelization is not just about adding more processors but about minimizing serial operations. Amdahl's Law is pivotal for GPU programming, as GPUs excel in highly parallel tasks, but their performance rapidly diminishes when faced with serial bottlenecks or tasks with poor parallelization potential. Therefore, understanding and adhering to the principles of Amdahl's Law is crucial when designing parallel algorithms for GPU architectures, as it ensures the highest possible utilization of their massively parallel cores. [3]

To understand how parallel programs are structured and executed, it is beneficial to explore Flynn's taxonomy, which categorizes computer architectures based on their ability to handle multiple instructions and data streams. Flynn's taxonomy is a framework that divides computer systems into four categories and is shown in Table 1.1. [4]

	Single Data Stream	**Multiple Data Stream**
Single Instruction	SISD: Traditional von Neumann architecture	SIMD: Parallel processing on multiple data
Multiple Instructions	MISD: Multiple instructions on one data	MIMD: Multi-core and distributed systems

Table 1.1: Flynn's taxonomy of computer architectures.

Among these architectures, SIMD (Single Instruction, Multiple Data) is particularly relevant to GPU programming because it maximizes data-level parallelism. SIMD architectures can be divided into two primary forms. Array processors execute the same instruction across multiple processing elements (PEs) simultaneously in space, each operating on different data elements. Vector processors execute the same instruction on multiple data elements sequentially in time using a single PE, but over successive time steps. Modern GPUs are essentially a combination of both forms, hundreds of highly pipelined PEs, performing computations parallel across time and space. [5]

A modern GPU typically consists of thousands of CUDA cores (in NVIDIA GPUs) or Stream Processors (in AMD GPUs), each designed to perform SIMD operations. The GPU's memory hierarchy and thread scheduling model are optimized to ensure that large groups of threads (warps or wavefronts) execute identical instructions in lockstep, maximizing throughput.

2 Background

2.1 HIP Programming Model

There are many programming models that are suitable for a heterogeneous system (e.g. GPU + CPU), such as OpenMP or OpenCL. However, NVIDIA GPUscout was developed with a focus on CUDA (Compute Unified Device Architecture), so the choice of programming model supported by AMD GPUscout was HIP (Heterogeneous-computing Interface for Portability), AMD's attempt to provide a programming model that is as close to CUDA as possible.

However, the programming model is ultimately not that important because NVIDIA and AMD GPUscout examine assembly code most of the time, which should be identical regardless of the programming model.

Like CUDA, HIP is an extension of the C and C++ programming languages. HIP is similar to the Single-Program Multiple-Data (SPMD) programming model in that a HIP programmer writes a single serial program that calls parallel functions, called kernels, which are executed in parallel by many functional units and therefore process many data elements in parallel. The serial parts of a HIP program run on the CPU, while the various kernels run on the GPU. Kernels use the concept of threads to express parallelism. A programmer who is aware of the data-level parallelism in their computation can define multiple threads in a kernel that work in parallel. These threads all share the same program logic. Each thread works independently on a single scalar data value. In order for threads to cooperate with each other, for example by sharing access to a memory area, there is the concept of thread blocks. Threads that share the same thread block can cooperate. A thread block consists of multiple concurrent threads. Each thread in a thread block has a unique thread ID (TID), which ranges from 1 to the size of the host thread block. The programmer must declare the shape of the thread block (1D, 2D or 3D). Varying the thread block dimensionality allows programmers to match the thread block dimensions to the inherent dimensions of the data, resulting in a more natural and efficient mapping of data to threads. Depending on the thread block dimensionality, a thread TID has one, two, or three dimensions. Thread blocks are grouped in grids. Grids are also defined to have up to three dimensions. Different thread blocks in a grid are not intended to communicate with each other, they (the thread blocks) are intended to run independently of each other. Each thread block in a grid has a unique block ID. Sequential grids (generated by sequential kernels) can be enforced by synchronization barriers. [6, p. 11-24]

Since AMD wanted to make HIP attractive to CUDA programmers, the HIP concepts are almost identical to those of CUDA. Only a few basic terms differ, which are compared in Table 2.1.

Warps or wavefronts in Table 2.1 have less to do with the programming model and more to do with the execution model of individual threads on a GPU. Threads of thread blocks are combined into warps or wavefronts and then executed in parallel on SIMD execution units on the GPU.

The term shader is used as a synonym for kernel in the MI200 ISA reference guide [7, p. 7].

NVIDIA Terminology	**AMD Terminology**	**Description**
Thread	Thread, Work Item	Single execution entity with individual execution flow.
Warp	Wavefront	32 (NVIDIA) or 64 (AMD) threads executed in lockstep.
Thread Block	Thread Block, Workgroup	The organizational unit of a thread. Threads in a thread block can communicate through local (NVIDIA) or private (AMD) memory.
Grid	Grid	The organizational unit of a thread blocks. Communication between threads only through global memory.
Kernel	Kernel, Shader	Functions executed on the GPU.

Table 2.1: Comparison of NVIDIA and AMD terminology, table taken from [8, p. 22].

2.2 AMD CDNA2 Architecture

This section provides an introduction to the AMD CDNA2 architecture. A comparison between the NVIDA Ampere and AMD CDNA2 architectures is given in Subsection 2.2.2. In Subsection 2.2.3 particular attention is paid to the memory and cache hierarchy of the AMD CDNA2 architecture. Compute relevant components are mentioned to provide a coherent architecture representation, but these are not explained in detail.

At the time of writing, CDNA2 is not the latest AMD accelerator architecture. The considerations for covering the CDNA2 architecture instead of the newer AMD CDNA3 architecture,

in order from most compelling to least compelling, are as follows:

1. **Accelerator availability**: The accelerator available for this work is the AMD MI210 based on the CDNA2 architecture.

2. **Architecture relevance**: At the time of writing, the CDNA2 architecture is only one generation behind the latest release, making it still relevant to current research and applications in the field.

3. **Information availability**: CDNA2 based accelerators have been on the market for longer and are therefore more extensively documented and analysed than the newer CDNA3 accelerators. As a result, there is a greater amount of publicly available information on the CDNA2 architecture compared to the newer CDNA3 architecture.

2.2.1 Architecture History

CDNA2 compute units take an evolutionary approach, building on the foundation established by previous generations. The original AMD CDNA architecture was derived from the earlier AMD GCN architecture but included matrix cores to support new matrix data types. CDNA2 continues this approach with minor enhancements to the CDNA compute units. [9, p. 10]

Both AMD and NVIDIA are constantly evolving their architectures in iterative steps. For this reason, information about older generations of architectures often applies to the newer generations as well. If the information on the architectures under consideration is too thin, information that does not explicitly apply to them, but to their predecessor architectures, is used. In such a case, the architecture generation to which the information explicitly applies is indicated with the source.

2.2.2 Architecture Overview

The AMD CDNA 2 architecture is the foundation for the Graphics Compute Die (GCD) used in the MI200 Series of accelerators. This series includes the MI210, MI250, and MI250X models. While the MI210 has a single GCD, the MI250 and MI250X integrate two GCDs in a single package. These dual-GCD accelerators are interconnected using AMD Infinity Fabric. From a software perspective, an MI250 or MI250X accelerator is recognized as two separate GPUs. This means that applications must be designed with multi-GPU awareness to fully utilize the available hardware resources. Each GCD is equipped with two Video Codec Next (VCN) units that provide hardware acceleration for encoding and decoding multiple video formats. The memory subsystem features High Bandwidth Memory (HBM2e), with each GCD providing 64GB of capacity managed by dedicated memory controllers. In multi-chip configurations such as the MI250 and MI250X, this results in a total of 128GB of accessible memory, while the MI210 offers 64GB with its single GCD. The HBM2e is referred to as Memory Phy in Figure 2.1. Each GCD contains an L2 cache. This cache is shared by all resources within the

GCD, ensuring efficient data access and reducing memory bottlenecks. The command processor within each GCD is responsible for receiving API-level commands from the host CPU and translating them into tasks that can be distributed to different parts of the architecture. The command processor organizes the work into groups of threads (workgroups) that are then executed as wavefronts, sets of 64 threads processed on a single instruction multiple data (SIMD) unit within a compute unit (CU). All wavefronts belonging to a single workgroup are scheduled on the same CU. Each CU can manage multiple workgroups simultaneously, up to a total of 32 wavefronts (with a maximum of 8 per SIMD). Task distribution from the command processor is managed by four Asynchronous Compute Engines (ACEs) that send kernel wavefronts to the CUs. The command processor uses hardware scheduling (HWS) to distribute tasks among the ACEs. Each ACE is responsible for dispatching tasks to a specific Shader Engine (SE). Each SE consists of multiple CUs and a workload manager that allocates tasks to individual CUs within the SE. These workload managers have four dedicated slots, one per ACE, for staging incoming work. This structure ensures that the activity of one ACE does not block others from accessing compute resources. Ideally, work is assigned to CUs in a round-robin fashion, but when resource availability is limited, smaller tasks can be prioritized to optimize execution efficiency. [9, p. 4, 5], [10, sec. AMD Instinct MI250 Microarchitecture], [11, p. 7], [12, sec. Command Processor (CP)], [13]

Each CU is equipped with dedicated pipelines optimized for scalar, vector, and matrix instructions, allowing it to process a diverse range of operations, including floating-point, integer, and matrix-based calculations. At the core of each CU is a scheduler that can manage up to 32 active wavefronts simultaneously. The scheduler issues instructions on a round-robin basis, selecting wavefronts from one of four SIMD units per cycle. While each wavefront can execute only one instruction at a time, the CU can issue up to five instructions per cycle in different execution categories, including vector ALU (VALU), scalar ALU (SALU), vector memory (VMEM), local data share (LDS), and control operations such as branches. The four SIMD vector execution units within each CU consist of 16 processing lanes for a total of 64 shader cores. These units support a wide range of operations, including logical, integer, and floating-point arithmetic at multiple precision levels (FP16, FP32, FP64), as well as Fused Multiply-Add (FMA) operations. Vector instructions operate on wavefronts containing 64 threads, with execution taking place over four cycles for double precision (FP64) operations, as each cycle processes 16 threads. In addition to vector processing, each CU contains four matrix core units optimized for matrix operations. Context switching between active wavefronts causes no overhead as their execution context remains stored within the CU. Importantly, all wavefronts within a single workgroup must be scheduled on the same CU to ensure data locality and minimize synchronization overhead. However, they can be distributed across different SIMD units within that CU to optimize execution. [9, p. 10], [11, p. 9, 10], [14, sec. Hardware Implementation], [10, sec. AMD Instinct MI250 Microarchitecture]

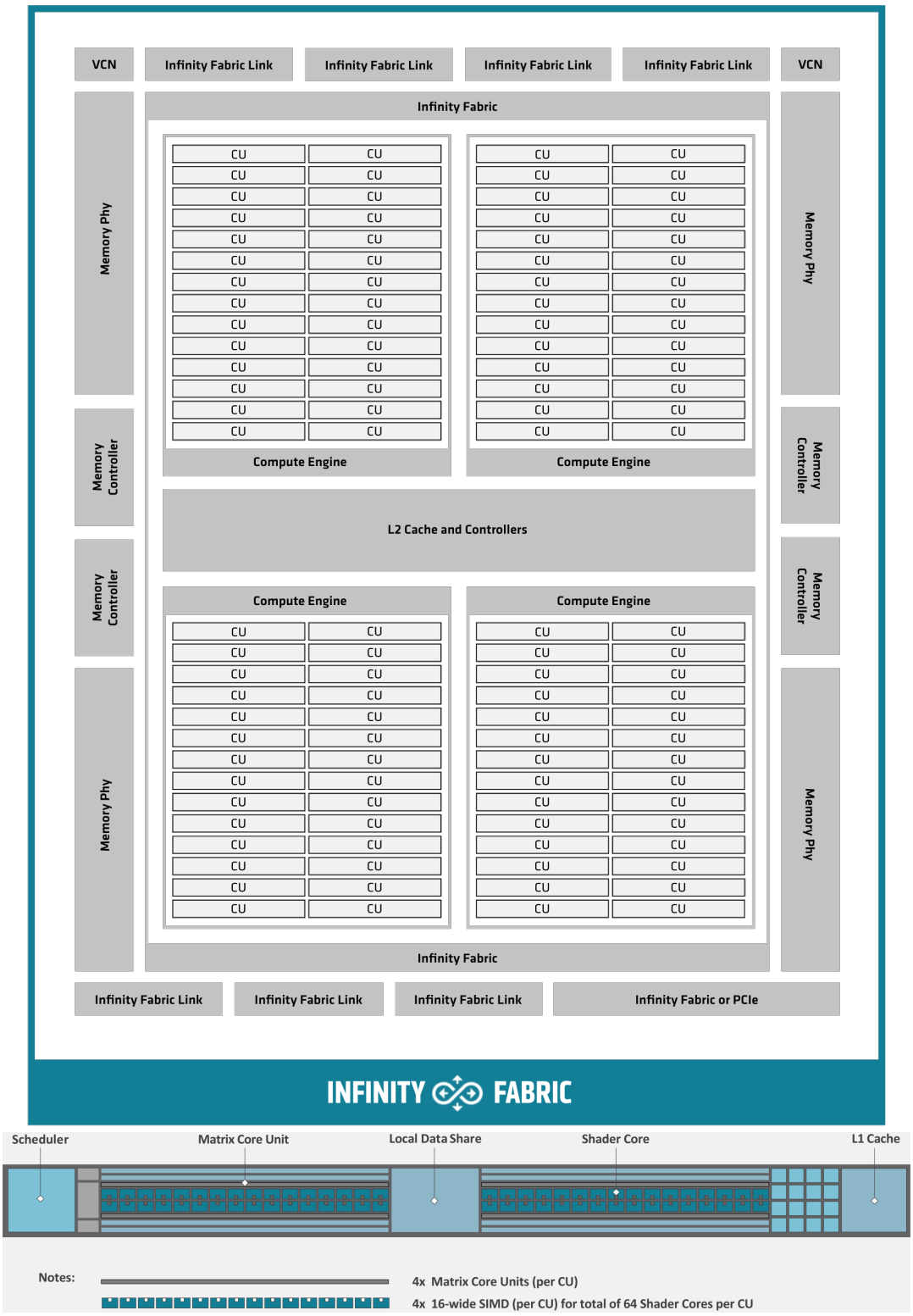

Figure 2.1: Figure showing a single AMD CDNA2 GCD at the top and a single AMD CDNA2 CU at the bottom, figures taken from [9, p. 3, 10]

2.2.3 Memory and Cache Hierarchy

This section provides information about the memory and cache hierarchy used by the AMD CDNA2 architecture. Most other architectural information such as detailed information about the AMD compute components (e.g. SALU, VALU, MFMA unit, etc.) is omitted as it is not the focus of GPUscout's optimization advice. Unlike Subsection 2.2.2, a bottom-up approach is used to present the components in the memory and cache hierarchy.

Compute Units (CU) and L1 Caches

Figure 2.2 shows two adjacent CUs, their directly attached caches, and some of their internal components. A single CU consists of several independent execution pipelines and functional units. Note, however, that Figure 2.2 is not accurate; for example, it lacks detailed information about the connections between the components shown. Nevertheless, it provides a useful overview for understanding the components being explained.

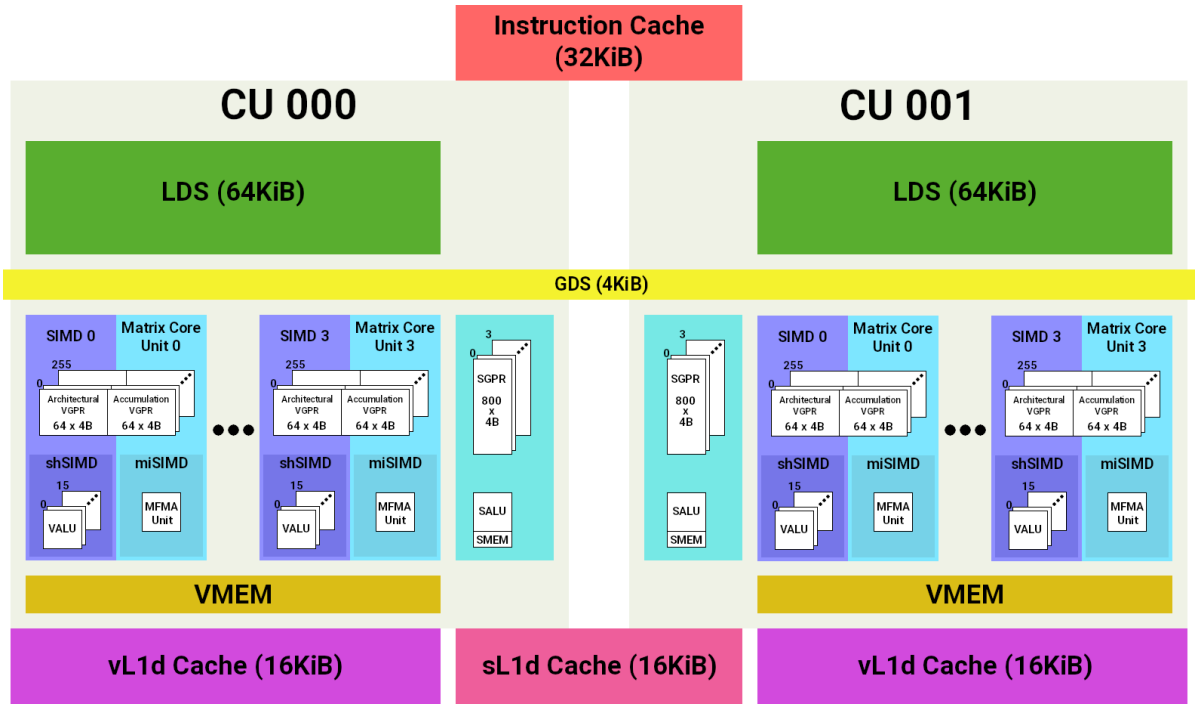

Figure 2.2: Figure showing two adjacent AMD CDNA2 CUs. The Figure is a mixture of Figure 2 in [7, p. 8], [11, p. 12] and Figure 1 in [15].

Scalar General Purpose Register (SGPR)

SGPRs are used to store data that is known at compile time to be uniform across an entire wavefront. Each CU has one SGPR file with a total size of 12.5 KiB, derived from the configuration of 4 SIMD units per CU, where each SIMD unit contains 800 SGPRs, and each

register occupies 4 bytes. This means that each SGPR is 4 bytes in size, although they can be combined to handle larger data types. For example, a 64-bit double precision floating point value or pointer would be stored in two consecutive SGPRs. [16, sec. Registers and Occupancy], [11, p. 8], [17, sec. Types of Instructions and Registers]

A wavefront can be allocated 16 to 112 SGPRs, in units of 16 SGPRs. However, not all of the allocated SGPRs can be used for general purpose use because the highest numbered allocated SGPRs are occupied for fixed functionality such as the Vector Condition Code (VCC) register (which occupies two SGPRs). Therefore, if a wavefront has a maximum of 112 SGPRs allocated, only 102 SGPRs can be used for general purpose. On the other hand, if a wavefront has the minimum number of SGPRs allocated (16 SGPRs), then typically only 2 of these will be reserved (VCC register), leaving 14 SGPRs for general purpose use. [7, p. 15][11, p. 19]

Scalar Arithmetic Logic Unit (SALU) and Scalar Memory (SMEM) Unit

The Scalar Arithmetic Logic Unit (SALU) is shared by all threads within a wavefront and is responsible for executing instructions that remain consistent across the wavefront at compile time. It operates exclusively on SGPRs because scalar instructions can only process data stored in these registers. The SALU is limited to certain operations, such as integer and logical computations. To handle memory interactions, the SALU uses a SMEM unit that facilitates data transfers between SGPRs and memory. Data retrieved through the SMEM unit can be cached in the sL1d cache and is primarily used for read-only, uniform accesses, such as kernel arguments or memory declared as `__constant__` in HIP. [12, sec. Compute Unit (CU), Pipeline Descriptions], [17, sec. Registers and Occupancy, Processor Subunits]

SMEM instructions facilitate data transfer between memory and SGPRs via the sL1d cache. These instructions allow data to be loaded from memory into SGPRs or data to be stored from SGPRs back into memory. They support reading between 1 and 16 double words (dwords), where each dword in the MI200 ISA is 32 bits in size, and writing between 1 and 4 dwords per operation. Data is transferred directly to SGPRs without format conversion. [7, p. 50], [17, sec. Computer Architecture Tidbits]

Scalar Level 1 Data (sL1d) Cache

The SMEM unit is connected to the read/write sL1d cache, which is directly attached rather than integrated into the CU. The sL1d cache is shared across two CUs, providing 16 KiB for two CUs. [11, p. 8], [10, Accelerator and GPU Hardware Specifications]

Vector General Purpose Register (VGPR)

VGPRs store data that varies across the wavefront, meaning that each thread within the wavefront can hold different values. Each CU contains a 512 KiB VGPR file consisting of 256 architectural VGPRs per SIMD across 4 SIMDs, and 256 accumulation VGPRs per matrix core unit across 4 matrix cores. Each VGPR contains 64 entries, with each entry occupying 4 bytes. Within a single wave, up to 256 KiB of architectural VGPRs and an additional 256 KiB of accumulation VGPRs can be dynamically allocated. Similar to SGPRs, each VGPR is 32 bits in size, but can be combined to store larger data types as needed. [16, sec. Registers and Occupancy], [12, sec. Pipeline Descriptions], [11, p. 9], [17, sec. Types of Instructions and Registers]

1. Architectural VGPR

Architectural VGPRs serve as the primary general purpose registers within a CU and are directly controlled by the VALU. Standard VALU instructions operate only on these registers. Each SIMD has access to 256 architectural VGPRs, which are used for a wide range of computations. [16, sec. Registers and Occupancy], [12, sec. Pipeline Descriptions], [11, p. 9]

2. Accumulation VGPR

Accumulation VGPRs are specialized registers designed for Matrix Fused Multiply-Add (MFMA) operations, providing optimized storage for intermediate results. They extend the standard architectural VGPR limit by adding 256 additional registers specifically for accumulation. Dedicated instructions such as `v_accvgpr_*` can be used to transfer data between architectural and accumulation VGPRs. These instructions also help compilers handle register spilling more efficiently by using the fast accumulation VGPRs as temporary storage. [12, Pipeline Descriptions]

Vector Arithmetic Logic Unit (VALU) and Vector Memory (VMEM) Unit

The VALU handles most computational tasks, including floating-point and integer operations. It processes vector instructions across an entire wavefront, with each thread (or vector lane within a SIMD) potentially working on different data. While vector instructions operate primarily on data stored in VGPRs, they can also access values from SGPRs. Memory operations from all four SIMD units are managed by the VMEM unit, which facilitates data transfers between VGPRs and memory. Each thread provides its own memory address and either retrieves or stores unique data. The VMEM unit is capable of handling uncoalesced memory accesses and supports the EXEC mask to control execution at the thread level. [12, sec. Compute Unit (CU), Pipeline Descriptions], [17, Types of Instructions and Registers, Processor Subunits], [11, p. 11, 12]

Vector Level 1 Data (vL1d) Cache

The VMEM unit interfaces with the read/write vL1d cache, which is attached externally rather than embedded in the CU. Each CU has a dedicated 16KiB vL1d cache with a 64-byte cache line size, while the L2 cache uses 128-byte lines. The vL1d cache operates as a write-through cache, ensuring that written data is immediately propagated to the next level of memory. [11, p. 11]

Coherency between vL1d caches across multiple CUs is managed by explicit software instructions, ensuring data consistency when needed. [12, sec. Compute Unit (CU)].

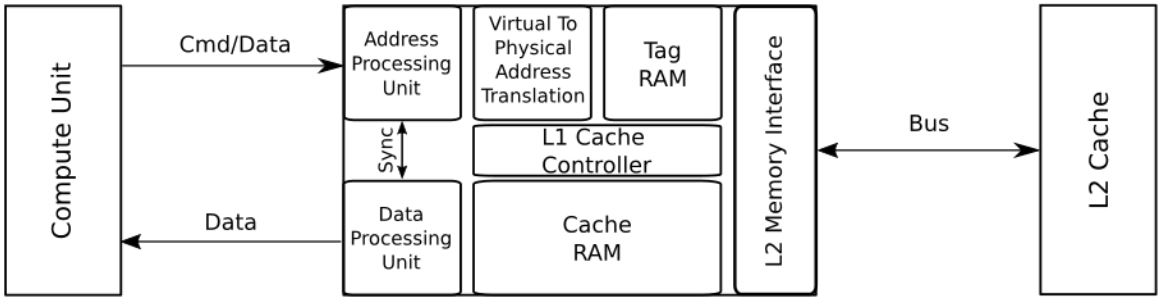

Figure 2.3: Figure that shows the structure of the vL1d cache, the figure is taken from [12, Vector L1 Cache (vL1d)].

A more detailed overview of the vL1d cache is provided in Figure 2.3, as some components are mentioned in Chapter 3. The process begins with the address processing unit, which receives instructions from the CUs and consolidates them into fewer requests for efficient cache access. These requests then pass through the address translation unit, where virtual addresses are converted to physical addresses for cache lookups. Next, the Tag RAM determines if the requested cache line already exists. The result is passed to the L1 cache controller, which directs the request accordingly, either to the L2 memory interface if the data is missing, or to the cache RAM if it is a hit. Cache RAM stores frequently accessed data for fast retrieval when needed. When data is fetched from the L2 cache, it is first placed in the cache RAM before being sent back to the CUs. Finally, the data processing unit manages the return of requested data to the appropriate compute unit. [12, Vector L1 Cache (vL1d)]

The AMDGPU backend guide translates DS in the acronyms LDS, GDS to data store, while the MI200 ISA reference guide translates DS to data share [18, sec. Address Spaces], [7, p. 8].

Global Data Store/Share (GDS) or Global Wave Store/Share (GWS)

AMD CDNA 2 GPUs include a global synchronization unit designed to coordinate workgroups across all CUs. This unit, known as the GDS, functions similarly to the LDS, but is

accessible by all CUs and serves as a dedicated global synchronization point for wavefronts. [7, p. 9], [19, p. 39][1].

Local Data Store/Share (LDS)

The LDS is a high-speed, on-chip scratchpad memory within each CU that can be explicitly managed by software to facilitate data sharing and synchronization among wavefronts in a workgroup. It has a capacity of 64 KiB and consists of 32 banks with built-in conflict resolution. The LDS supports a range of hardware atomic operations for integer, logical, and floating-point calculations. Within a CU, SIMDs are connected to the LDS in pairs, with only one SIMD in each pair able to issue an LDS instruction at a time, although both pairs can issue simultaneously. [12, sec. Local Data Share (LDS)], [11, p. 12]

Instruction Cache

Each pair of CUs shares a 32KiB instruction cache that is supported by the L2 cache. [10, sec. Accelerator and GPU Hardware Specifications], [7, p. 4].

L2 Cache, Infinity Fabric and Memory

Figure 2.4 is an extension of Figure 2.2 and shows the memory and cache hierarchy of a single AMD CDNA2 GCD. It excludes memory and cache further up the hierarchy (e.g. adjacent GPU memory, CPU RAM, storage (SSD, HDD) or remote storage, etc.). This narrow view of the cache and memory hierarchy of an AMD CDNA2 GPU is justified because GPUscout only suggests optimizations that affect performance caused by components in this narrow view.

Level 2 (L2) Cache

Each GCD has an L2 cache that is physically partitioned into 32 slices, with each slice corresponding to a memory controller. This cache is shared among all resources within a single GCD and follows a 16-way set associative design, providing a total capacity of 8MiB per GCD. Each L2 slice provides a bandwidth of 128 bytes per clock cycle, while the connection between the memory controllers and the L2 cache operates over a 64-byte wide interface. The L2 cache is organized into 32 independent channels using 256-byte address interleaving. To efficiently distribute workloads, a hashing mechanism maps incoming requests to specific L2 channels, ensuring balanced access across the cache. Requests that do not find a match in the L2 cache are forwarded through the Infinity Fabric to the appropriate memory location. [12, sec. L2 Cache (TCC)], [9, p. 5]

[1] This source mentions that GDS acts as an explicit global synchronization point between wavefronts for the AMD GCN architecture.

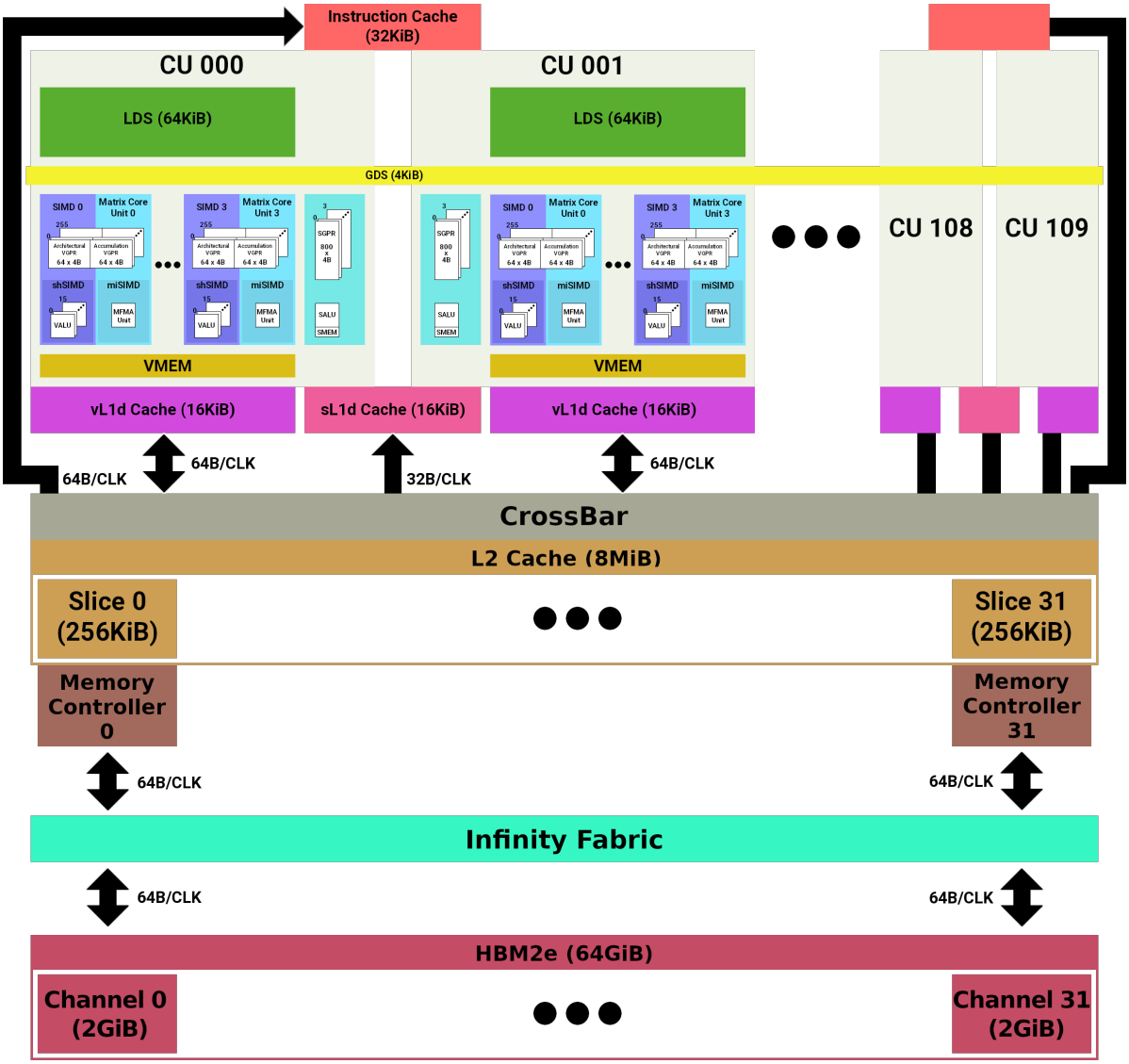

Figure 2.4: Figure shows the memory and cache hierarchy of a single AMD CDNA2 GCD.

Inifity Fabric

The Infinity Fabric facilitates connectivity between the GCD and other system components, such as memory, GPU peer-to-peer communication or I/O [10, AMD Instinct MI250 Microarchitecture], [12, L2 Cache (TCC)].

High Bandwidth Memory (HBM) 2e

Each GCD is equipped with 64GB of HBM2e memory organized into four stacks. The HBM2e memory operates at 3.2Gbit/s and is divided into 32 channels, each providing a bandwidth of 64 bytes per clock cycle. This configuration enables a peak memory bandwidth of 1.6TB/s per GCD. [9, p. 5], [10, Accelerator and GPU Hardware Specifications], [10, AMD Instinct MI250 Microarchitecture], [10, AMD Instinct MI100 Microarchitecture][2]

2.3 AMDGCN

AMD calls its instruction set architecture AMDGCN (AMD Graphics Core Next) [17, sec. Terminology] whereas NVIDIA calls it SASS. NVIDIA SASS instructions are not fully documented publicly by NVIDIA, but NVIDIA provides full documentation for an intermediate representation between source code and SASS called PTX.

2.4 AMDGPU Address Spaces

AMDGPU is an umbrella term for several AMD GPU architectures such as GCN4, CDNA2 or RDNA2 used by the AMDGPU backend guide [18, sec. Introduction].

In the address spaces section, the AMDGPU backend guide relates address space names to hardware names, among other things [18, sec. Address Spaces]. The NVIDIA PTX documentation refers to memory areas as state spaces [20, p. 31], so this name is also used in Table 2.2. Despite the different memory models, Table 2.2 tries to relate the names used by the three into relation.

Generic is not a state space according to the NVIDIA PTX documentation, but is referred to as an address space, so a comparison with the AMD generic address space seems appropriate [20, p. 87]. The remaining mapping of NVIDIA state spaces to AMD address spaces is based on page 23 of the Bauman et al. presentation [8, p. 23]. The NVIDIA PTX documentation [20, p. 31] describes additional state spaces such as param and tex, but there are no counterparts in the AMDGPU backend guide, so they are not listed in Table 2.2.

For more information on NVIDIA state spaces, see Sen's master's thesis [1, p. 18-21].

In Chapter 3, the AMD addresses space and hardware names are used synonymously.

[2]This source mentions that HBM2 is organized in four stacks in the AMD CDNA architecture.

2.5 MI200 ISA Vector Memory Operations

The following operations load/read or store/write a piece of data to or from VGPRs, separately for each thread in a wavefront. They differ in the type of memory they address and therefore in the address space they use (see Table 2.2). In contrast, scalar memory operations transfer a single piece of data shared across all threads in the wavefront.

Vector Memory (VM) Buffer (MUBUF, MTBUF) and VM Image (MIMG) Operations

MUBUF, untyped buffer operation (raw memory access without data format interpretation), MTBUF, typed buffer operation (accesses data with a defined format, such as 32-bit floats), and MIMG, image operations, all use a buffer (MUBUF, MTBUF) or image (MIMG) resource constant, a 128-bit or 256-bit value stored in SGPRs, to define the memory address, data format, stride, and other properties of the buffer or image. All VM operations are processed by the cache system (L1 and L2) and the resource constant is sent to it during operation execution. VM buffer loads have the option of returning data to the VGPRs or directly to the LDS. [7, p. 6, 55, 56, 68]

Data Share (DS) Operations

Each thread supplies a unique address via its VGPRs in a DS operation and receives unique data back from LDS to it's VGPRs. It is also possible for the LDS to provide a single dword value to all threads in a wavefront in a DS ALU operation, which is used as the `SRC0` input to the operation. [7, p. 88]

FLAT Operations

FLAT memory operations (including both GLOBAL and SCRATCH) allow the kernel to read, write, or perform atomic operations using a single flat address, without the need for a resource constant. This address can dynamically resolve to one of three memory types: global memory, scratch memory, and LDS. [10, sec. MI300 and MI200 Series Performance Counters and Metrics], [7, p. 80]

NVIDIA State Space Name	AMD Address Space Name	AMD Hardware Name
Generic *address space*	**Generic** The generic address space uses hardware-supported flat addressing to manage two specific virtual address spaces: the private aperture and the local aperture. These apertures are outside the addressable range of the global memory. [18, sec. Address Spaces]	Flat
Global	**Global** The global and constant memory areas use the same addressing system as the CPU. However, not all memory locations are shared: some are CPU only, some are GPU only, and some can be accessed by both. [18, sec. Address Spaces]	Global
	Region The region address space utilizes the hardware GDS and provides a shared memory space accessible to all wavefronts running on the same device. Compared to global memory, the region address space offers higher performance. [18, sec. Address Spaces]	GDS Global Data Store Global Data Share
Shared	**Local** The local address space is based on the hardware LDS, providing high performance memory shared between wavefronts within a workgroup. [18, sec. Address Spaces]	LDS Local Data Store Local Data Share
Constant	**Constant** *same as global*	 *same as global*
Local	**Private** The private address space uses hardware scratch memory. Each thread accesses unique memory for a given private address, even within the same or different wavefronts. [18, sec. Address Spaces]	Scratch

Table 2.2: Table that lists the AMDGPU address spaces and compares them to the NIVIDA state spaces.

3 Comparing NVIDIA and AMD GPUscout

GPUscout is a tool to analyze memory usage related bottlenecks in programs running on GPUs. GPUscout focuses on the memory hierarchy provided by a single GPU, meaning that performance-related memory operations, e.g. between two GPUs or between GPU and CPU, are not taken into account.

GPUscout looks primarily at the assembly code of the program being analyzed for memory bottlenecks. If performance-impairing patterns (in most cases performance-impairing instructions) are found in the assembly code, they are supported with PC stalling information, sometimes live register information, and general performance metrics for program execution. As there is often a single instruction behind a performance-impairing pattern, the PC stalling and live register information can be accurately mapped to these problematic instructions. The performance metrics, on the other hand, provide information about the overall execution time of the program on the GPU. Figure 3.1 shows the fundamental procedure of GPUscout.

Since the live register information is not fundamental to the concept of GPUscout, but only complementary, it is not shown in Figure 3.1 and Figure 3.3.

PC Stalling

Program Counter (PC) stalling describes a state in which the program counter temporarily does not send any further instructions to the execution units of a GPU. The reasons for this can vary, e.g. the next instruction may not yet have been loaded from memory, or one of the execution units is still busy and therefore cannot accept a new instruction. In particular, reasons concerning the GPU's memory system are of course of interest to GPUscout.

Live Registers

The number of registers used by a program during execution changes as the program runs. The registers in use at any given time during program execution are the live registers of the program at that time.

Performance Metrics

Performance metrics that are collected in the course of a program execution on a GPU can be, for example, the number of bytes that were loaded from memory or the number of clock cycles that were used for calculations. Of course, the metrics that are of particular interest to GPUscout are those relating to the GPU's memory system.

GPUscout identifies nine different potentially performance-impairing assembly code patterns. For each of these patterns, corresponding PC stalling reasons, sometimes live register

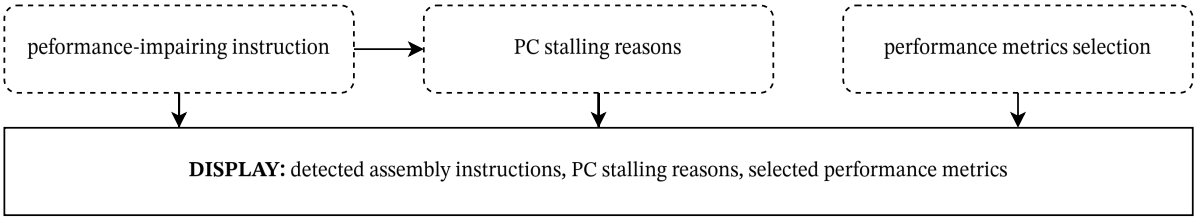

Figure 3.1: Fundamental procedure of GPUscout.

information, and appropriate performance metrics are displayed. Assembly code patterns, corresponding PC stalling reasons, live register information, and appropriate performance metrics are summarized under the term GPUscout analysis. Figure 3.2 shows the basic program flow of GPUscout schematically.

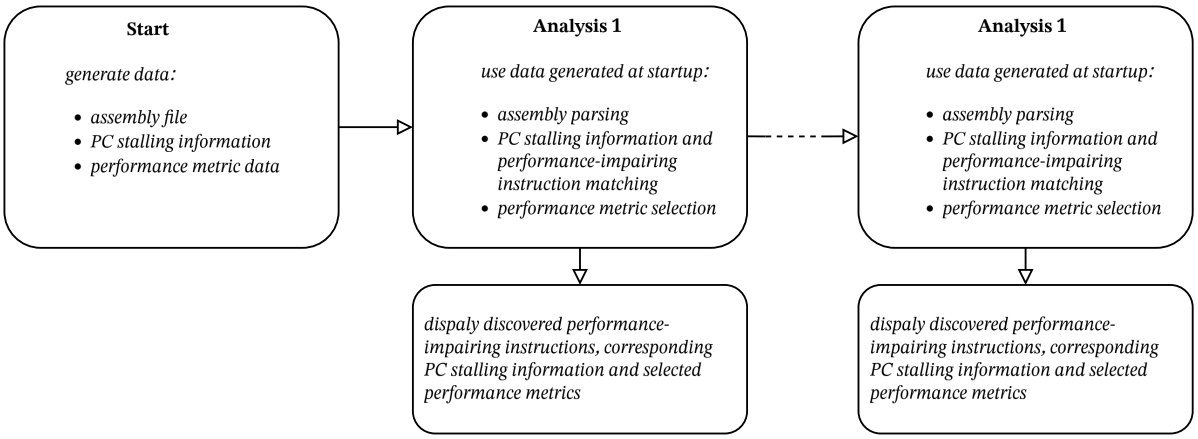

Figure 3.2: Basic program flow of the GPUscout tool.

The goal of this thesis is to extend GPUscout, which was originally designed to analyze NVIDIA-specific programs, to also analyze programs designed for AMD GPUs.

This goal is only partially achieved, as both the generation of PC stalling information for AMD GPUs and the generation of live register information to supplement some of the analyses are not currently supported by AMD GPUscout. Unlike NVIDIA GPUs, the collection of PC stalling information, called PC sampling, for AMD GPUs is still in its infancy at the time of writing [21]. Accordingly, the PC sampling utilities are still under intense development and are therefore still in beta.

On the test system (see Subsubsection 3.12) used for this work, it was not possible to generate PC stalling information using the `rocprofv3` utility, the only utility known to the author of this thesis to support PC sampling as a (beta) feature. Implementing the required functionality independently of `rocprofv3` using the ROCprofiler-SDK (the `rocprofv3` utility is based on the ROCprofiler-SDK) was not possible due to time constraints. In addition, the demo implementation of PC sampling provided by the ROCprofiler-SDK did not produce

any PC stalling reasons for the benchmarked programs on the test system.

The RGA tool for obtaining live register information could not be made to run on the test system because the available `glibc` library was too outdated for the utility.

The majority of this chapter discusses the nine analyses that GPUscout performs, with a focus on how they have been transferred to the AMD environment. At the end of the chapter, the information that NVIDIA GPUscout provides on PC stalling and live registers is presented. As this information is not available in AMD GPUscout, only a short overview is shown. In addition, selected AMD GPUscout analyses are presented with examples.

Before describing the first analysis, a brief comparison of the utilities used by NVIDIA GPUscout and AMD GPUscout is given.

3.1 Utility Program Comparison

NVIDIA GPUscout	**AMD GPUscout**
disassembling GPU binaries	
• `nvdisasm` *disassemble binary into SASS instructions* • `cuobjdump` *disassemble binary into PTX instructions*	• `roc-obj-ls` *list kernels in binary* • `roc-obj-extract` *extract kernel object out of binary* • `llvm-objdump` *disassemble kernel object into AMDGCN instructions*
collecting PC stall information	
• CUDA Profiling Tools Interface (CUPTI)	• ROCprofiler-SDK
collecting performance metrics	
• Nsight Compute (NCU)	• ROCm Compute Profiler
generating live register information	
• `nvdisasm` *executed with the -lrm=count argument*	• Radeon GPU Analyzer (RGA)

Table 3.1: Table that lists the individual utilities used in GPUscout and compares them between AMD GPUscout and NVIDIA GPUscout.

As mentioned above, PC sampling and live register information collection are not implemented in AMD GPUscout. However, the respective utilities listed in Table 3.1 as counterparts

to the NVIDIA utilities should theoretically provide PC sampling (ROCProfiler-SDK) and live register information (RGA), or at least should do so in the future.

The only NVIDIA GPUscout analysis whose parser uses the PTX code (generated by `cuobjdump`) of the program being analysed is the atomic instruction analysis (Section 3.4).

With the introduction of the utility programs used by GPUscout, the fundamental GPUscout procedure from Figure 3.1 can be supplemented with these. In addition, Figure 3.3 provides a comparison of NVIDIA and AMD GPUscout.

PC stalling information is not yet part of AMD GPUscout, so Figure 3.3 should be seen more as a glimpse into the future of AMD GPUscout, a future in which not only performance-impairing pattern detection and performance metrics collection work, but also the gathering of PC stalling and live register information.

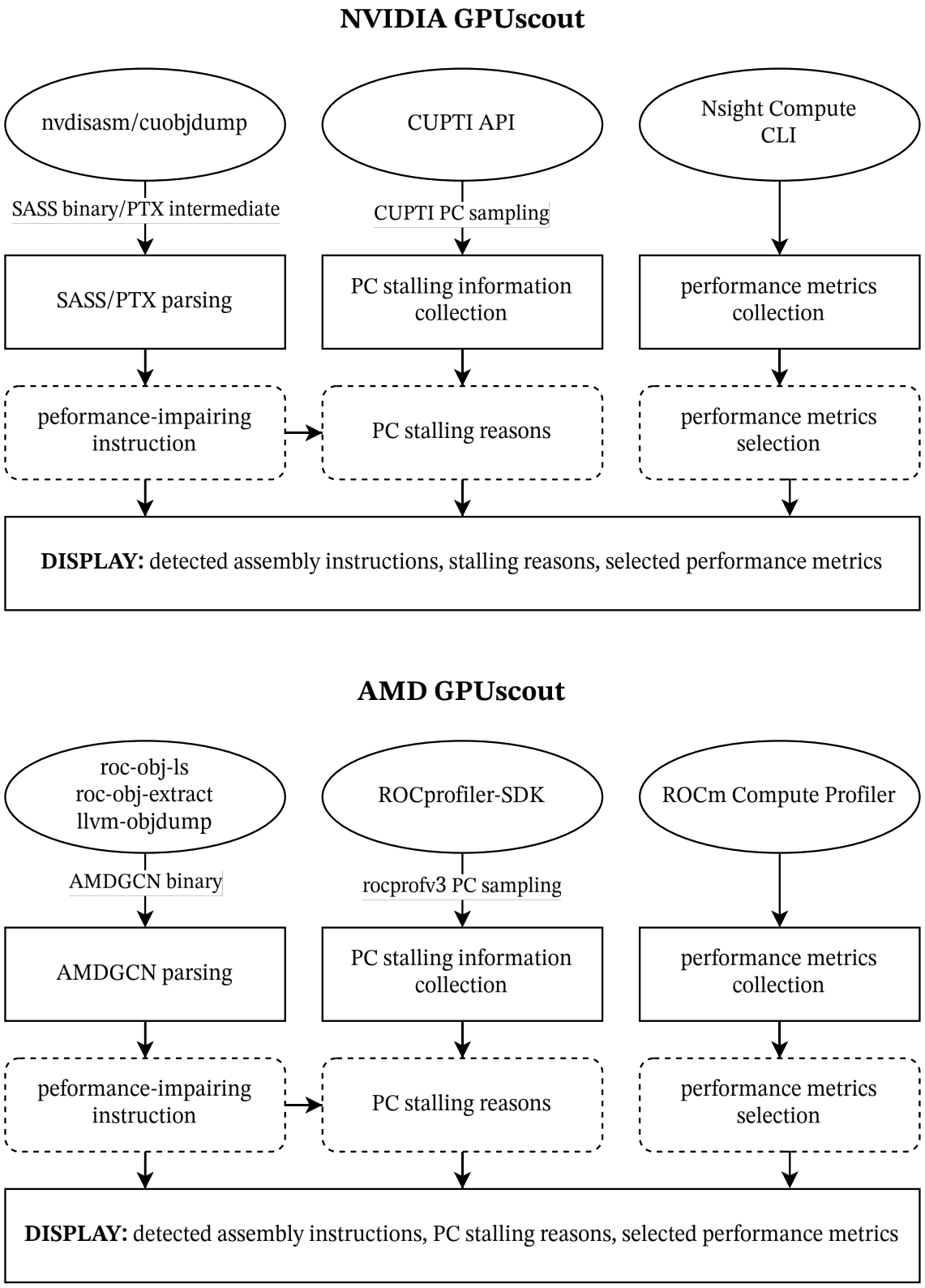

Figure 3.3: Comparison between NVIDIA and AMD GPUscout based on Figure 1 in [2].

3.2 Register Spilling Analysis

Register spilling occurs when the number of registers used by a single thread exceeds the number of registers allocated to that thread. In such a scenario, register data must be offloaded to a thread's private memory location and retrieved when the offloaded data is needed again. This process takes time, especially if the swapped data needs to go into memory because there is no space left in the L2 cache. Therefore, it is desirable to avoid register spilling to improve performance. [2]

3.2.1 Register Spilling Parser

The previously mentioned performance-impairing patterns in the assembly code are found by GPUscout in so-called parsers.

AMD GPUscout Parser Description

Identify spilled registers by monitoring stores to local memory (NVIDIA) or accumulation VGPRs, private "scratch" memory (AMD). Once a register spill is detected, capture the succeeding arithmetic instruction writing to the spilled register to determine which instruction may have triggered the spill.

Differences between the NVIDIA and AMD GPUscout register spilling parser

The NVIDIA GPUscout register spilling parser looks at both local memory load and local memory store instructions. If a local memory instruction uses an unroll, the NVIDIA GPUscout register spilling parser remembers the base address register of the instruction. If no unroll is used, the last register listed by the instruction (data register for store, base address register for load) is remembered.

The NVIDIA GPUscout register spilling parser defines register spills as patterns in which a value is first swapped out using a local store and later reloaded using a local load. AMD GPUscout simplifies the register spilling pattern in that swapping values to accumulation VGPRs or private memory is already recognized as a register spill. The AMD GPUscout register spilling parser does not take into account the reloading of the spilled values. The `vdata` register is noted for each accumulation VGPR write or private load instruction found.

The NVIDIA GPUscout register spilling parser tries to memorize the previous arithmetic instruction for each register that has been remembered due to a local memory instruction and whose destination register is identical to the remembered register.

For each accumulation VGPR write or private memory store instruction, the AMD GPUscout register spilling parser attempts to find the subsequent arithmetic instruction whose `vdst` register matches the `vdata` register of the preceding accumulation VGPR write or private memory store instruction.

AMD GPUscout Parser Output

Identify each accumulation VGPR write or global store instruction found as a register spill. Also provide the arithmetic instruction following the register spill that uses the spill register as its target.

Table 3.2 shows a comparison of SASS and AMDGCN instructions. The performance-impairing patterns searched for in NVIDIA GPUscout, which are originally based on SASS instructions, are transferred to AMDGCN instructions.

The AMDGCN instructions are provided with an asterisk at the end of the name to indicate that many different name endings can follow the specified beginning of the name. The term `scratch_store*`, for example, refers to instructions such as `scratch_store_byte` or `scratch_store_dword` etc..

Just like the AMDGCN instructions, the SASS instructions listed in Table 3.2 are also a description for a whole range of specific instructions. Since NVIDIA does not publish any official documentation on SASS instructions, it is not always clear which specific instructions are hidden behind designations such as `PRO` or `MUFU`. No AMDGCN equivalent was found for the SASS `PRO` instructions, whereas SASS `MUFU` instructions were translated to AMDGCN instructions based on Table V in the paper by Abdelkhalik et al. [22].

As described in Section 2.2, the AMD GPU architecture, unlike the NVIDIA GPU architecture, distinguishes between vector and scalar operations. It was decided to initially translate the SASS instructions exclusively into AMDGCN vector instructions when both instruction variants are given, as the vector instruction type should provide the most instructions for execution in a standard HIP kernel. For example, the set of vector instructions `v_add*` has a corresponding set of scalar instructions `s_add*`; only `v_add*` instructions are considered. But e.g. `s_cbranch*` has no equivalent set of vector instructions, so `s_cbranch*` instructions are considered by AMD GPUscout.

In Table 3.2 `flat_store*` is crossed out because flat instructions can generally be used to load/store to any address space, but it is not obvious when this is the case in the AMDGCN assembly code. Accordingly, most AMD GPUscout parsers will ignore any flat instructions.

As mentioned in Section 2.5, MIMG, MUBUF, and MTBUF memory operations require a resource constant that specifies, among other things, the memory address for the operation. This resource constant is stored in scalar registers. If the address space addressed is of interest (global or private), it is important to know which resource constant is used or, as a hint to this information, which scalar registers the resource constant is stored in. According to the AMDGPU backend guide [18, sec. Private Segment Buffer], there are two scalar register ranges that indicate that the operation is currently operating in the private address space. The first register range is `s[0:3]`, i.e. scalar registers 0 to 3, which is also supported in the presentation by Alessandro Fanfarillo and Nicholas Curtis [23, p. 6]. The second register

range differs from kernel to kernel and is located in four high numbered SGPRs. Since the second register range is not a constant range, it is difficult to detect in AMDGCN assembly code, so this second register range is ignored by all AMD GPUscout parsers.

The resource constant is specified in the `SRSRC` field of the MIMG, MUBUF and MTBUF memory instructions, i.e. for AMD GPUscout an instruction with `SRSRC == s[0:3]` is an instruction that operates in the private address space, while an instruction with `SRSRC != s[0:3]` is an instruction that operates in the global address space. The problem with this categorization is that not all global memory operations are recognized, as it may be that a memory operation accesses private address space using a resource constant stored in four high SGPR, but GPUscout recognizes this as a global memory access.

Based on the presentation by Alessandro Fanfarillo [11, p. 20], in AMD GPU architectures with accumulation VGPRs, the compiler can first write spilled registers into accumulation VGPRs before they have to be stored into private memory. Accordingly, the AMD GPUscout register spilling parser also pays attention to accumulation VGPR writes in addition to private memory stores.

For the GPUscout analyses, only the AMD GPUscout parser is described in pseudocode, but not the NVIDIA GPUscout parser. If there is a difference between the NVIDIA GPUscout parser and the AMD GPUscout parser, this difference is explicitly stated, as in the case for the register spilling parser. If there is no comment about the difference between the NVIDIA and AMD parsers, both parsers are logically identical.

Expressions such as accumulation VGPR write, private vector load, or arithmetic vector instruction in the pseudocode of parsers refer to instructions that are listed under the same name in the AMDGCN column of the previous appearing table on translating SASS or PTX to AMDGCN.

The italicized text in parser pseudocode, as in the register spilling parser, is to be understood as a comment.

(NVIDIA) SASS	(AMD) AMDGCN
local store • `STL`	accumulation VGPRs write • `v_accvgpr_write*`
	private "scratch" vector store • ~~`flat_store*`~~, `scratch_store*` • `image_store*`, `buffer_store*`, `tbuffer_store*` `SRSRC == s[0:3]`
arithmetic instructions • `ADD` • `MUL` • `MAD` • `FMA` • `MUFU` • `PRO`	arithmetic vector instructions • `ADD` `v_add*` *add* • `MUL` `v_mul*` *multiply* • `MAD` `v_mad*` *multiply add* • `FMA` `v_fma*` *fused multiply add* `v_mfma*` *packed FMA* `v_div_fma*` *FMA with fused scale* • `MUFU` `v_rcp*` *reciprocal* `v_sqrt*` *square root* `v_rsq*` *reciprocal square root* `v_sin*` *sin* `v_cos*` *cos* `v_log*` *base 2 logarithm* `v_exp*` *base 2 exponentiation* • ~~`PRO`~~

Table 3.2: Table showing the translation of SASS into AMDGCN instructions for the register spilling parser.

AMD GPUscout register spilling parser

Scan through the ADMGCN instructions until:

1. An accumulation VGPR write or private vector store instruction is encountered.

   Remember:
   - The opcode,
   - the **vdata** register,
   - the PC offset and
   - the corresponding source code location of the write/store instruction.

2. An arithmetic vector instruction is encountered.

   Check if the **vdst** register of the encountered arithmetic vector instruction was the **vdata** register of an already encountered accumulation VGPR write or private vector store instruction.

   If this is the case.

      Find the last remembered such write/store instruction and check if the found instruction already has a successor.

      If this is not the case.

         Mark the write/store instruction as having a successor.

         Remember:
         - The opcode,
         - the **vdst** register,
         - the PC offset and
         - the corresponding source code location of the arithmetic vector instruction.

vdata *in a write/store instruction := register whose data is stored in memory*
vdst *in an arithmetic instruction := register where the result of the arithmetic instruction is stored*

3.2.2 Register Spilling Performance Metrics

Since NVIDIA and AMD GPU hardware architectures are different, the performance metrics provided by both manufacturers for their GPUs are also different. Therefore, a 1-to-1 translation of the performance metrics from NVIDIA GPUscout to AMD GPUscout is not possible. For each performance-impairing assembly code pattern, the appropriate performance metrics offered by ROCm Compute Profiler are collected and displayed, but for the most part independently of NVIDIA GPUscout performance metrics. However, ROCm Compute Profiler's performance metrics are often similar to NCU's performance metrics.

For AMD GPUscout's performance metrics, the reason for their selection is provided; for NVIDIA GPUscout, this information can largely be found in Soumya Sen's master's thesis [1].

The NCU names of the performance metrics collected and displayed by NVIDIA GPUscout are shown in Table 3.3. The unit of the performance metric can be taken from the end of the name.

In the case of AMD GPUscout, the performance metric names in Table 3.4 are not alphanumerical names, but identification numbers used by the ROCm Compute Profiler utility. For this reason, the units of the performance metrics are given in brackets after the respective identification number.

Only the performance metrics that are marked with a bullet point are displayed by NVIDIA and AMD GPUscout. For example, in Table 3.3 only three performance metrics (long scoreboard, local/global throttle and percentage of L2 requests) are shown, but not the auxiliary values that are required to calculate any combined performance metric.

The descriptions of the performance metrics in Table 3.4 are taken verbatim from the ROCm Compute Profiler documentation [12].

NVIDIA

- The percentage of time that active warps have been stalled by due to a long scoreboard.

  `smsp__warp_issue_stalled_long_scoreboard_per_warp_active.pct`

- The percentage of time that active warps have been stalled by due to local and global memory throttling.

  `smsp__warp_issue_stalled_lg_throttle_per_warp_active.pct`

- Approximate percentage of total L2 operations due to local memory.

$$\frac{\text{Estimated number of operations to L2 due to local memory.}}{\text{Total L2 operations.}}$$

Total L2 operations.

$$\begin{aligned}&\texttt{lts\_\_t\_sectors\_op\_read.sum} + \texttt{lts\_\_t\_sectors\_op\_write.sum}\\ +\ &\texttt{lts\_\_t\_sectors\_op\_atom.sum} + \texttt{lts\_\_t\_sectors\_op\_red.sum}\end{aligned}$$

Estimated number of operations to L2 due to local memory.

$$\begin{aligned}&8 \times (\#\text{SMs} = 16) \times \text{L1 miss percent.}\\ &\times \text{ Number of local memory load/store operations.}\end{aligned}$$

L1 miss percent.

$$1 - \left(\frac{\texttt{l1tex\_\_t\_sector\_hit\_rate.pct}}{100}\right)$$

Number of local memory load/store operations.

$$\begin{aligned}&\texttt{smsp\_\_inst\_executed\_op\_local\_ld.sum}\\ +\ &\texttt{smsp\_\_inst\_executed\_op\_local\_st.sum}\end{aligned}$$

Table 3.3: NVIDIA GPUscout register spilling performance metrics.

AMD

- The total number of spill/stack memory instructions executed on all compute units on the GPU.

  `Metric_ID: 15.1.9` *(instructions per kernel)*

- The number of cycles the address processing unit spent working on spill/stack instructions.

  `Metric_ID: 15.1.13` *(cycles per kernel)*

- The number of cycles a wavefront in the kernel dispatch stalled waiting on memory of any kind.

  `Metric_ID: 7.2.4` *(cycles per kernel)*

- Approximate percentage of total L2 cache requests due to private memory.

$$\texttt{Metric\_ID: 15.1.9} \times \frac{\texttt{Metric\_ID: 16.3.5}}{\texttt{Metric\_ID: 17.3.1}}$$

  The ratio of the number of vL1D cache line requests that hit in vL1D cache over the total number of cache line requests to the vL1D Cache RAM.

  `Metric_ID: 16.3.5` *(requests per kernel)*

  The total number of incoming requests to the L2 from all clients for all request types.

  `Metric_ID: 17.3.1` *(requests per kernel)*

Interpreting performance metrics provided by AMD GPUscout

Register spilling often results in frequent private memory accesses. Therefore, the number of private memory (spill/stack) instructions seen by the vL1d address processing unit and the time spent by the address processing unit on spill/stack instructions are displayed. Both metrics should be reduced by avoiding register spills.

The approximate percentage of total L2 requests due to private memory should be reduced. However, since the accumulation VGPR registers could also be used for register spills, this may not be the case.

As general information, the number of cycles a wavefront is stalled in kernel dispatch waiting for memory of any kind is given. Since, in the best case, fewer memory instructions are executed when register spilling is avoided, this metric could also decrease. However, as with the approximate percentage of total L2 requests, it is also possible that VGPR registers were used for register spilling, which would not cause this metric to decrease.

Table 3.4: AMD GPUscout register spilling performance metrics.

3.3 Wavefront Divergence Analysis

The term for a bundle of threads is not standardized, NVIDIA calls it warp, AMD wavefront. NVIDIA GPUscout therefore talks about warp divergence, whereas AMD GPUscout talks about wavefront divergence. However, as this thsis is more or almost exclusively concerned with AMD GPUscout, the term wavefront divergence is used hereafter.

Divergence within a wavefront occurs when not all threads are active when an instruction is executed, i.e., due to non-uniform control flow within a wavefront. The conditional execution of instructions can reduce efficiency because it requires both branches of the conditional to be executed with different sets of active threads. This sequential execution reduces parallelism and often leads to underutilization of hardware resources. [2]

3.3.1 Wavefront Divergence Parser

AMD GPUscout Parser Description

Identify conditional branches. Once a conditional branch is encountered, note its corresponding source code location. Then find its branch target label and note the corresponding source code location of the branch target label.

AMD GPUscout Parser Output

For each conditional branch detected, issue a warning and display the corresponding source location. Additionally, display the source code location of the branch target.

The instructions `s_cbranch_i_fork`, `s_cbranch_g_fork` and `s_cbranch_join` are crossed out in Table 3.5, because they work differently to their namesakes and do not take a label as a parameter, but an address.

(NVIDIA) SASS	(AMD) AMDGCN
conditional branch • `BRA` *a predicate check precedes this instruction*	conditional relative branch • `s_cbranch*` ~~`s_cbranch_i_fork`~~, ~~`s_cbranch_g_fork`~~, ~~`s_cbranch_join`~~

Table 3.5: Table showing the translation of SASS into AMDGCN instructions for the wavefront divergence parser.

AMD GPUscout wavefront divergence parser

Scan through the ADMGCN instructions until:

1. A conditional branch instruction is encountered.

   Remember:
   - The referenced branch label and
   - the corresponding source code location of the branch instruction.

2. A branch label is encountered.

   Remember the corresponding source code location of the branch label.

3.3.2 Wavefront Divergence Performance Metrics

NVIDIA	• The branch divergence percent.
	$100 \times \frac{\text{The average number of branch targets that diverge.}}{\text{The average number of unique branch targets assigned to the kernel.}}$
	The average number of branch targets that diverge.
	`sm__sass_branch_targets_threads_divergent.avg`
	The average number of unique branch targets assigned to the kernel.
	`sm__sass_branch_targets.avg`

AMD	• The total number of branch operations issued. `Metric_ID: 10.1.6` *(instruction per kernel)* • The percentage of the kernel's duration the branch unit was busy executing instructions. `Metric_ID: 11.2.5` *(percent)*
Interpreting performance metrics provided by AMD GPUscout The percentage of kernel time that the branch unit was busy executing instructions and the total number of branch operations issued should give an indication of how much wavefront divergence could affect the runtime.	

Table 3.6: NVIDIA and AMD GPUscout wavefront divergence performance metrics.

3.4 Atomic Instruction Analysis

Atomic operations are operations that combine data from registers and data in memory, and write the result back to memory; they also often have the option of returning the pre-operation value to the registers. Atomic operations guarantee that the operation is performed without interference from other threads, i.e. no other thread can access the memory until the operation with the current thread is complete. This property causes threads to be serialized when they perform an atomic operation on the same memory address. A high frequency of global atomic instructions relative to shared (NVIDIA) or local (AMD) atomic instructions often indicates a potential memory bottleneck. [2] [7, p. 87-89]

3.4.1 Atomic Instruction Parser

AMD GPUscout Parser Description

Count the number of local and global atomic instructions. Also check if the found atomic instructions are inside a loop.

Differences between the NVIDIA and AMD GPUscout atomic instruction parser

The NVIDIA GPUscout atomic instruction parser examines PTX code instead of SASS code. This is due to the fact that atomic instructions in PTX code are better recognized, and a better distinction can be made between global and shared atomic instructions. [1, p. 48]

The AMD GPUscout atomic instruction parser analyzes AMDGCN code.

The NVIDIA GPUscout atomic instruction parser only observes and recognizes atomic add instructions, whereas the AMD GPUscout atomic instruction parser recognizes all atomic instructions listed in the AMDGCN column of Table 3.7.

AMD GPUscout Parser Output

Display any global or local atomic instruction found, and whether the instruction could be inside a loop.

In AMDGCN, local atomic instructions are not easily distinguished from local read or write instructions, because local atomic instructions do not have the term atomic or anything similar in their name. For this reason, a subset of all local atomic instructions has been selected for the AMD GPUscout parsers to look for when searching for local atomic instructions. This selection contains the most commonly used local atomic operations and covers a large portion of all available local atomic instructions. This selection is shown in Table 3.7.

In AMDGCN global atomic instructions are clearly marked with the sub-designation atomic, which is why the problem that occurs with local atomic instructions does not occur with global atomic instructions.

(NVIDIA) PTX	(AMD) AMDGCN
branch • `bra`	unconditional and conditional relative branch • `s_branch*` • `s_cbranch*` ~~`s_cbranch_i_fork`~~, ~~`s_cbranch_g_fork`~~, ~~`s_cbranch_join`~~
atomic instructions • `atom.global.add` • `atom.shared.add`	atomic vector instructions • `atom.global` ~~`flat_atomic*`~~ *~~flat atomics~~* `global_atomic*` *global atomics* `image_atomic*` *image atomics* `buffer_atomic*` *buffer atomics* • `atom.shared` `ds_add*` *data share (d. s.) add* `ds_and*` *d. s. and* `ds_dec*` *d. s. decrement* `ds_inc*` *d. s. increment* `ds_max*` *d. s. max* `ds_min*` *d. s. min* `ds_or*` *d. s. or* `ds_rsub*` *d. s. subtraction with reversed operands* `ds_xor*` *d. s. xor*

Table 3.7: Table showing the translation of PTX into AMDGCN instructions for the atomic instruction parser.

AMD GPUscout atomic instruction parser

Scan through the ADMGCN instructions until:

1. A global atomic vector instruction is encountered.

   - Increase the global atomic vector instruction count by 1.
   - Remember the global atomic vector instruction.

2. A local atomic vector instruction is encountered.

   - Increase the local atomic vector instruction count by 1.
   - Remember the local atomic vector instruction.

3. A branch instruction is encountered.

   - Remember the branch instruction.
   - Check if the branch instruction references a branch label that has already been encountered.

   If this is the case.

   Mark the referenced branch label as defining a loop.

4. A branch label is encountered.

   Remember the branch label.

3.4.2 Atomic Instruction Performance Metrics

AMD	• The number of cycles a wavefront in the kernel dispatch stalled waiting on memory of any kind. `Metric_ID: 7.2.4` *(cycles per kernel)* • The total number of global and generic memory atomic (with and without return) instructions executed on all compute units on the GPU. `Metric_ID: 16.3.3` *(requests per kernel)* • The total number of cycles spent on LDS atomics with return. `Metric_ID: 12.2.5` *(cycles per kernel)* • The total number of atomic requests (with and without return) to the L2 from all clients. `Metric_ID: 17.3.4` *(requests per kernel)* • The percent of write requests generated by the L2 cache that are atomic requests to any memory location. `Metric_ID: 17.2.7` *(percent)* • The total number of L2 requests to Infinity Fabric to atomically update 32B or 64B of data in any memory location. `Metric_ID: 17.5.10` *(requests per kernel)*
Interpreting performance metrics provided by AMD GPUscout The first metric is again a metric to get a general picture of the kernel's memory situation. The second and third metrics are intended to estimate how much time is spent on global and local atomics. The last three metrics deal with the L2, as many global atomic instructions are processed there. The last metric is particularly interesting as it indicates how many atomic operations could not be resolved in L2 but were passed on to other locations. This is likely to have a large impact on performance, so should be of particular interest. [9, p. 5]	

Table 3.8: AMD GPUscout atomic instruction performance metrics.

NVIDIA

- The percentage of time that active warps have been stalled by due to local and global memory throttling.

  `smsp__warp_issue_stalled_lg_throttle_per_warp_active.pct`

- The percentage of time that active warps have been stalled by due to a long scoreboard.

  `smsp__warp_issue_stalled_long_scoreboard_per_warp_active.pct`

- The percentage of time that each active warp is stalled due to MIO (Memory I/O) throttling.

  `smsp__warp_issue_stalled_mio_throttle_per_warp_active.pct`

- The L1 miss percent, due to global memory atomic operations.

  $$100 - \texttt{l1tex\_\_t\_sector\_pipe\_lsu\_mem\_global\_op\_red\_hit\_rate.pct} + \texttt{l1tex\_\_t\_sector\_pipe\_lsu\_mem\_global\_op\_atom\_hit\_rate.pct}$$

- The number of global atomic operations observed by L1.

  $$32 \times \texttt{l1tex\_\_t\_sectors\_pipe\_lsu\_mem\_global\_op\_red.sum} + \texttt{l1tex\_\_t\_sectors\_pipe\_lsu\_mem\_global\_op\_atom.sum}$$

- The number of global atomic operation from L1 to L2.

  $$\text{The number of global atomic operations observed by L1.} \times \left(1 - \frac{\text{The L1 Cache miss percent, due to global memory atomic requests.}}{100}\right)$$

- The L2 miss percentage of L1 atomic operations.

  $$100 - \texttt{lts\_\_t\_sector\_op\_red\_hit\_rate.pct} + \texttt{lts\_\_t\_sector\_op\_atom\_hit\_rate.pct}$$

- The number of global atomic operations from L2 to memory.

  $$\text{The number of global atomic operation from L1 to L2.} \times \left(1 - \frac{\text{The L2 miss percentage of L1 atomic operations.}}{100}\right)$$

- The number of shared memory bytes accessed by atomic operations.

  `sm__sass_data_bytes_mem_shared_op_atom.sum`

Table 3.9: NVIDIA GPUscout atomic instruction performance metrics.

3.5 Local Memory Analysis

Shared (NVIDIA) or local (AMD) memory provides faster access times compared to global memory. However, for local memory to be effective, the data must be accessed multiple times. If not, the cost of moving data in and out of local memory may surpass the performance gains. [2]

3.5.1 Local Memory Parser

AMD GPUscout Parser Description

Identify the registers involved in global memory loads and subsequent local memory writes, and check whether they are used within a for loop. Also, count how many arithmetic and atomic instructions use such registers.

AMD GPUscout Parser Output

Consider only registers with a load count greater than zero, an atomic/arithmetic destination count greater than one, and an atomic/arithmetic destination count greater than the load count. Display the load count and the number of times an atomic or arithmetic instruction used the register as a target for these registers. Also display any global load and LDS write instructions that use them. For global load instructions that don't load directly into LDS, indicate if they are in a loop and recommend thinking about using LDS instead of global memory to reduce the impact of these instructions on the memory system.

The addition *LDS bit in bitmap* or *no LDS bit in bitmap* in Table 3.10 indicates that some instructions can write directly from memory to the LDS. These instructions are equated with the `LDGSTS` SASS instructions in the translation from SASS to AMDGCN.

The MI200 ISA reference guide contradictorily describes an LDS bit for `tbuffer_load*` instructions, but this is not to be found in their bitmap description. Additionally, `tbuffer_load*` instructions are not listed in the documentation section that lists instructions that can load directly into the LDS. [7, p. 66, 188]

It is assumed that the LDS can only be addressed by the vector units and not by the scalar units because Chapter 11 Data Share Operations of the MI200 ISA reference guide [7, p. 86] only mentions the VALUs that read or write to the LDS. Therefore, Table 3.10 does not explicitly refer to vector LDS instructions, but only to LDS instructions.

NVIDIA SASS distinguishes between two types of atomic operations, reduction operations (`RED`) and atomic operations (`ATOM`). The former do not return a value, whereas the latter do. This distinction does not exist in AMDGCN, as almost all global atomic operation can return a value if desired (`GLC` bit) and for local atomic operations a possible return value depends on the respective instruction. For this reason, no translation for SASS `RED` instructions is given in Table 3.10. [1, p. 48] [7, p. 9, 68, 70, 80, 83, 88]

(NVIDIA) SASS	(AMD) AMDGCN
branch • BRA	unconditional and conditional relative branch • s_branch* • s_cbranch* ~~s_cbranch_i_fork~~, ~~s_cbranch_g_fork~~, ~~s_cbranch_join~~
global load • LDG • LDGSTS	global vector load • ~~flat_load*~~, global_load* *LDS bit in bitmap* • buffer_load* *LDS bit in bitmap* SRSRC != s[0:3] • tbuffer_load*, image_load* *no LDS bit in bitmap* SRSRC != s[0:3]
shared store • STS	local (LDS) write • ~~flat_store*~~ • ds_write*
arithmetic instructions • ADD • MUL • MAD • FMA • MUFU	arithmetic vector instructions • ADD v_add* • MUL v_mul* • MAD v_mad* • FMA v_fma*, v_mfma*, v_div_fma* • MUFU v_rcp*, v_sqrt*, v_rsq*, v_sin*, v_cos*, v_log*, v_exp*
atomic instructions • ATOMG • ATOMS • RED	atomic vector instructions • ATOMG ~~flat_atomic*~~, global_atomic*, image_atomic*, buffer_atomic* • ATOMS ds_add*, ds_and*, ds_dec*, ds_inc*, ds_max*, ds_min*, ds_or*, ds_rsub*, ds_xor* • ~~RED~~

Table 3.10: Table showing the translation from SASS to AMDGCN instructions for the local memory parser.

AMD GPUscout local memory parser

Scan through the AMDGCN instructions until:

1. A global vector load instruction is encountered.

   - Remember the global vector load instruction and mark it, if it loads directly into LDS instead of VGPRs, as using the LDS bit.
   - Check if the **vdst** register of the instruction has already been used as the **vdst** register for another global vector load instruction or as a **vdata** register of a LDS write instruction.

   If this is the case.

   Increase the load count of the **vdst** register by 1.

   If this is not the case.

   Remember that the **vdst** register was used by a global vector load instruction.

2. A LDS write instruction is encountered.

   - Remember the LDS write instruction.
   - Check if the **vdata** register of the LDS write instruction has already been used by a global vector load instruction as **vdst** register or as a **vdata** register of a LDS write instruction.

   If this is not the case.

   - Remember that the **vdata** register was used by a LDS write instruction.

3. An arithmetic or (global | local) atomic vector instruction is encountered.

   Check if any register used by the instruction has already been used by a global vector load instruction as **vdst** register or as a **vdata** register of a LDS write instruction.

   If this is the case.

   Increase the count of arithmetic and atomic vector instructions using the register by 1.

4. A branch instruction is encountered.

   - Remember the branch instruction.
   - Check if the branch instruction references a branch label that has already been encountered.

   If this is the case.

   Mark the branch instruction as defining a loop.

5. A branch label is encountered.

   Remember the branch label.

3.5.2 Local Memory Performance Metrics

NVIDIA	• The percentage of time that active warps have been stalled by due to a long scoreboard.
	`smsp__warp_issue_stalled_long_scoreboard_per_warp_active.pct`
	• The percentage of time that each active warp is stalled due to MIO (Memory I/O) throttling.
	`smsp__warp_issue_stalled_mio_throttle_per_warp_active.pct`
	• The number of shared memory load instructions.
	`sm__sass_inst_executed_op_shared_ld.sum`
	• The average amount of data (in bytes) accessed during shared memory load operations (shared memory efficiency).
	`smsp__sass_average_data_bytes_per_wavefront_mem_shared_op_ld.pct`
	• The number of memory operations per shared memory load instruction (number of bank conflicts).
	$\left\lfloor \frac{\text{The number of shared memory load operations.}}{\text{The number of shared memory load instructions.}} \right\rfloor$
	The number of shared memory load operations.
	`l1tex__data_pipe_lsu_wavefronts_mem_shared_op_ld.sum`

Table 3.11: NVIDIA GPUscout shared memory performance metrics.

AMD	• The number of cycles a wavefront in the kernel dispatch stalled waiting on memory of any kind. `Metric_ID: 7.2.4` *(cycles per kernel)* • The percentage of the kernel's duration the LDS was actively executing instructions (including, but not limited to, load, store and atomic operations). `Metric_ID: 12.1.0` *(percent of peak)* • The percentage of SIMDs in the VALU actively issuing LDS instructions, averaged over the lifetime of the kernel. `Metric_ID: 12.1.1` *(percent of peak)* • The percentage of active LDS cycles that were spent servicing bank conflicts. `Metric_ID: 12.1.3` *(percent of peak)*
Interpreting performance metrics provided by AMD GPUscout The first metric gives a general picture of the memory situation of the kernel. The following three metrics all deal with LDS memory. The percentage of kernel time that the LDS was actively executing instructions and the percentage of SIMDs in the VALU that were actively issuing LDS instructions are displayed to show the programmer how much LDS is already being used in the kernel. If LDS is used intensively, this can lead to undesirable bank conflicts, so the percentage of active LDS cycles spent resolving bank conflicts is displayed.	

Table 3.12: AMD GPUscout local memory performance metrics.

3.6 `__restrict__` Analysis

Pointer aliasing occurs when multiple pointer variables refer to the same memory location [24, sec. Pointer Aliasing].

The HIP keyword `__restrict__` can be used to tell the compiler about non aliasing pointers, allowing the compiler to potentially optimize the code better [14, sec. HIP C++ Language Extensions].

3.6.1 `__restrict__` Parser

AMD GPUscout Parser Description

Examine all the registers into which data is loaded. Then check if these registers are a target for an arithmetic operation, i.e. if these registers are written to.

Differences between the NVIDIA and AMD GPUscout `__restrict__` parser

The NVIDIA GPUscout parser also remembers whether global load operations have loaded from a read only cache.

This is not the case for the AMD GPUscout parser, as it is assumed that only scalar operations can access a so-called constant cache, but not vector operations. The block diagram on page four of the MI200 ISA reference guide [7, p. 4, 21] connects the constant cache only to the SGPRs.

However, the HIP documentation [14, sec. HIP C++ Language Extensions] mentions the keyword `__constant__`, which can be used to tell the compiler that variables are constant. These variables can then be loaded from the constant cache, but it is not entirely clear whether they are scalar or vector variables.

AMD GPUscout Parser Output

For any register that is used exclusively as a target by global loads, but is never used as a target register by arithmetic or atomic instructions, provide the code location of the first global load instruction that uses such a register, and suggest that the programmer use the `__restrict__` keyword.

(NVIDIA) SASS	(AMD) AMDGCN
global load • LDG	global vector load • ~~flat_load*~~, global_load* • image_load*, buffer_load*, tbuffer_load* SRSRC != s[0:3]
arithmetic instructions • ADD • MUL • MAD • FMA • MUFU	arithmetic vector instructions • ADD v_add* • MUL v_mul* • MAD v_mad* • FMA v_fma*, v_mfma*, v_div_fma* • MUFU v_rcp*, v_sqrt*, v_rsq*, v_sin*, v_cos*, v_log*, v_exp*
atomic instructions • ATOMG • ATOMS • RED	atomic vector instructions • ATOMG ~~flat_atomic*~~, global_atomic*, image_atomic*, buffer_atomic* • ATOMS ds_add*, ds_and*, ds_dec*, ds_inc*, ds_max*, ds_min*, ds_or*, ds_rsub*, ds_xor* • ~~RED~~

Table 3.13: Table showing the translation from SASS to AMDGCN instructions for the __restrict__ parser.

AMD GPUscout __restrict__ parser

Scan through the ADMGCN instructions until:

1. A global vector memory load instruction is encountered.

   If the **vdst** register of the load instruction has not been encountered before, remember it as unused.

2. An arithmetic or atomic vector instruction is encountered.

   If the **vdst** register of the encountered instruction was also used as the **vdst** register for a global load instruction, mark the register as used.

3.6.2 `__restrict__` Performance Metrics

NVIDIA	• The percentage of immediate constant cache (IMC) misses. `smsp__warp_issue_stalled_imc_miss_per_warp_active.pct`

Table 3.14: NVIDIA GPUscout `__restrict__` performance metrics.

Since the AMD CDNA2 architecture does not provide a constant cache for vector operations, and it is not clear how the compiler optimizes with the `__restrict__` keyword set, AMD GPUscout therefore does not display performance metrics for the `__restirct__` analysis.

3.7 Vectorized Load Analysis

Load instructions that fetch multiple data elements are called vectorized loads in this analysis. Loading multiple data elements in one piece with a single instruction, rather than with multiple smaller load instructions, may improve memory bandwidth utilization, but it certainly reduces the number of instructions that must be executed to fetch data. Therefore, vectorized load instructions should be preferred when applicable. [2]

The term vectorised load refers to a load operation that loads multiple data elements at once. This is different from a vector instruction (which has been described several times in this chapter) in the AMD CDNA2 architecture sense, which refers to an instruction that is executed by the SIMD units in the CDNA2 architecture.

3.7.1 Vectorized Load Parser

AMD GPUscout Parser Description

Locate global load instructions and note the base address register and offset they use to access memory.

Differences between the NVIDIA and AMD GPUscout vectorized load parser

The NVIDIA GPUscout vectorized load parser considers so-called unrolls, offsets, which are added to or subtracted from the base address. The MI200 ISA reference guide does not mention unrolls, but instructions can specify an offset in several ways. The AMD GPUscout vectorized load parser considers immediate offsets and equates them to NVIDIA unrolls.

AMD GPUscout Parser Output

Display all global load instructions found. If the global load instruction only transfers a single double word, and there is more than one global load instruction using the same vaddr address with different offsets, inform that it may be beneficial to load multiple dwords at once.

In addition to `buffer_load_dword*` and `global_load_dword*` in Table 3.15, there are also the instructions `buffer_load_format*`, `tbuffer_load_format*` and `global_load_format*`, which can load several data elements. However, the data is stored as packed 16-bit values, which is why these instructions are not taken into account by the AMD GPUscout vectorized load parser. [7, p. 56]

The AMD GPUscout vectorized load parser distinguishes between the `vaddr` and `srsrc` registers. For `global_load_dword*` instructions, the `vaddr` register is remembered, while for `buffer_load_dword*` instructions, the `srsrc` registers are remembered. This distinction is made because, as explained in Subsection 3.2.1, the memory address for a buffer memory

operation is stored in a resource constant. Therefore, buffer instructions that want to load from the same memory address are likely to share the same resource constant.

(NVIDIA) SASS	(AMD) AMDGCN
global load • `LDG`	global vector load • ~~`flat_load_dword*`~~, `global_load_dword*` • `buffer_load_dword*` `SRSRC != s[0:3]`

Table 3.15: Table showing the translation from SASS to AMDGCN instructions for the vectorized load parser.

AMD GPUscout vectorized load parser

Scan through the ADMGCN instructions until:

1. A global vector load instruction is encountered.

   - Increase global vector load instruction count by 1.
   - Check if the encountered global vector load instruction has the same corresponding source code location as a previously seen global vector load instruction.

   If this is not the case:

   Remember:
   - The number of bytes loaded,
   - the PC offset,
   - the **vaddr** or **srsrc** register(s)
   - the offset (if any) and
   - the corresponding source code location of the global vector load instruction.

   If this is the case:

   Check if there is already a global vector load instruction with the same corresponding source code location that uses the same **vaddr** or **srsrc** register(s).

   If this is not the case:

   Remember:
   - The number of bytes loaded,
   - the PC offset,
   - the **vaddr** or **srsrc** register(s)
   - the offset (if any) and
   - the corresponding source code location of the global vector load instruction.

   If this is the case:

   Add the offset of the newly seen global load instruction to the number of offsets used by global load instructions for the **vaddr** or **srsrc** register(s).

   *The global load instruction may have an offset because there was already another global load instruction in the same code location using the same **vaddr** register, so without an offset the two instructions would overwrite each other.*

***vaddr** in a vector load/read instruction := the register that holds the memory address from which the data will be loaded*

3.7.2 Vectorized Load Performance Metrics

NVIDIA	• The percentage of time that active warps have been stalled by due to a long scoreboard.
	`smsp__warp_issue_stalled_long_scoreboard_per_warp_active.pct`
	• The peak occupancy of the GPU.
	`sm__warps_active.avg.pct_of_peak_sustained_active`

AMD	• The number of cycles a wavefront in the kernel dispatch stalled waiting on memory of any kind. `Metric_ID: 7.2.4` *(cycles per kernel)*
Interpreting performance metrics provided by AMD GPUscout Vectorized loads could reduce the time that wavefronts wait for memory because the memory system could be used more efficiently. Therefore, the number of cycles a wavefront has been stalled waiting for memory of any type is displayed to the programmer.	

Table 3.16: NVIDIA and AMD GPUscout vectorized load performance metrics.

3.8 Datatype Conversion Analysis

Data type conversions are computationally costly on GPUs because they raise the number of instructions and may demand extensive use of multiple GPU pipelines. As a result, conversions like F2F (floating-point to floating-point), I2F (integer to floating-point), and F2I (floating-point to integer) should be minimized whenever possible. [2]

3.8.1 Datatype Conversion Parser

AMD GPUscout Parser Description

Locate F2F, F2I and I2F datatype conversion instructions.

AMD GPUscout Parser Output

Mention any F2F, F2I, and I2F conversions found by displaying the corresponding source code location.

The AMD GPUscout datatype conversion parser ignores the VOP3 versions of the instructions listed in the AMDGCN column of Table 3.17. This is because they are not mentioned in the MI200 ISA reference guide, which only documents packed VOP3 conversion instructions.

AMD GPUscout datatype conversion parser

Scan through the ADMGCN instructions until:

1. A float to float conversion instruction is encountered.

   Increase the F2F count by 1 and remember the corresponding source code location of the instruction.

2. A float to integer conversion instruction is encountered.

   Increase the F2I count by 1 and remember the corresponding source code location of the instruction.

3. An integer to float conversion instruction is encountered.

   Increase the I2F count by 1 and remember the corresponding source code location of the instruction.

(NVIDIA) SASS	(AMD) AMDGCN
datatype conversion instructions • F2F • I2F • F2I	datatype conversion instructions • F2F `v_cvt_f16_f32*` *float 32 to float 16* `v_cvt_f32_f16*` *float 16 to float 32* `v_cvt_f32_f64*` *float 64 to float 32* `v_cvt_f64_f32*` *float 32 to float 64* • I2F `v_cvt_f16_i16*` *integer 16 to float 16* `v_cvt_f16_u16*` *unsigned integer 16 to float 16* `v_cvt_f32_i32*` *integer 32 to float 32* `v_cvt_f32_u32*` *unsigned integer 32 to float 32* `v_cvt_f64_i32*` *integer 32 to float 64* `v_cvt_f64_u32*` *unsigned integer 32 to float 64* • F2I `v_cvt_flr_i32_f32*` $\lfloor$*float 32*$\rfloor$ *to integer 32* `v_cvt_i16_f16*` *float 16 to integer 16* `v_cvt_i32_f32*` *float 32 to integer 32* `v_cvt_i32_f64*` *float 64 to integer 32* `v_cvt_u16_f16*` *float 16 to unsigned integer 16* `v_cvt_u32_f32*` *float 32 to unsigned integer 32* `v_cvt_u32_f64*` *float 64 to unsigned integer 32*

Table 3.17: Table showing the translation of SASS into AMDGCN instructions for the datatype conversion parser.

3.8.2 Datatype Conversion Performance Metrics

NVIDIA	• The percentage of time that each active warp is stalled due to MIO (Memory I/O) throttling.
	`smsp__warp_issue_stalled_mio_throttle_per_warp_active.pct`
	• The percentage of time that each active warp is stalled due to the texture unit.
	`smsp__warp_issue_stalled_tex_throttle_per_warp_active.pct`
	• The percentage of time that active warps have been stalled by due to a short scoreboard.
	`smsp__warp_issue_stalled_short_scoreboard_per_warp_active.pct`

AMD	• The total number of type conversion instructions (such as converting data to or from F32 to F64) issued to the VALU. `Metric_ID: 10.2.14` *(instructions per kernel)*
Interpreting performance metrics provided by AMD GPUscout The total number of type conversion instructions issued to the VALU is displayed in order to estimate how large the performance losses could be with regard to data type conversions.	

Table 3.18: NVIDIA and AMD GPUscout datatype conversion performance metrics.

3.9 Deadlock Detection Analysis

Deadlocks can potentially occur when threads try to synchronize after branching from a conditional while holding a mutex. When a deadlock occurs, the kernel cannot complete its computation, which is definitely not desired. [1, p. 57]

3.9.1 Deadlock Detection Parser

AMD GPUscout Parser Description

Locate situations where a synchronization instruction is encountered after a conditional branch instruction is encountered after a mutex is acquired.

AMD GPUscout Parser Output

If a potential deadlock situation is detected, this is displayed.

Deadlock detection is only concerned with the fact that an atomic instruction is executed, but not in which memory area this takes place. Accordingly, the AMD GPUscout deadlock detection parser also searches for `flat_atomic*` instructions, as shown in Table 3.19.

The asterisk in the Table 3.19 in `@P* BRA` is interpreted the same way as the asterisk in the AMDGCN instructions. Accordingly, the asterisk represents any character string.

(NVIDIA) SASS	(AMD) AMDGCN
conditional branch • `@P* BRA`	conditional relative branch • `s_cbranch*` ~~`s_cbranch_i_fork`~~, ~~`s_cbranch_g_fork`~~, ~~`s_cbranch_join`~~
synchronization barrier • `SYNC`	synchronization barrier • `s_barrier`
atomic exchange • `ATOM.E.EXCH`	atomic vector swap • `flat_atomic_swap*`, `global_atomic_swap*` • `image_atomic_swap*`, `buffer_atomic_swap*`
atomic compare and swap • `ATOM.E.CAS`	atomic vector compare and swap • `flat_atomic_cmpswap*`, `global_atomic_cmpswap*` • `image_atomic_cmpswap*`, `buffer_atomic_cmpswap*`

Table 3.19: Table showing the translation from SASS to AMDGCN instructions for the deadlock detection parser.

AMD GPUscout deadlock detection parser

CAS	:= false	*compare and switch*
BRCafterCAS	:= false	*branch after compare and switch*
PTLDL	:= false	*potential for deadlock*

Scan through the AMDGCN instructions until:

1. An atomic compare and swap instruction is encountered.

   Set CAS to true.

2. A conditional branch instruction is encountered.

   Check if CAS is true.

   If this is the case.

   Set BRCafterCAS to true.

3. A synchronization instruction is encountered.

   Check if BRCafterCAS is true.

   If this is the case.

   Set PTLDL to true.

4. An atomic swap instruction is encountered.

   Set CAS to false.

3.9.2 Deadlock Detection Performance Metrics

NVIDIA GPUscout does not provide performance metrics related to deadlock detection, nor does AMD GPUscout.

3.10 Texture Memory Analysis

Originating in graphics programming, texture memory is a dedicated read-only resource on GPUs that is shared by all threads through specialized APIs. Unlike traditional memory systems, it optimizes access for spatially clustered patterns, such as 2D/3D data layouts, by exploiting locality to accelerate reads. Historically, this dedicated cache reduced the load on global memory and coexisting caches, but modern AMD architectures have merged the texture cache with the L1 cache. [14, sec. Device Memory]

Therefore, texture cache analysis is not really feasible on a modern AMD GPU because, unlike NVIDIA GPUs, there is no longer a dedicated texture cache. For this reason, AMD GPUscout does not perform a texture cache analysis.

However, the term texture is occasionally used in the context of the cache hierarchy of the AMD CDNA2 architecture, for example in Chapter 11.2 Dataflow in Memory Hierarchy of the MI200 ISA reference guide [7, p. 87]. The use of the term texture in this context is probably historical, as the CDNA2 architecture evolved from the GCN architecture, which was primarily designed for the gaming sector (see Subsection 2.2.1).

3.11 PC Stalling and Live Register Information

The matching of parser results with PC stalling information is generally done via the PC offset. If a performance-impairing pattern and a corresponding performance-impairing instruction are found, the corresponding PC offset can be used to match the instruction with the PC stalling information, as each recorded PC stall has a corresponding PC offset.

The live register information is specified as additional information for each instruction line and therefore has a corresponding PC offset that can be used for matching.

When a performance-impairing instruction is found by an NVIDIA GPUscout parser, all stalls for that instruction, if any, are displayed, e.g. stalled barrier or stalled branch. Additionally, for each stall reason, the percentage of stalls due to that reason compared to the total number of stalls caused by the instruction is displayed.

The live register information that is sometimes additionally displayed is always the same. Firstly, the total number of registers used at the time the instruction is executed, and secondly, if the number of registers used has increased since the previous instruction, the difference is also shown.

Table 3.20 shows whether PC stalling and/or live register information is displayed for the respective analysis.

Analysis	**PC Stalling Information**	**Live Register Information**
register spilling	✓	✓
wavefront divergence	✓	✗
atomic instruction	✓	✗
local memory	✓	✗
`__restrict__`	✓	✓
vectorized load	✓	✓
datatype conversion	✗	✗
deadlock detection	✗	✗
texture memory	✓	✗

Table 3.20: Table showing whether the respective analysis PC uses stalling and/or live register information.

3.12 Examples

This section shows examples of some AMD GPUscout analyses. The examples shown are more of a proof of concept as they are intended to explicitly generate performance degrading code patterns.

Requirements

Before using AMD GPUscout, the following points should be considered in order to be able to test the examples presented.

The ROCm Compute Profiler documentation states that `rocprof-compute` is shipped with ROCm version 6.2 or higher [12, sec. Installing and Deploying ROCm Compute Profiler]. However, this is not the case on the test system used for this thesis, so `rocprof-compute` was installed manually. In order for AMD GPUscout to find the `rocprof-compute` binary, the `rocprof-compute` directories were added to the `PATH` and `PYTHONPATH` variables.

```
export PATH=~/rocprofiler-compute/3.0.0/bin:$PATH
export PYTHONPATH=~/rocprofiler-compute/python-libs
```

`rocprof-compute` relies on a utility called `rocprof` (as explained in Section 3.1, the third version of this utility is intended to provide PC sampling for AMD GPUscout). To force `rocprof-compute` to use the newer `rocprofv2` instead of the older `rocprofv1`, the `ROCPROF` variable has been set.

```
export ROCPROF=/opt/rocm/bin/rocprofv2
```

In order for AMD GPUscout to display source line numbers for individual instructions, the binary to be analyzed must be compiled via `hipcc` with at least the `-gline-tables-only` flag (the `-g` flag is also possible).

Test System

The BEAST cluster of the Leibniz Supercomputing Centre (LRZ) served as the test system for the example and the entire work. The node on which all tests were performed is based on two AMD EPYC 7773X processors and has access to eight AMD Instinct MI210 GPUs. However, all tested kernels use only one AMD Instinct MI210 GPU.

3.12.1 Register Spilling Example

The HIP kernel in Listing 3.1 is compiled with the `-O0` hipcc flag to avoid compiler-side optimizations that would eliminate the performance degradations forced for demonstration purposes by this example.

The following kernel makes use of a lot of single registers and performs arithmetic operations with them. At the end, it outputs the calculated value. Since many individual variables are used that need to be stored in registers, and since each thread of the kernel has only a limited number of registers available, register spilling is to be expected.

The source line numbers in the listings, e.g. in Listing 3.1, can only partially be matched with the information given by the AMD GPUscout output, as the example kernels shown are not at the beginning of the source code, as in the listings. In addition, as seen in Listing 3.1, not all lines of source code that make up the kernel are presented, making the listing line numbers invalid.

```
1  __global__ void register_spilling_example(float* out, const float* in) {
2      int idx = threadIdx.x + blockIdx.x * blockDim.x;
3
4      if (idx >= N) return;
5
6      float v0 = in[idx];
7      float v1 = v0 * 2.0f + 1.0f;
8      float v2 = v1 / 1.5f - v0;
9      float v3 = v2 + v1 * 3.0f;
10     float v4 = v3 - sqrtf(v2);
11     float v5 = v4 * cosf(v3);
12
13     // follow this pattern
14
15     float v254 = v253 - sqrtf(v252);
16     float v255 = v254 * cosf(v253);
17
18     out[idx] = v255;
19 }
```

Listing 3.1: Kernel used as an example for the register spilling analysis.

Compiling Listing 3.1 with the hipcc `-Rpass-analysis=kernel-resource-usage` flag gives us information about whether register spilling occurs, which it does.

A snippet of the AMD GPUscout output is shown in Listing 3.2. AMD GPUscout detects a large number of register spills and displays performance metrics.

```
================================================================================================================
==== analysis    : register spilling
==== kernel name : register_spilling_example
================================================================================================================

==== WARNING
==== register spilling detected in file register_spilling.cpp at line 9
==== instruction: v_accvgpr_write_b32
==== vdata register: v31

...

==== WARNING
==== register spilling detected in file register_spilling.cpp at line 269
==== instruction: buffer_store_dword
==== vdata register: v0
==== the succeeding compute instruction using register v0
     after spilling was v_add_co_u32_e64 in file register_spilling.cpp at line number 272

==== INFO
==== number of cycles a wavefront in the kernel dispatch stalled waiting on memory of any kind
     1.85586e+06
==== total number of spill/stack memory instructions executed on all compute unitson the accelerator
     6896
==== number of cycles the address processing unit spent working on spill/stack instructions
     27584
==== APPROXIMATE percentage of total L2 cache requests due to private memory
==== if the percentage is high, the memory traffic between the CUs and L2 is mostly due to private
     memory (need to contain register spills)
     36.6469
```

Listing 3.2: AMD GPUscout output for the kernel used as example for the AMD GPUscout register spilling analysis.

3.12.2 Wavefront Divergence Example

The sample wavefront divergence kernel in Listing 3.3 was taken from Sen's master's thesis [1, p. 55] and adapted for HIP. The kernel processes an integer array based on the thread index. For indices less than 16, it modifies the values by doubling even numbers and incrementing odd numbers to make them even. For indices between 16 and 31, it sets the values to zero. All indices greater than 31 remain unchanged. AMD GPUscout reports the detected branches, as shown in Listing 3.4.

```
1  __global__ void wavefront_divergence_example(int* a)
2  {
3      int tid = threadIdx.x + blockIdx.x * blockDim.x;
4
5      if (tid < 16)
6      {
7          if (a[tid] % 2 == 0)
8          {
9              a[tid] = a[tid] * 2;
10         } else
11         {
12             a[tid] = a[tid] + 1;
13         }
14     }
15     else
16     {
17         if (tid < 32)
18         {
19             a[tid] = 0;
20         }
21     }
22 }
```

Listing 3.3: Kernel used as an example for the wavefront divergence analysis.

Many of the performance metrics show no or zero values. It is not really clear why this is the case, especially for performance metrics such as the number of branch instructions shown in Listing 3.4. For performance metrics related to utilisation, the very reduced examples used may not have generated enough load to cause the GPU to run into performance constraints.

```
================================================================================================
==== analysis    : wavefront divergence
==== kernel name : wavefront_divergence_example
================================================================================================

==== WARNING
==== conditional branching detected in file wavefront_divergence_example.cpp at line number 8 of your
     code, with target branch: L1
     (target branch starts at in file wavefront_divergence_example.cpp at line number 20)

==== WARNING
==== conditional branching detected in file wavefront_divergence_example.cpp at line number 20 of your
     code, with target branch: L0
     (target branch starts at in file wavefront_divergence_example.cpp at line number 23)

==== WARNING
==== conditional branching detected in file wavefront_divergence_example.cpp at line number 8 of your
     code, with target branch: L3
     (target branch starts at in file wavefront_divergence_example.cpp at line number 12)

==== WARNING
==== conditional branching detected in file wavefront_divergence_example.cpp at line number 10 of your
     code, with target branch: L2
     (target branch starts at in file wavefront_divergence_example.cpp at line number 15)

==== WARNING
==== conditional branching detected in file wavefront_divergence_example.cpp at line number 10 of your
     code, with target branch: L3
     (target branch starts at in file wavefront_divergence_example.cpp at line number 12)

==== INFO
==== total number of branch operations issued
     0
==== what percent of the kernel's duration the branch unit was busy executing instructions
     0
```

Listing 3.4: AMD GPUscout output for the kernel used as example for the AMD GPUscout wavefront divergence analysis.

3.12.3 Atomic Instruction Example

This HIP kernel in Listing 3.5 demonstrates atomic operations on both global and local memory. Each thread increments a global counter atomically, resulting in a total count of all threads. Simultaneously, within each block, threads increment a local counter, which tracks how many threads ran in that block. After synchronizing, the local counter value is recorded into a global results array, showing the number of threads per block. AMD GPUscout reports the atomic instructions found, as shown in Listing 3.6.

```
1  __global__ void atomic_instruction_example(int* global_counter, int* block_results) {
2      __shared__ int local_counter;
3
4      if (threadIdx.x == 0) {
5          local_counter = 0;
6      }
7
8      __syncthreads();
9
10     atomicAdd(global_counter, 1);
11
12     atomicAdd(&local_counter, 1);
13
14     __syncthreads();
15
16     if (threadIdx.x == 0) {
17         block_results[blockIdx.x] = local_counter;
18     }
19 }
```

Listing 3.5: Kernel used as an example for the atomic instruction analysis.

```
================================================================================================================
==== analysis    : atomic instruction
==== kernel name : atomic_instruction_example
================================================================================================================

==== INFO
==== number of global atomic instructions: 1

==== INFO
==== global atomic instruction found in file amd_hip_atomic.h at line 217 of your source code

==== INFO
==== number of shared atomic instructions in the assembly file 1

==== INFO
==== shared atomic instruction found in file amd_hip_atomic.h at line number 217 of your source code.

==== INFO
==== total number of global & generic memory atomic (with and without return) instructions executed on
     all compute units on the accelerator
     0
==== total number of cycles spent on LDS atomics with return
     0
==== total number of atomic requests (with and without return) to the L2 from all clients
     0
==== percent of write requests generated by the L2 cache that are atomic requests to any memory
     location
     0
==== total number of L2 requests to Infinity Fabric to atomically update 32B or 64B of data in any
     memory location
     0
==== number of cycles a wavefront in the kernel dispatch stalled waiting on memory of any kind
     0
```

Listing 3.6: AMD GPUscout output for the kernel used as example for the AMD GPUscout atomic instruction analysis.

3.12.4 `__restrict__` Example

This HIP kernel in Listing 3.7 performs vector addition by adding three input arrays element by element and storing the result in an output array. Each thread computes an element of the output vector by adding corresponding elements from the three input vectors. Without the `__restrict__` keyword, the compiler assumes that the input arrays overlap, which they don't, limiting the optimization.

```
1 __global__ void restrict_example(const float* a, const float* b, const float* c, float* r, int n) {
2     int tid = threadIdx.x + blockIdx.x * blockDim.x;
3     if (tid < n) {
4         r[tid] = a[tid] + b[tid] + c[tid];
5     }
6 }
```

Listing 3.7: Kernel used as an example for the `__restrict` analysis.

AMD GPUscout reports the possibilities of using `__restrict__`, as shown in Listing 3.8.

```
================================================================================================================
==== analysis    : __restrict__
==== kernel name : restrict_example
================================================================================================================

==== INFO
==== register v5, first appearing used by global load in file restrict.cpp at line 9 of your
     code, could benefit from using __restrict__
```

Listing 3.8: AMD GPUscout output for the kernel used as example for the AMD GPUscout `__restrict__` analysis.

3.12.5 Datatype Conversion Example

The kernel in Listing 3.9 performs datatype conversions on an input float array, converting it to an integer, back to a float, and also to a double.

```
1  __global__ void datatype_conversion_example(float* f_data, int* i_data, double* d_data) {
2      int tid = threadIdx.x + blockIdx.x * blockDim.x;
3
4      float f_val = f_data[tid];
5      int i_val = static_cast<int>(f_val);
6      float f_from_int = static_cast<float>(i_val);
7      double d_val = static_cast<double>(f_val);
8
9      i_data[tid] = i_val;
10     f_data[tid] = f_from_int;
11     d_data[tid] = d_val;
12 }
```

Listing 3.9: Kernel used as an example for the datatype conversion analysis.

AMD GPUscout reports the three data type conversions performed, as shown in Listing 3.10.

```
================================================================================================
==== analysis    : datatype conversion
==== kernel name : datatype_conversion_example
================================================================================================

==== WARNING
==== there are 1 F2F conversions found at the following locations:
     file name datatype_conversion.cpp line 10

==== WARNING
==== there are 1 I2F conversions found at the following locations:
     file name datatype_conversion.cpp line 9

==== WARNING
==== there are 1 F2I conversions found at the following locations:
     file name datatype_conversion.cpp line 8

==== INFO
==== total number of type conversion instructions (such as converting data to or from F32 to F64)
     issued to the VALU
     0
```

Listing 3.10: AMD GPUscout output for the kernel used as example for the AMD GPUscout datatype conversion analysis.

4 Future Work

In the future, AMD GPUscout should reach feature parity with its NVIDIA counterpart. This includes implementing support for PC stalling and live register information. As illustrated in Table 3.1, suitable tools for collecting this information have already been identified or are expected to become available. However, the existing features should also be extended and tested, since, as explained in Subsection 3.12.2, reading performance metrics sometimes raises questions about their values. Once AMD GPUscout offers the same feature set as NVIDIA GPUscout, the focus should shift towards improving the robustness of the AMD GPUscout codebase.

Currently, AMD GPUscout uses regular expressions to parse the assembly code of the target program, scanning it line-by-line for specific instructions. Instruction details, such as addresses, destination registers, or offsets, are also extracted via regular expressions. However, these expressions are currently somewhat imprecise, designed to capture a broad range of instructions to avoid missing any relevant information. Since AMD provides its ISA documentation, these regular expressions could be significantly refined and made more accurate, ensuring better parsing precision without sacrificing completeness.

Another major area for improvement is runtime efficiency, both for GPUscout in general and for AMD GPUscout in particular. Currently, each of the 8 or 9 different analyses runs a separate parser, scanning through the entire assembly file independently. This results in multiple redundant passes over the same data. A more efficient approach would involve parallelizing multiple analyses and sharing parsing results. By loading the parsed data into a shared data structure first, all subsequent analyses could access the same data without reparsing, thus significantly reducing the overall runtime.

Additionally, future updates will likely be necessary to maintain compatibility with evolving AMD architectures. Currently, AMD GPUscout is closely tailored to the MI200 series ISA, as the test system used for this research is equipped with AMD MI210 GPUs. Although the ISA generally evolves gradually, making it likely that AMD GPUscout could still support MI100 and MI300 GPUs without major modifications, future ISA changes may require adjustments to the tool to maintain its relevance and accuracy.

The broader AMD ROCm ecosystem is rapidly evolving, presenting new opportunities for integration. For example, AMD GPUscout currently uses `rocprofv2` through the ROCm Compute Profiler, but `rocprofv3`, which is already in development, promises new features such as PC sampling, which should be integrated into future versions of AMD GPUscout to leverage the latest capabilities of the ROCm stack.

Finally, additional analyses could be implemented for both AMD and NVIDIA GPUscout, expanding their diagnostic capabilities. For example, a memory coalescing analysis could help identify inefficient memory access patterns, further enhancing the tools' ability to optimize

GPU performance.

In summary, these improvements, ranging from feature parity and parsing efficiency to hardware compatibility and new analyses, will ensure that AMD GPUscout remains a powerful and up-to-date tool for analyzing and optimizing GPU workloads.

5 Conclusion

This thesis focuses on porting GPUscout to the AMD ecosystem, a task that required an in-depth exploration of AMD's programming model and a comparative analysis of key concepts between AMD and NVIDIA architectures. To establish a solid foundation, the thesis provides a detailed introduction to AMD hardware, with special attention to memory instructions and address spaces, both of which are essential for conducting GPUscout analyses.

The core contribution of this work lies in the methodical porting of NVIDIA GPUscout to AMD hardware, with a detailed description of how each GPUscout analysis was translated into its AMD counterpart. For each analysis, the thesis outlines both the parsing approach and the collection of performance metrics. For some analyses, an example was also provided that presents the output of AMD GPUscout. A notable exception is the Texture Cache Analysis, which could not be ported due to fundamental architectural differences between NVIDIA and AMD hardware.

Additionally, despite significant efforts, PC stalling and live register information were not incorporated into AMD GPUscout, as the necessary utilities for collecting this information could not be successfully deployed.

Nonetheless, this thesis represents a significant step forward in porting GPUscout to the AMD ecosystem, providing a functional tool that successfully implements two out of three major pillars of the original GPUscout framework. The effectiveness of the tool is demonstrated through numerous examples, underscoring its capability to deliver meaningful performance insights on AMD hardware. This work not only contributes to the advancement of performance analysis utilities for AMD GPUs but also lays a solid foundation for future enhancements and feature extensions in the AMD GPUscout framework.

List of Figures

List of Tables

Bibliography

[1] S. Sen. "Discovering data movement-related bottlenecks on NVIDIA GPUs". Master's thesis. Munich, Germany: Technical University Munich, 2023.

[2] 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. Denver, CO, USA: Association for Computing Machinery, 2023, pp. 1392–1402. ISBN: 9798400707858. DOI: 10.1145/3624062.3624208. URL: https://doi.org/10.1145/3624062.3624208.

[3] G. M. Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, 1967, pp. 483–485. ISBN: 9781450378956. DOI: 10.1145/1465482.1465560. URL: https://doi.org/10.1145/1465482.1465560.

[4] M. Flynn. "Very High-Speed Computing Systems". In: *Proceedings of the IEEE* 54 (Jan. 1967), pp. 1901–1909. DOI: 10.1109/PROC.1966.5273.

[5] P. Chawan, J. Patil, R. Naik, A. Madgundi, and N. Gupta. "MasPar MP-1: An SIMD Array Processor". In: *International Journal of Emerging Technology and Advanced Engineering* 2 (Dec. 2012).

[6] Y. Sun, T. Baruah, and D. Kaeli. *Accelerated Computing with HIP*. Dec. 2022. ISBN: 979-8218107444. DOI: 10.29003/m4171.978-5-317-07227-8.

[7] AMD. *"AMD Instinct MI200" Instruction Set Architecture*. Accessed: 2025-02-06. 2022. URL: https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/instruction-set-architectures/instinct-mi200-cdna2-instruction-set-architecture.pdf.

[8] P. Bauman, N. Chalmers, N. Curtis, C. Freitag, J. Greathouse, N. Malaya, D. McDougall, S. Moe, R. van Oostrum, N. Wolfe, and G. Sitaraman. *Introduction to AMD GPU Programming with HIP*. Accessed: 2025-02-11. 2021. URL: https://www.olcf.ornl.gov/wp-content/uploads/2021/04/IntroGPUProgramming-ORNL-Hackathon-May24-26-2021.pdf.

[9] AMD. *Introducing AMD CDNA 2 Architecture*. Accessed: 2025-02-07. 2021. URL: https://www.amd.com/content/dam/amd/en/documents/instinct-business-docs/white-papers/amd-cdna2-white-paper.pdf.

[10] AMD. *AMD ROCm 6.3.2 documentation*. Accessed: 2025-01-05. 2025. URL: https://rocm.docs.amd.com/en/latest/index.html.

[11] A. Fanfarillo. *CDNA2 Memory Hierarchy*. Accessed: 2025-02-03. 2023. URL: https://www.olcf.ornl.gov/wp-content/uploads/03-MemoryHierarchy.pdf.

[12] AMD. *ROCm Compute Profiler 3.0.0 Documentation*. Accessed: 2025-01-03. 2024. URL: https://rocm.docs.amd.com/projects/rocprofiler-compute/en/latest/.

[13] N. Otterness and J. H. Anderson. "Exploring AMD GPU Scheduling Details by Experimenting With "Worst Practices"". In: *Proceedings of the 29th International Conference on Real-Time Networks and Systems*. RTNS '21. NANTES, France: Association for Computing Machinery, 2021, pp. 24–34. ISBN: 9781450390019. DOI: 10.1145/3453417.3453432. URL: https://doi.org/10.1145/3453417.3453432.

[14] AMD. *HIP 6.3.2 Documentation*. Accessed: 2025-02-08. 2025. URL: https://rocm.docs.amd.com/projects/HIP/en.

[15] G. Schieffer, D. A. De Medeiros, J. Faj, A. Marathe, and I. Peng. "On the Rise of AMD Matrix Cores: Performance, Power Efficiency, and Programmability". In: *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2024, pp. 132–143. DOI: 10.1109/ISPASS61541.2024.00022.

[16] A. Fanfarillo and N. Curtis. *Register pressure in AMD CDNA2 GPUs*. Accessed: 2025-01-04. 2023. URL: https://gpuopen.com/learn/amd-lab-notes/amd-lab-notes-register-pressure-readme/.

[17] A. Mishra and C. Robeck. *Reading AMD GPU ISA*. Accessed: 2025-02-02. 2024. URL: https://rocm.blogs.amd.com/software-tools-optimization/amdgcn-isa/README.html.

[18] LLVM Project. *User Guide for AMDGPU Backend*. Accessed: 2025-02-05. 2025. URL: https://llvm.org/docs/AMDGPUUsage.html.

[19] L. Mah. *The AMD GCN Architecture - A Crash Course*. Accessed: 2025-01-08. 2014. URL: https://www.slideshare.net/slideshow/gs4106-the-amd-gcn-architecture-a-crash-course-by-layla-mah/30667382#3.

[20] NVIDIA. *PTX ISA Release 8.7*. Accessed: 2025-02-12. 2025. URL: https://docs.nvidia.com/cuda/pdf/ptx_isa_8.7.pdf.

[21] V. Indic and T. Paltashev. *The Road to PC Sampling on AMD InstinctTM MI200 GPU Series*. Accessed: 2025-02-09. 2024. URL: https://tu-dresden.de/zih/das-department/ressourcen/dateien/toolsworkshop/2024_09_20-VladimirIndic.pdf?lang=en.

[22] H. Abdelkhalik, Y. Arafa, N. Santhi, and A.-H. Badawy. *Demystifying the Nvidia Ampere Architecture through Microbenchmarking and Instruction-level Analysis*. 2022. arXiv: 2208.11174 [cs.AR]. URL: https://arxiv.org/abs/2208.11174.

[23] A. Fanfarillo and N. Curtis. *Intro to register pressure in AMD compilers*. Accessed: 2025-02-05. 2022. URL: https://www.olcf.ornl.gov/wp-content/uploads/Intro_Register_pressure_ORNL_20220812_2083.pdf.

[24] AMD. *AI Engine Kernel and Graph Programming Guide*. Accessed: 2025-02-07. 2024. URL: https://docs.amd.com/r/en-US/ug1079-ai-engine-kernel-coding/.