Isolating GPU Architectural Features Using Parallelism-Aware Microbenchmarks

Rico van Stigt
University of Amsterdam
Amsterdam, The Netherlands
ricovanstigt@ziggo.nl

Stephen Nicholas Swatman*
University of Amsterdam
Amsterdam, The Netherlands
s.n.swatman@uva.nl

Ana-Lucia Varbanescu
University of Amsterdam
Amsterdam, The Netherlands
a.l.varbanescu@uva.nl

ABSTRACT

GPUs develop at a rapid pace, with new architectures emerging every 12 to 18 months. Every new GPU architecture introduces new features, expecting to improve on previous generations. However, the impact of these changes on the performance of GPGPU applications may not be directly apparent; it is often unclear to developers how exactly these features will affect the performance of their code. In this paper we propose a suite of microbenchmarks to uncover the performance of novel GPU hardware features in isolation. We target features in both the memory system and the arithmetic cores. We further ensure, by design, that our microbenchmarks capture the massively parallel nature of the GPUs, while providing fine-grained timing information at the level of individual compute units. Using this benchmarking suite, we study the differences between three of the most recent NVIDIA architectures: Pascal, Turing, and Ampere. We find that the architecture differences can have a meaningful impact on both synthetic and more realistic applications. This impact is visible both in terms of outright performance, but also affects the choice of execution parameters for realistic applications. We conclude that microbenchmarking, adapted to massive GPU parallelism, can expose differences between GPU generations, and discuss how it can be adapted for future architectures.

CCS CONCEPTS

• **Computer systems organization** → **Multicore architectures**; • **Software and its engineering** → *Software performance*; • **Computing methodologies** → *Parallel programming languages*.

KEYWORDS

Microbenchmark; GPU; Massively parallel architecture; NVIDIA Turing; NVIDIA Pascal; NVIDIA Ampere

ACM Reference Format:
Rico van Stigt, Stephen Nicholas Swatman, and Ana-Lucia Varbanescu. 2022. Isolating GPU Architectural Features Using Parallelism-Aware Microbenchmarks. In *Proceedings of the 2022 ACM/SPEC International Conference on Performance Engineering (ICPE '22), April 9–13, 2022, Bejing, China.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3489525.3511673

*Also with European Organization for Nuclear Research.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPE '22, April 9–13, 2022, Bejing, China
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9143-6/22/04…$15.00
https://doi.org/10.1145/3489525.3511673

1 INTRODUCTION

General purpose graphics processing units (GPGPUs) are an ubiquitous part of the modern high performance computing landscape. As with other processor types, new and improved GPGPU architectures are constantly being developed in order to satisfy ever-growing performance requirements [26]. It is often the case that such improvements are implemented as new architectural features in the current and next generations of GPGPU microarchitectures.

However, as microarchitectures become more complex, the impact of such architectural innovations becomes difficult to quantify. For real applications, performance is increasingly hard to predict as run time depends on the non-trivial interplay of a large number of features across a wide range of subsystems, including the memory, the schedulers, and the arithmetic processors [1, 11]. Moreover, the increased complexity and fast development of GPGPU microarchitectures renders existing (micro-)benchmarks incomplete [4], if not obsolete [28]. Microbenchmarks for specific architectural features can help alleviate this problem by decomposing the performance of a microarchitecture as a whole into independent factors.

In this paper, we present the design and implementation of a benchmarking suite that quantifies the impact of several recent GPU architectural developments in isolation, such that they can be compared and evaluated between different microarchitectures. This feature-specific approach to microbenchmarking adds to the existing body of GPGPU benchmarking literature, which has primarily focused, so far, on holistic device-wide measurements.

Specifically, we design benchmarks to isolate features related to both memory and arithmetic, such as caches and so-called *Tensor Cores*, on three recent NVIDIA GPU architectures: *Pascal* (released in 2016) [18, 20], *Turing* (2018) [21], and *Ampere* (2020) [22, 23]. To this end, we employ fine-grained timing and a massively parallel approach to GPU microbenchmarking, thus exposing new details on operations scheduling, the impact of increasing parallelism, and the effect of complex caching policies.

Using our benchmarking suite, we have performed extensive experiments (over twelve hundred runs) to illustrate the specifics of each architecture. We find, for example, that the over-provisioning of GPU resources has different performance implications on the different architectures (section 6), and that cache operators have limited performance impact on all three GPUs (section 7).

In summary, the main contributions of this work are:

- We propose a set of feature-isolating microbenchmarks for GPGPUs.
- We extend our microbenchmarks to capture the massively parallel nature of GPGPU hardware.
- We propose a timing methodology that captures device performance on a per compute-unit level of granularity.

- We analyse and compare three recent NVIDIA GPU architectures, empirically demonstrating the performance effects induced by architectural features.
- We discuss how these features *can* impact the performance of real-world applications.

The rest of this paper is structured as follows. Section 2 introduces the basics of general purpose GPU computing, and presents the GPU architectures that are compared in this paper. Section 3 covers related work. We describe the design of our benchmarks in section 4, and our experimental setup in 5. We present a selection of benchmarking results in sections 6, 7, 8, and 9. We further analyse the implications of our microbenchmarks (and results) on application development, and reflect on some of the limitations of our current solution, in section 10. Finally, section 11 presents concluding remarks and future research directions.

2 NVIDIA GRAPHICS PROCESSING UNITS

General purpose graphic processing unit (GPGPU) programming encompasses the idea of executing general workloads on hardware designed for graphics processing. GPGPU programming leverages the massively parallel nature of graphics hardware to accelerate data-parallel tasks in a much wider domain of computational problems. Indeed, GPGPUs have been successfully deployed to fields such as machine learning, bioinformatics, physics, and many more.

The architecture of GPUs varies significantly not only between vendors, but also between different generations from the same vendor. In this work, we specifically target NVIDIA GPUs; thus, we adhere to NVIDIA's terminology wherever talking about microarchitecture features, and to CUDA terminology when presenting code or programming-related concepts. This section briefly discusses the NVIDIA features relevant for this work: the compute architecture and the memory system, the execution model, the CUDA programming model, and the most recent features added for compute efficiency.

2.1 Architecture and execution model

An NVIDIA graphics processing unit consist of a number of *streaming multiprocessors*, often referred to as SMs. Each SM contains a large number of cores, possibly diversified for different types of operations (e.g., floating-point cores or tensor cores) and a relatively small amount of control flow hardware. NVIDIA GPUs use an execution model called Single-Instruction Multiple Threads (SIMT), where groups of threads - referred to as a *warps* - are directed by the same control flow, and thus execute the same operation in lock-step, albeit on different data.

Most NVIDIA GPUs have four layers of memory: the register files and L1 caches, which are private resources for each SM, and the L2 cache and the global memory, which are shared across the SMs [16]. These different memories vary wildly in throughput and latency, and selecting the right memory is often critical for high-performance applications [17].

2.2 CUDA

The most common programming model for NVIDIA GPUs is *CUDA*. Within CUDA, computational kernels consist of a multi-dimensional *grid*, which is subdivided into *blocks*, which, in turn, consist of threads [9]. These CUDA concepts map onto the hardware features of the GPU: grids roughly map onto entire devices, blocks map onto SMs, and threads map onto individual cores.

Table 1: Data types and tensor sizes for tensor operations.

A ($M \times K$), B ($K \times N$)	C ($M \times N$)	Tensor ($M \times N \times K$)
16-bit float	32-bit float	$16 \times 16 \times 16$
16-bit float	32-bit float	$32 \times 8 \times 16$
16-bit float	32-bit float	$8 \times 32 \times 16$
16-bit float	16-bit float	$16 \times 16 \times 16$
16-bit float	16-bit float	$32 \times 8 \times 16$
16-bit float	16-bit float	$8 \times 32 \times 16$
8-bit integer	32-bit integer	$16 \times 16 \times 16$
8-bit integer	32-bit integer	$32 \times 8 \times 16$
8-bit integer	32-bit integer	$8 \times 32 \times 16$

In CUDA, for a block to be executed, it must first be assigned onto an SM - that is, the block becomes *resident* on the SM. Consequently, a portion of that SM's resources, such as registers and shared memory, are dedicated to that block. Once resident on an SM, a block remains resident until it has ceased execution. When resources permit it, multiple blocks can reside on the same SM, potentially improving performance and/or efficiency. However, the allocation of blocks to SMs is performed in hardware: programmers cannot determine the allocation directly. Instead, programmers must tune their use of threads, registers, shared memory, and other resources carefully.

CUDA applications consist of host-code and device-code: the former executes on the CPU, while the latter, encapsulated in *kernels*, executes on the GPU. During the compilation of a CUDA program, the host code is compiled for the CPU, with a traditional compiler, while device code is compiled (with a CUDA compiler) to *Parallel Thread eXecution*, henceforth referred to as PTX [24, 25].

2.3 Reduced-precision computing

To expose the trade-off between computational efficiency and precision [6], NVIDIA GPUs support several new floating point formats. Support for IEEE 754 16-bit floating point numbers was adopted in the *Maxwell* microarchitecture [10], and support for the machine learning-optimised 16-bit *Brain floating point* format (`bfloat16`) format was introduced in the *Ampere* architecture [23]. In addition, recent architectures provide increased performance for 8-bit and 4-bit integer operations.

2.4 Tensor cores

Starting with the *Volta* microarchitecture (released in 2017), NVIDIA GPUs feature *Tensor Cores*: additional computing hardware, specifically designed to efficiently perform matrix multiply-accumulate operations at different (mixed) levels of precision. Two of the microarchitectures featured in this paper, *Turing* and *Ampere*, are equipped with tensor cores, while *Pascal* is not. Tensor cores are embedded in each SM, and are shared between the threads executing on that SM.

While tensor cores are still programmed using CUDA, each tensor operation is performed by an entire warp instead of a single thread. This is referred to as *warp-wide* programming. In addition, tensor operations require a significant amount of pre-processing. For each warp, matrix fragments for input matrices A and B, and the accumulator matrix C, must be declared. For each of these fragments, data types and sizes have to be provided, from a limited set of possible configurations, seen in Table 1. Then the fragments have to be filled from memory; the provided memory sections also need to be properly aligned [15].

3 RELATED WORK

To illustrate the novelty of our microbenchmarking suite, we provide a brief analysis of relevant related work for GPGPU benchmarking in general, and microbenchmarking in particular.

There are several benchmarking suites commonly used to compare GPU architectures. For example, Rodinia [5] and SPECAccel [14] combine *by construction* applications with different compute- and memory-bounds, thus providing a comprehensive analysis of the performance advantages and limitations of GPUs for a large spectrum of workloads. Similarly, SHOC [7] also combines multiple applications, while further grouping them into complexity levels. By virtue of using multiple, potentially portable, programming models, SHOC, Rodinia, SPECAccel are further available to non-NVIDIA GPU. More specialised benchmarks, like Gardenia [29], further assess the ability of GPUs for domain-specific applications. Compared to these suites, our work addresses GPU features in isolation, thus bearing some resemblance to the SHOC level 0 codes, although we expose finer-granularity features.

Zooming in to different features of the GPUs, memory analysis using suites such as GPU-STREAM v2.0 [8] provides an accurate account of GPU memory throughput (as accessed through different programming models), but does not go further into investigating the hardware features that impact the observed performance.

GPGPU microbenchmarking has also been attempted before. In [28], authors demonstrate how to reverse engineer machine features using microbenchmarking data; in our work, we follow a similar idea, while analysing newer, more complex GPGPU microarchitectures. In MIPP [4], the authors also propose a microbenchmarking suite, but focus exclusively on detecting operation costs in terms of cycles and energy. Our microbenchmark targets newer architectures, and enables more insight into microarchitectural details such as scheduling or multi-layer caching. Last, but not least, the latest work on microbenchmarking NVIDIA GPUs [12, 13, 16] also focuses on microarchitectural details, but limits their analysis to a single SM. Instead, our work captures *by construction* the massively parallel nature of the GPGPU, thus uncovering more scheduling-related issues than all previous work.

Finally, of specific importance to this work are the platform-specific documents released by NVIDIA: we used the programming guide [24] and the white papers extensively [18, 20–23] to build our microbenchmarking suite. However, the results of our microbenchmarking suite provide new insights, interpretations, and performance data beyond those included in these sources.

4 BENCHMARKING PRINCIPLES

Our microbenchmarking suite is designed to meet three core requirements: *feature isolation*, *massive parallelism* and *fine-grained timing*. To isolate specific architectural features, each microbenchmark must capture *one* specific feature, and be affected to the smallest degree possible by unrelated features. In addition, as applications of GPUs to real-world problems are virtually always massively parallel [26], a realistic analysis of GPU features must be performed in a massively parallel environment. Finally, we must provide accurate, fine-grained timing mechanisms, that can isolate performance at warp-level. In the reminder of this section, we briefly describe how we approached the design of the microbenchmarks to address these three requirements.

4.1 Feature isolation

Our benchmark facilitates the analysis of both the memory and the arithmetic subsystems of GPGPUs. The features we target are: (1) memory latency, (2) bandwidth, (3) arithmetic performance (in the context of variable precision), and (4) tensor cores performance. For each of these features, we design and/or adapt specific microbenchmarks. Thus, we *adapt* a synthetic benchmark for memory latency by Mei and Chu [16] (see section 6), we *design* a specific benchmark for bandwidth analysis (see section 7), based on matrix transpose and the specifics of caching in new GPGPUs, we use a basic synthetic benchmark for variable precision arithmetic (see section 8), and we refactor a basic matrix multiplication operation to assess tensor core performance (see section 9). For memory-related microbenchmarks, we virtually eliminate (expensive) arithmetic operations, while for arithmetic microbenchmarks we eliminate any unnecessary memory operations (i.e., we enforce the use of registers).

4.2 Massively parallel microbenchmarks

To capture the massively parallel nature of GPGPUs, we use a systematic approach for extending single threaded microbenchmarks to highly parallel ones. In this process, we take into account the design of the GPU: the size of hardware blocks and SMs, how many SMs are available on the entire GPU, and the resources available on each block and SM. In practice, we follow a two step process: first we scale single-threaded benchmarks up to use an entire SM by utilising multiple threads per block, and then we use multiple blocks to scale up from single SM execution to GPU-wide execution. Given that we are interested in very specific architectural features, we ensure that each thread operates a completely separate microbenchmark instance, thus preventing communication overhead from interfering with the results.

We note that, although we focus in this paper primarily on NVIDIA GPU architectures, our approach could be ported to other GPUs, too. Because we align our approach with the design of the hardware, we are able to scale benchmark parallelism based on the characteristics of the used device. As a result, for GPUs that share similar architectural principles, this methodology will be portable. Possible targets include past and future NVIDIA architectures, but also architectures from other vendors such as Intel and AMD.

4.3 Mechanisms for accurate timing

Naively, we can time GPU code by a simple differential method, where we capture the time before and after the execution of the computational kernel, and equate the execution time of the kernel to the (absolute) difference between the recorded timestamps. This approach, however, can be too coarse for the microbenchmarks we describe in this paper, because it amortises the kernel execution time

Table 2: The GPU systems used for benchmarking.

System	GPU	CPU	Mem. size
ShowCees	GTX 1080 Ti	2 × Intel Xeon Gold 6148 (2.4 GHz)	376 GiB
DAS-5 [3]	RTX 2080 Ti	2 × Intel Xeon Silver 4110 (2.1 GHz)	96 GiB
DAS-6 [3]	A100	2 × AMD EPYC 7402 (2.8 GHz)	2 TiB

over the entire computation and device, and hides any potentially interesting scheduling behaviour (which is likely to emerge in a massively-parallel execution model).

Instead of this naive approach, we expose more fine-grained timing results by instrumenting the kernels at both *block*- and *warp-level*, using PTX special registers: the `%clock`, `%clock_hi`, and `%clock64` can be accessed to provide clock cycle-accurate timing. The value of these registers can be combined with the values of the `%warpid` and `%ctaid` registers, which provide insight into the logical execution space of the kernel, as well as the `%smid` register which provides information about the hardware on which the code is being executed. Combined, these registers provide the granularity required to determine when specific blocks and warps begin and end execution, and further infer how far along in their workload a block or warp are at a given time.

Using this fine-grained timing information, we investigate the way blocks are scheduled across multiprocessors, as well as the way warps are scheduled within the multiprocessor on which they are resident. Such information provides us valuable insight into (often proprietary) scheduling mechanisms on GPUs, which are otherwise esoteric and sparsely understood [1]. We can also expose the interference between blocks when many of them are running in parallel; the contention of resources such as the L2 cache and the main memory bandwidth can be identified using discrepancies between the run times of jobs at the block level, for example.

We note that our fine-grained timing approach is used in conjunction with (and not as a replacement for) more traditional kernel-wide timing; in fact, we use *both*, in order to provide kernel timing data at multiple levels of granularity. Where applicable, we use our fine-grained timing strategy to elucidate the scheduling behaviour of the GPU; to this end, we track the times at which individual threads and blocks in our kernel begin and end their execution. By then comparing these data along a temporal axis, we can determine precisely when the on-chip scheduling mechanism schedules certain parts of the execution kernel. In this work, we show results about scheduling, gathered using this technique, when the devices are under memory-intensive workloads (section 7.4), arithmetically-intensive workloads (section 8.3), and when the tensor cores are heavily utilised (section 9.3).

5 EXPERIMENTAL SETUP

All the experiments presented in this paper are executed on three distinct machines, presented in Table 2. For all of the experiments presented in this paper, we ensured that our benchmarks were the sole occupants of the machines' resources for the entire duration of the experiments. The characteristics of the GPUs we study in this work are further detailed in Table 3.

6 MEMORY LATENCY

In our first set of experiments, we aim to elucidate the impact of contention on memory access latency at various levels of the memory hierarchy.To this end, we adapt an existing memory microbenchmark [16], which relies on a variant of pointer chase in which timing measurements are performed for individual accesses, rather than as an aggregate over a larger number of reads. These measurements are stored in the low-latency shared memory to reduce overhead.

Table 3: The GPU devices analysed in this work.

	Device		
Feature	GTX 1080 Ti	RTX 2080 Ti	A100
Microarchitecture	*Pascal*	*Turing*	*Ampere*
Chip identifier	GP102	TU102	A100
Compute Capability	6.1	7.5	8.0
Clock frequency			
Base	1480 MHz	1350 MHz	765 MHz
Boost[1]	1582 MHz	1545 MHz	1410 MHz
Number of SMs	28	68	108
SM configuration			
Number of compute cores			
FP32	128[2]	64	64
FP64	4	2	32
INT32	128[2]	64	64
Number of warp schedulers	4	4	4
Number of *Tensor Cores*	N/A[3]	8	4
L1 cache size	48 KiB	64 KiB[4]	192 KiB[4]
Memory subsystem			
Total size	11 GiB	11 GiB	40 GiB
Bus type	GDDR5X	GDDR6	HBM2
Bandwidth	484 GB/s	616 GB/s	1555 GB/s
L2 cache size	2816 KiB	5632 KiB	40 960 KiB

[1] The frequencies given in this table are derived from the base specifications, frequency ranges may differ in experiments.
[2] *Pascal* does not distinguish between compute cores for FP32 and INT32.
[3] *Pascal* does not feature *Tensor Cores*.
[4] For *Turing* and *Ampere*, the L1 cache and shared memory are unified.

6.1 Microbenchmark design

We investigate the effects of two different kinds of contention: within a single block, by varying the number of warps in that block, and between blocks, by executing a multi-block kernel on the same SM. In our single-block experiment, all warps share a common buffer, which is accessed by multiple threads in a sequential pattern, such that the same data is read by all warps at the same time. In the multi-block experiment, each block has its own dedicated buffer. In order to accurately measure the impact of contention, we record timing data about every individual read access. We record not only *the duration* of each access, but also *the moment* at which the access takes place. By collecting these data for each warp, we can indirectly measure the impact of concurrent warp execution.

6.2 Warp-level scheduling

The results of the experiments with a *single block* are shown in Figure 1. We note that, for a single warp per block, the access time per element is very consistent, and the entire buffer is read in roughly 25 000 cycles for all of three GPUs. However, when using 32 warps, on *Pascal*, the overall execution time increases to over 50 000 cycles. In addition, we observe a discrepancy between the throughput of different warps: low-ID warps execute at a constant throughput, and finish in around 40 000 cycles, while high-ID warps exhibit throughput changes during execution. This indicates that the scheduler favours low-ID warps and, once they complete, the high-ID ones gain precedence, and their throughput increases.

The *Turing* and *Ampere* architectures exhibit this contention pattern to a far lesser extent: while the total execution time still

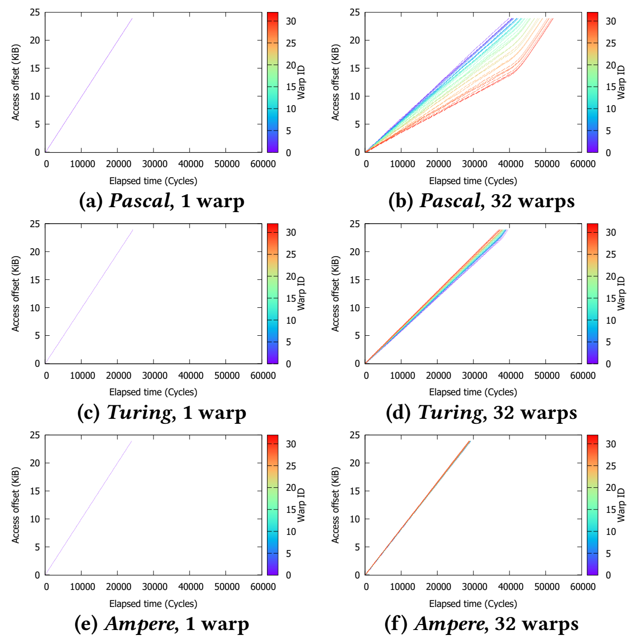


Figure 1: Per-warp memory latency for a single block. Individual accesses are represented as horizontal lines with length proportionate to the access latency.

increases as the number of warps does, the difference in execution time between the shortest- and longest-running warps is much smaller than on *Pascal*. On *Pascal*, this difference exceeded 10 000 cycles. Meanwhile, the difference is reduced to roughly 3000 cycles on *Turing*, and this discrepancy all but disappears on *Ampere*.

6.3 Block-level scheduling

The results for the experiments with *multiple blocks*, shown in Figure 2, indicate another significant difference between the microarchitectures: when only a single block is run (i.e., no contention), the observed latency is around 100 cycles. When we run two blocks on the same SM, *Pascal* still shows no contention: the individual accesses perform similarly to the single-block experiment. When utilising three (or more) blocks, the effects of contention are visible: two of the three blocks interfere with each other, and thus we observe an access time of 250 cycles, while the third block is unaffected, preserving its access time of 100 cycles. On *Turing* and *Ampere*, the effect of contention is apparent with as few as two blocks. This indicates that *Pascal*'s SM cache consists of two halves, each only accessible by half of the cores; the *Turing* and *Ampere* SMs have a single block of cache, accessible to all cores in the SM. In essence, this behaviour indicates that the *Pascal* multiprocessor is split in two halves, which is not the case for the two other architectures.

7 BANDWIDTH

We further present our microbenchmarks for device memory and cache *bandwidth*. In particular, we investigate the L1 cache bandwidth when retrieving data using different instructions. We further

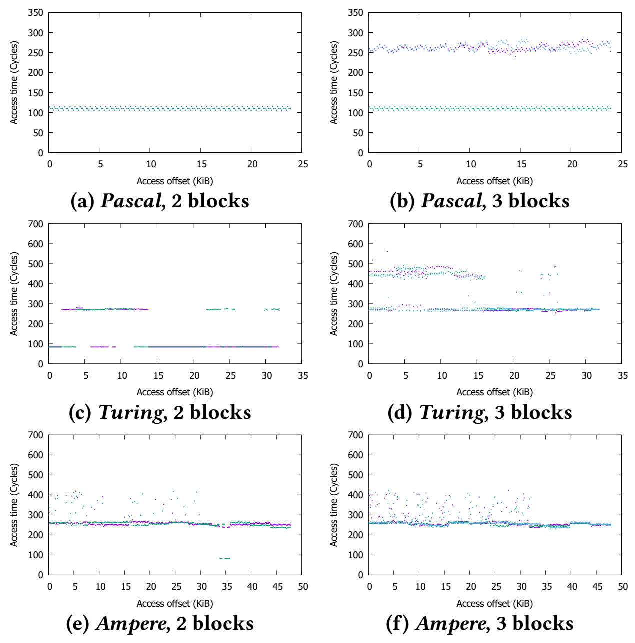


Figure 2: Access latency with multiple blocks per warp reading separate buffers. Blocks are identified by colour.

explore the effects of *cache operators*, a feature of the PTX instruction set which provides fine-grained control over cache mechanisms, on memory bandwidth.

7.1 Microbenchmark design

In this section, we present the design of our two bandwidth experiments. Since the goals of these microbenchmarks are very different, they require significantly different design approaches.

7.1.1 L1 cache bandwidth. To measure the achievable bandwidth from the L1 cache, we base our microbenchmarks on work by Jia et al., which we extend by utilising multiple blocks with a variable number of warps per block [13]. In these experiments, each block accesses a separate buffer, but all warps within a block access the same buffer. To measure the performance of each warp separately, we register timestamps at the start and end of each warp (see subsection 4.3. Because we observed the non-negligible impact of the operations and data-types used in [13], we use 32-bit float numbers, and compare the behaviour of three operations: ADD, NOT, and MOV.

7.1.2 Caching policies. CUDA, and the underlying PTX byte code, allow for cache hints to be passed to the memory subsystem, in order to influence the caching behaviour of a specific memory operation. Developers can obtain fine-grained control over cache hints by using specific cache operators to PTX code, and further inlining this code in C/C++ code [24, 25]. We use this method - i.e., inlined PTX code with cache operators - to apply cache modifiers to data movement operations. It is worth noting that support for cache operators through high-level interfaces has been expanded in CUDA 11.5 through so-called *annotated pointers* [2]. We consider this a valuable development as it increases the applicability of the results of our microbenchmarks to real-world applications.

The PTX instruction set provides five cache operators for memory load instructions, which are as follows [25]:

`ld.ca` cache loaded data at all levels (commonly L1 and L2).
`ld.cg` cache loaded data at the L2 level, but not at the L1 level.
`ld.cs` cache loaded data at all levels, but with an evict-first policy.
`ld.lu` indicates the last use of a specific cache line, equivalent to `ld.cs` for global memory.
`ld.cv` do not cache loaded data; any existing cache lines containing that data are considered stale.

In addition, there are four cache operators for memory store instructions, which are the following [25]:

`st.wb` cache stored data at all coherent levels.
`st.cg` cache stored data at the L2 level, but not in the L1 level.
`st.cs` cache stored data at all levels, but with an evict-first policy.
`st.wt` do not cache stored data: write to system memory, writing through the L2 cache.

As an example of how cache operators work in PTX, the following are two instructions that load a 32-bit floating point value from global memory to a register. The first instruction is not decorated with a cache operator, while the second instruction is augmented with a cache hint through the addition of a period-separated qualifier (in this case, `ca`):

```
ld.global.f32 r1, [r2];
ld.global.ca.f32 r1, [r2];
```

In cases where a cache operator is not provided, a default is used. This default depends on the chip on which the code is executed: for the *Pascal* (GP102) chip examined in this work, the default load operator is `ld.cg` [19]. For the *Turing* (TU102) and *Ampere* (A100) chips, the default load operator is `ld.ca` [25]. For all three chips, the default store operator is `st.wb` [25].

To better understand the effect of the cache operators, we apply them to an implementation of matrix transposition. We have chosen this application as it is purely memory bound, allowing us to measure the effect of cache operators with as little interference from outside factors as possible. In addition, this application utilises both read and write instructions, allowing us to examine the entire parameter space for load and store operations.

We create three different implementations for matrix transposition, where both the input and output matrices use the same row-major data format. From our previous experiments we know that performance is not necessarily maximised by increasing the number of threads or blocks; rather, these values must be carefully chosen to fit the workload. As such, these benchmarks are all parameterized by the number of blocks, the size of each block, and the total size of the workload. The first implementation reads in column-major order and writes in row-major order, whereas the second implementation does the reverse. Finally, the third implementation loosely takes into account the cache architecture: each thread moves an entire cache line (as opposed to a single element), which should improve cache utilisation.

Since our implementations are configurable, we determine the optimal execution configuration (that is, the number of threads per block and the number of blocks) for each of the cache operators. We empirically found that the optimal block sizes are 64 for *Pascal*, 128 for *Turing*, and 512 for *Ampere*. These values, in turn, are used to determine the optimal number of blocks as follows: we increase the number of blocks, in increments of the number of SMs on the GPU, until no performance improvement is observed.

7.2 L1 cache bandwidth

The results of our L1 cache experiments, presented in Figure 3, highlight the high impact of instruction selection on bandwidth: ADD reduces the achieved performance to only 40 bytes per cycle, while the other instructions show two to three times better performance. We also note that, for *Pascal*, peak performance is only achieved when we use two or more blocks per SM; for *Turing* and *Ampere*, the best performance is achieved with only a single block. We interpret this difference as further evidence that the *Pascal* SM is split into two largely independent halves.

7.3 Cache operators

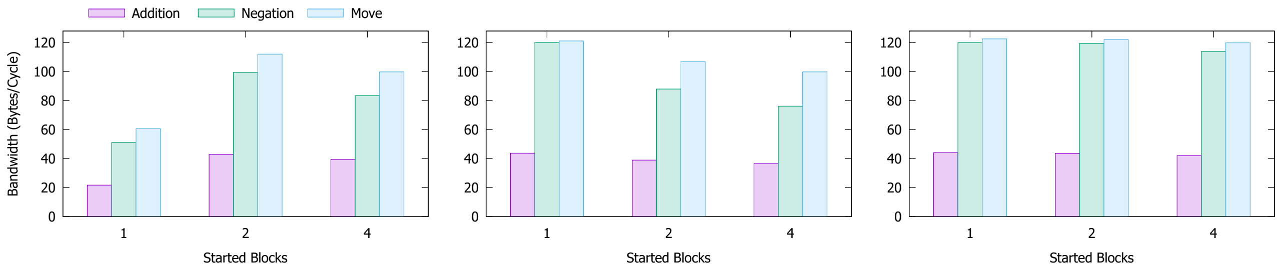
The impact of the chosen cache operator and the number of blocks started for each of the three implementations is shown in Figure 4. We show only the impact for the `ld.ca` and `ld.cg` load operators in combination with the `st.cg` store operator, as these are the only read operators which noticeably impacted performance on any device. The different write operators did not impact the achieved throughput in a meaningful way and are thus entirely excluded from the results for the sake of brevity.

We notice that the row-major read implementation achieves much lower throughput compared to the other implementations. This implementation is also not noticeably impacted by the different cache operators, whereas the other implementations show an increase in throughput when `ld.ca` is used. This is especially visible for the cache-optimised implementation on *Ampere*, where the throughput is roughly doubled with `ld.ca`. In this configuration, we achieved roughly 50 GiB/s higher throughput than for the column-major read implementation; for the other GPUs the performance is unaffected.

For all tested configurations, the optimal number of blocks for the cache-optimised implementation is a *different* multiple of the number of available SMs ($|SM|$). Specifically, *Pascal* reaches optimal performance for $4 \times |SM|_P = 112$ blocks, *Turing* shows optimal performance for $3 \times |SM|_T = 204$ blocks, and *Ampere* requires $|SM|_A = 108$ blocks. The performance penalty when using other values differs based on both the transposition implementation and the GPU. Using more or fewer blocks does not have a significant impact on the row-major read implementation on the *Pascal* and *Turing* GPUs; however, on *Ampere*, the effect of sub-optimal kernel configuration is far more pronounced, with a peak at 108 blocks started. Finally, for all of the used GPUs, we observe a drop in achieved bandwidth when a large number of blocks are started, even if that number is a multiple of the number of SMs. A more in-depth analysis including all combinations of cache operators is given by van Stigt [27].

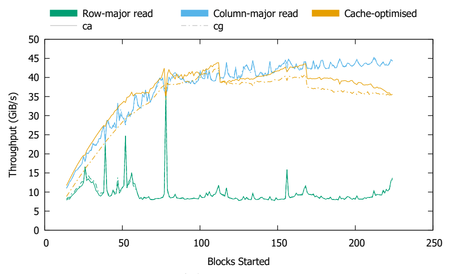
7.4 Scheduling

Finally, Figure 5 shows the impact of scheduling on the execution of an SM. For the *Pascal* architecture, we observe that two blocks execute simultaneously (i.e., they start and end at roughly the same time). For *Turing*, only a single block is executed at a time. *Ampere* shows yet another kind of behaviour: the first two blocks start

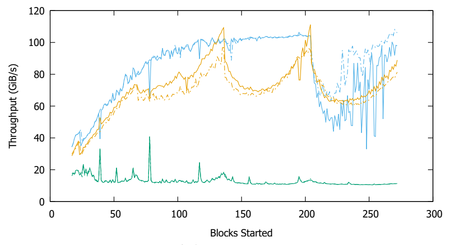


(a) *Pascal* (b) *Turing* (c) *Ampere*

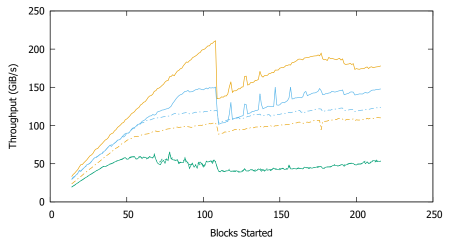
Figure 3: L1 Bandwidth with varying numbers of blocks per SM and different operations.



(a) *Pascal*



(b) *Turing*



(c) *Ampere*

Figure 4: Throughput using the `ld.ca` and `ld.cg` load operators, and the `st.wb` store operator on a 5000×5000 **matrix.**

executing simultaneously, but the second block takes much longer to terminate. Once the second block has terminated, blocks three and four run sequentially, taking roughly the same amount of time to complete. We speculate that *Ampere* performs more optimistic scheduling, starting execution of additional blocks even if there are insufficient resources available. In addition, we observe significant scheduling delays between blocks on both *Pascal* and *Turing*. We do not currently have an adequate explanation for why this behaviour occurs, and why *Ampere* does not exhibit it.

8 ARITHMETIC PERFORMANCE

Perhaps the most common application of GPGPUs is to perform massive computations *very quickly*. Thus, arithmetic performance is a very important overall performance indicator for any GPGPU. As a result, there are existing microbenchmarks for the arithmetic performance of graphics processing units [4]. In order to expand on this previous work, we focus specifically on two features that are less commonly analysed: the performance when using reduced precision arithmetic, and scheduling behaviour under arithmetically intense workloads.

8.1 Microbenchmark design

Our arithmetic microbenchmark investigates the performance of fused multiply-add operations on various data types. The benchmark is designed to operate on registers only, in order to eliminate the influence of the memory subsystem. Furthermore, the control flow consists of a manually unrolled loop executing eight instruction in sequence, thus minimising the impact of pipeline stalls. In each case, the loop is executed ten million times, corresponding to eighty million instructions, and one-hundred and sixty million floating point operations. We measure the performance of these operations on a per-warp level, with a single block of variable size. The experiments are run for five different data types: 64-bit double precision floating point numbers (FP64), 32-bit single precision floating point numbers (FP32), 16-bit half precision floating point numbers (FP16), packed 16-bit floating point numbers (FP16x2), and 32-bit integers (INT32). In our second set of experiments, we investigate the effects of scheduling on the throughput of arithmetic workloads. To this end, we utilise the same general approach, with 32 warps (1024 threads) per block and up to four blocks per SM. These values were found empirically.

8.2 Variable-precision computing

The results of the floating point arithmetic microbenchmark are shown in Figure 6. As expected given the low number of arithmetic cores for these data types, FP16 and FP64 performance on *Pascal* is very poor, whereas FP32 and INT32 performance is much better. However, when using only a single block, *Pascal* actually offers better performance for FP32 and INT32 than *Turing* and *Ampere*, a direct result of the *Pascal* architecture featuring a larger number of arithmetic cores per multiprocessor (Table 3).

For *Turing*, we note that the dedicated FP16 hardware provides significant performance benefits, achieving roughly twice the performance of the FP32 benchmark on the same device. We do note a difference between FP16 and packed FP16, where the latter performs worse. FP64 performance is, similar to *Pascal*'s, very poor due to the reduced number of double-precision arithmetic cores.

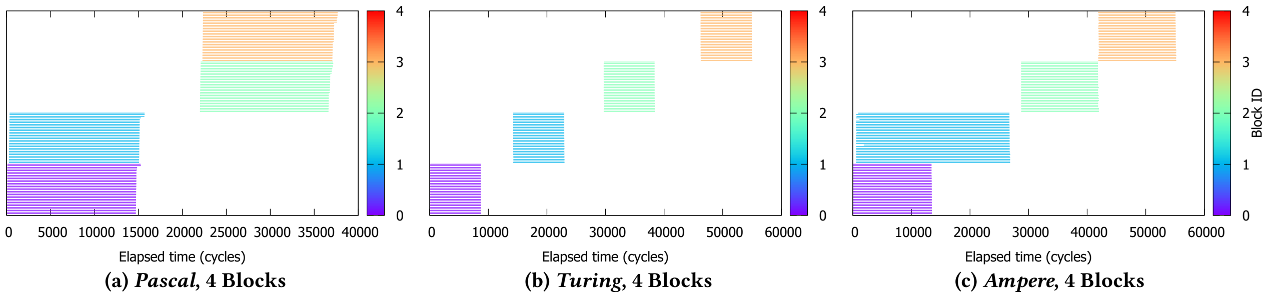


Figure 5: Memory-intensive blocks scheduling. Warps are represented by single lines, and blocks are represented by colour.

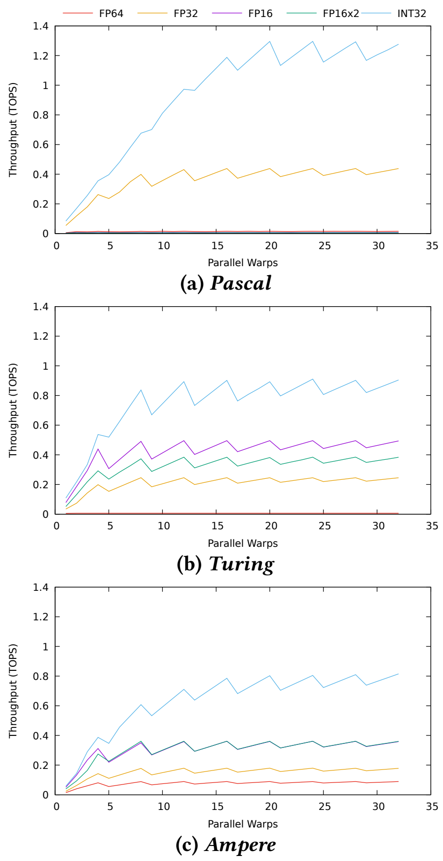


Figure 6: Integer and floating point throughput of fused multiplication-addition operations for various data types, with a variable number of warps in a single block.

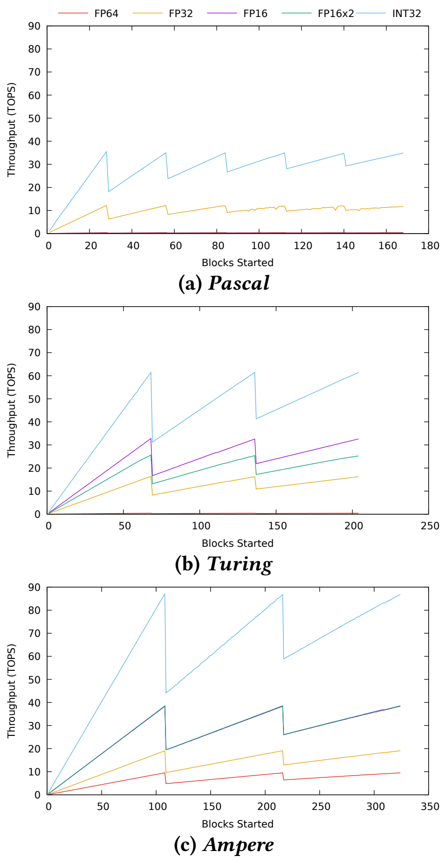
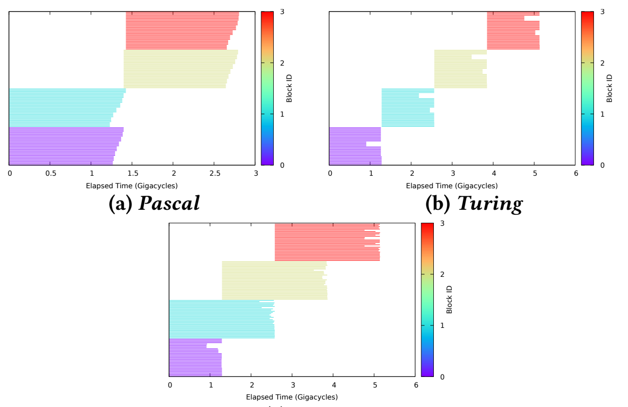


Figure 7: Integer and floating point throughput of fused multiplication-addition operations for various data types, with a variable number of blocks started, 32 warps (1024 threads) per block.

The achieved INT32 performance is much higher than either the FP16 or FP32 performance for all of the GPUs we tested.

Moving on to *Ampere*, we observe similar performance to the *Turing* GPU, albeit slightly lower due to the lower clock frequency (Table 3). *Ampere* does show much better FP64 performance compared to the other devices, and the difference between FP16 and packed FP16 is negligible. As for the different data types, we make the following observations: (1) integer operations offer much higher performance than any of the floating point types: 32-bit integers achieve roughly twice the performance of the fastest floating point format on *Turing* and *Ampere*, and this increases to a factor three on *Pascal*; (2) compared to floating-point data types, integer operations require more warps per block to reach optimal performance; and (3) the GPUs featuring hardware implementations for different floating point types, we see an inverse relation between the size of the datatype and the achieved performance, with each doubling of the lane width halves the achieved performance.

Scaling our experiment up to use multiple blocks in Figure 7, we observe that the results are largely as expected. For *Pascal*, the best performance is achieved when a multiple of 28 blocks is used; while this is expected based on the GPU specifications, it does



(a) ***Pascal*** **(b)** ***Turing*** **(c)** ***Ampere***

Figure 8: Arithmetic scheduling behaviour on different GPUs. Warps are represented by single lines, and blocks are represented by colour.

not necessarily align with our findings of a split SM design (see section 7). For both *Turing* and *Ampere*, the highest throughput, approximately equal to the peak reported in the specifications, is achieved when the number of blocks is a multiple of the number of SMs. Finally, the best performance for all tested GPUs is achieved when INT32 is used, with FP32 achieving roughly four times fewer operations per second. These results display a *sawtooth* pattern due to load imbalance: towards the end of the execution, there are insufficient remaining units of work to occupy all multiprocessors, leaving some of them idle. As the amount of work increases, the overhead of this inefficiency is amortised to zero. The achieved fraction of peak performance f for a given number of threads n on a device with $|SM|$ multiprocessors is dependent on the remainder of the total amount of work modulo the number of multiprocessors, and given by the following function:

$$f(n; |SM|) = \frac{n}{|SM| \left\lceil \frac{n}{|SM|} \right\rceil}$$

8.3 Scheduling

The results of our scheduling experiments are shown in Figure 8. The *Pascal* architecture, once again, appears to execute two blocks simultaneously. However, while this doubled the overall bandwidth in our memory-related benchmarks, the effects on arithmetic throughput are different. When we use a single block, the kernel takes 700 million cycles, while running two blocks simultaneously takes twice as much (1.4 billion cycles) to process twice as much work. We also note a discrepancy between different warps on the same block: some warps finish execution slightly earlier when two blocks are executed simultaneously. *Turing* shows that only a single block executes at a time. Like with *Pascal*, some warps within a block finish execution notably earlier. Finally, *Ampere* exhibits similar behaviour to the *Turing*, but the scheduling is different: at the very beginning, two blocks are started simultaneously, but the second block takes twice as long to terminate. Once the first block terminates, the third block starts, which then also takes twice as long to execute as the first block. This phenomenon creates a staircase-like pattern, which continues until all of the blocks have terminated. Also for *Ampere* some warps finish earlier than other warps within the same block.

9 TENSOR CORES

Tensor cores provide hardware-accelerated mixed-precision matrix multiplication, and were originally designed for machine learning workloads. Our benchmark evaluates their performance in terms of computational throughput. Because tensor cores are only present in the *Turing* and *Ampere* architectures, the *Pascal* architecture is not included in this comparison.

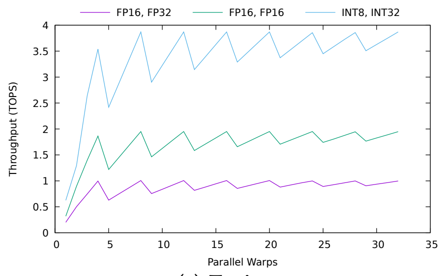
9.1 Microbenchmark design

To microbenchmark the tensor cores in isolation, we apply tensor cores to matrix multiplication problems in synthetic environments. Our experiments are designed to eliminate the influence of the memory subsystem on the performance of the tensor cores to the greatest possible degree. In order to achieve this, we repeatedly apply operations to relatively small ($16 \times 16 \times 16$) tensors which allows us to forego the overhead of retrieving data from the device memory, which would be an important factor in the performance of a real-world matrix multiplication program. These experiments aim to identify the weak scaling properties of the devices, as the workload increases with the number of warps. Throughout these experiments, we use the different configurations listed in Table 1, and we repeat the tensor operations 10^7 times.

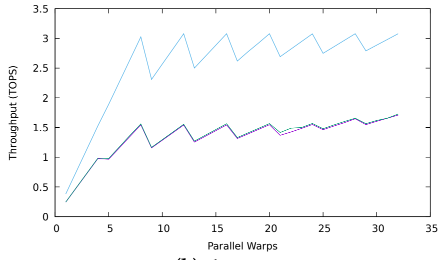
9.2 Synthetic performance

Figure 9 shows the performance of our synthetic benchmark. For both devices, we observe that peak performance is achieved when the number of warps is a multiple of four. This is likely because both architectures have four warp schedulers per multiprocessor [24]. Notably, the *Turing* architecture achieves performance close to its peak performance with as few as four warps, while the *Ampere* device does not reach peak performance until eight warps are used. This result is unexpected, as the *Turing* and *Ampere* architectures have eight and four tensor cores per multiprocessor, respectively. As such, one might expect the *Turing* architecture to achieve only half of its peak performance when four warps are used. Similarly, the *Ampere* tensor cores would be saturated by as few as four warps.

Between different data types, we see similar relative performance to what we describe in section 8: whenever the lane width doubles, the performance halves. The notable exception to this is on the *Ampere* architecture, where tensor operations on 16-bit floating point numbers have approximately similar performance, regardless of whether the accumulator tensor is half or single precision. On the *Turing* architecture, the difference in performance for different accumulator widths is as expected: the *Turing* white paper posits peak per-SM tensor core performance of 3.2 TOPS for 8-bit integer operations, and 1.6 TFLOPS and 0.8 TFLOPS for floating point operations with a half-precision and single-precision accumulator, respectively [21]. The data in Figure 9 exceeds these numbers significantly, which is explained by the device used in these experiments achieving a significantly higher boosted clock speed than that listed in the white paper (our device clocks at 1.930 GHz, while the white



(a) *Turing*

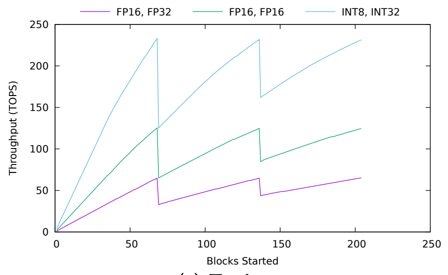


(b) *Ampere*

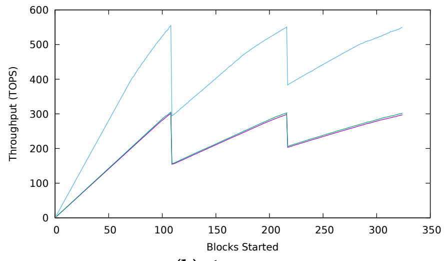
Figure 9: Throughput of tensor cores operating on $16\times16\times16$ **tensors, with different datatypes and a single block of varying size.**

paper device clocks at 1.545 GHz). After adjusting for this discrepancy, the achieved performance is not only within the specifications, but also very close to the theoretical peak performance.

As with our other experiments, we also extend these synthetic tensor core experiments to cover the entire GPU by spawning multiple blocks. The results, shown in Figure 10, are consistent with the results of the GPU-wide experiments in section 8. This is an expected result, as there is no contention for resources that are shared between multiprocessors, such as the memory. As a result, no performance degradation should appear when scaling

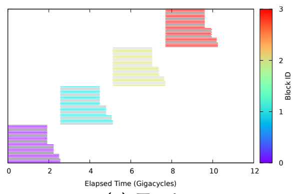


(a) *Turing*

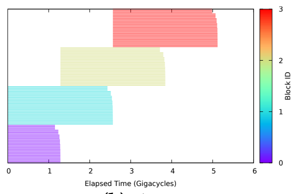


(b) *Ampere*

Figure 10: Throughput of tensor cores operating on $16\times16\times$ 16 **tensors of varying data types with a varying number of blocks of fixed size.**



(a) *Turing*



(b) *Ampere*

Figure 11: Tensor scheduling behaviour on different GPUs. Warps are represented by single lines, and blocks are represented by colour.

these microbenchmarks to cover the whole device. In addition, we observe the same sawtooth pattern observed in section 8.2, with the same root cause: load imbalance.

9.3 Scheduling

We finally investigate the scheduling behaviour of tensor workloads. For brevity, we limit these experiments to half-precision tensors of size $16\times16\times16$, with half-precision accumulators. We use 32 warps per block, which implies 2^{18} floating point operations per tensor operation per block. As with the previous tensor core experiments (section 9.2), we execute 10^7 tensor operations in sequence, resulting in a total of 2.62×10^{12} individual floating point operations. Given the expected tensor core performance of 1024 ops/cycle on *Turing* [21] and 2048 ops/cycle on *Ampere* [23], we expect the total span per block to be 2.56×10^9 and 1.28×10^9 cycles, respectively.

Figure 11 confirms the expected behaviour for the *Turing* microarchitecture: the cycle count between the start of the first warp in each block and the end of the last warp in each block is equal to the aforementioned span predictions. In addition, the blocks are scheduled strictly sequentially. This is not the case on the *Ampere* device, however, where multiple blocks are executed concurrently, even if the tensor core resources are not available. On both microarchitectures, there is a measurable and consistent discrepancy between the times at which warps within one block finish execution, indicating a predictable scheduling bias towards certain warps.

10 DISCUSSION

Given the successful design and deployment of our microbenchmarking suite on the most recent NVIDIA microarchitectures, we reflect in this section on our main findings, the few limitations of our current suite, and discuss portability to other architectures.

10.1 Main findings

This section highlights the differences we found, using our benchmarks, between the three architectures showcased in this work.

10.1.1 The memory subsystem. The way these architectures handle contention in the memory subsystem has improved over time, with *Ampere* exhibiting far lower variance in warp runtime than *Pascal*. We also find that the *Turing* and *Ampere* SMs experience contention with as few as two blocks, while *Pascal* is able to execute two blocks without interference (see section 6).

We also find that *Pascal* requires two blocks to saturate a multiprocessor's bandwidth, while the more recent architectures achieve peak bandwidth with a single block (see section 7). This is further reinforced by the results from our scheduling experiments, which show that the *Pascal* scheduler will always schedule two blocks to run concurrently. Additionally, we found that fine-tuning the caching policies (by means of PTX and caching operators) can have performance implication, mostly uniform between the three GPUs.

10.1.2 The arithmetic subsystem. In terms of arithmetic performance, we find that the choice of data type has a significant impact on computation throughput, and this impact differs per architecture. As expected (based on the GPU specifications), *Ampere* performs far better than the other two architectures for FP64 computation. For all architectures, integer performance is noticeably higher than floating point performance (see section 8). Arithmetic performance scales in a predictable fashion with the number of blocks started. The scheduling behaviour for arithmetic kernels is similar to the scheduling for memory-bound ones.

Finally, tensor cores provide very high performance for matrix operations, and the observed performance is very close to the theoretical peak performance. As with traditional arithmetic cores, the choice of data type impacts throughput significantly (see section 9).

10.2 Portability

In this work, we focus on NVIDIA microarchitectures, and we have shown that the microbenchmarks we propose are effective at isolating architectural features on three most recent such GPUs. However, the GPGPU architecture space is larger than this: a couple more hardware vendors produce GPGPU hardware, and it is unlikely that microarchitectural development by NVIDIA will cease in the near future. Therefore, the portability of our methodology to future NVIDIA microarchitectures and/or to microarchitectures designed by other vendors is relevant if these microbenchmarks are to be adopted as tools in broader performance engineering workflows.

10.2.1 Future NVIDIA GPUs. We expect the microbenchmarks presented in this work will remain usable without modification to NVIDIA GPUs for several generations. This is a result of our high-level design: the large majority of the code is written using CUDA, which is likely to remain the de-facto programming model for NVIDIA GPUs for the near future. The few PTX code segments we have used (a few tens of lines)[1] might require minor updates if PTX will not remain backwards compatible; however, there are no indications that PTX will be deprecated in the near future.

However, besides the actual usability, there remains the question of relevance: while the microbenchmarks proposed are likely to provide insight into the performance of those features on future architectures, it is not certain that those features will remain relevant. It is difficult to predict which direction GPGPU development will take in the coming years. The development of microarchitectures is largely at the behest of the workloads that they will run. For example, *Tensor Cores* were introduced by NVIDIA to support the astronomic rise of machine learning applications. It is likely that future microarchitectures will require us to design and implement additional microbenchmarks for features which we cannot yet predict. However, we believe that the idea of extending microbenchmarks to exploit massive parallelism of the GPU, as well as timing features on a compute unit level, will remain not only possible, but necessary, for the design of these future microbenchmarks.

[1]Source code available at https://zenodo.org/record/5788530.

10.2.2 Beyond NVIDIA GPUs. Porting these microbenchmarks to other vendors' architectures families poses significant challenges: not only are those devices not generally programmed using CUDA, but they are also not guaranteed to share the same features. Tensor cores, for example, are present on NVIDIA GPUs only, and as such tensor core microbenchmarks are not portable to non-NVIDIA devices. Similarly, the cache operators we investigated are a PTX feature, an instruction set architecture mainly supported by NVIDIA devices. The microbenchmarks for memory bandwidth, latency, and arithmetic performance are more generic on a conceptual level, but may still be challenging to port. Indeed, while most GPUs have similar memory and compute subsystems *in theory*, the actual measurements *in practice* rely strongly on a specific model of concurrency, and PTX-specific features.

10.3 Limitations

The primary limitation of the work presented in this paper is the lack of a framework for consistent interpretation of the results. Unlike GPGPU benchmarks, like Rodinia [5] or SPECAccel [14], our microbenchmarks do not return metrics that can directly be used to compare devices. Rather, the data must be always be separately interpreted, which can be a somewhat error-prone process. This is a direct consequence of the nature of our microbenchmarks: their goal is to *identify* discrepancies between microarchitectures. We consider the *quantification* of these discrepancies to be an open problem to which we do not currently have a solution. Regardless, some of our benchmarks provide metrics such as latency and throughput that can be used to compare different implementations of the same architectural feature.

As a direct consequence of the "manual" interpretation of our results, it is difficult to provide generic guidelines to link our findings to direct design recommendations for real-world applications. Our microbenchmarks are, by design, synthetic, in order to isolate features to the greatest possible extent; thus, they are different from most real-world applications where performance is influenced by the interplay between many parts of the microarchitecture. Although we can make specific recommendations based on specific observations, additional work is required to generalise them into guidelines for high-performing implementations of specific GPU applications. Nevertheless, our findings provide significant insight into the behaviour of modern NVIDIA GPGPUs, which in turn can (and should!) impact the *design* of new GPGPU algorithms.

Finally, this paper does not provide a rigid and reproducible methodology for constructing feature-isolating benchmarks. Each of the microbenchmarks presented in this paper was constructed manually, inspired by previous work and vendor documentation [12, 13, 16, 24]. This process is currently insufficiently well-defined to allow us to automate it for future microbenchmarks, but this paper and the open-source code provide sufficient information for these microbenchmarks to be *manually* extended to other features and, potentially, other architectures.

11 CONCLUSION AND FUTURE WORK

General purpose GPU computing has seen accelerated adoption in many scientific domains. As such, many compute-specific features – deeper memory hierarchies, tensor cores, mixed precision – have appeared in newer GPUs. While many of these features are rooted in one application domain or another (e.g., training neural networks or physics simulations), their actual performance impact is difficult to assess from simple specifications.

In this work, we propose an in-depth analysis of such features through microbenchmarking. Specifically, we propose a suite of microbenchmarks that assess the performance of specific GPU features in the memory and arithmetic subsystems. Our microbenchmarks capture the massively parallel nature of GPUs beyond existing work, thus enabling us to uncover their behaviour beyond a single SM. Furthermore, the proposed detailed timing mechanisms enable insight into the effect of thread and block scheduling (which are microarchitecture features) on performance.

We use our microbenchmarks to compare three different NVIDIA GPU microarchitectures: *Pascal*, *Turing*, and *Ampere*. Our main findings include specific scheduling details on Pascal, detailed performance data for the tensor cores on Turing and Ampere, and in-depth insight into the benefits of caching operators.

Although our microbenchmarks have been tested only on these three architectures, we are confident they can be run on more NVIDIA GPUs. However, extensions to other features and architectures, which need to be manually amended, are an important research topic for our future work. Additionally, we also plan to capture our findings into general guidelines for application developers, which requires introspection into several relevant case-studies.

REFERENCES

[1] Tanya Amert, Nathan Otterness, Ming Yang, James H. Anderson, and F. Donelson Smith. 2017. GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed. In *2017 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, Paris, France, 104–115. https://doi.org/10.1109/rtss.2017.00017

[2] Rob Armstrong, Arthy Sundaram, and Fred Oh. 2021. *Revealing New Features in the CUDA 11.5 Toolkit*. NVIDIA corporation. https://developer.nvidia.com/blog/revealing-new-features-in-the-cuda-11-5-toolkit/

[3] Henri Bal, Raoul Bhoedjang, Rutger Hofman, Ceriel Jacobs, Thilo Kielmann, Jason Maassen, Rob Van Nieuwpoort, John Romein, Luc Renambot, Tim Rühl, et al. 2000. The distributed ASCI supercomputer project. *ACM SIGOPS Operating Systems Review* 34, 4 (2000), 76–96.

[4] Nicola Bombieri, Federico Busato, Franco Fummi, and Michele Scala. 2016. MIPP: A microbenchmark suite for performance, power, and energy consumption characterization of GPU architectures. In *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*. IEEE, Krakow, Poland, 1–6. https://doi.org/10.1109/SIES.2016.7509423

[5] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, Austin, Texas, United States of America, 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[6] Stefano Cherubin and Giovanni Agosta. 2020. Tools for Reduced Precision Computation. *Comput. Surveys* 53, 2 (jul 2020), 1–35. https://doi.org/10.1145/3381039

[7] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. 2010. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units* (Pittsburgh, Pennsylvania, USA) *(GPGPU-3)*. Association for Computing Machinery, New York, NY, USA, 63–74. https://doi.org/10.1145/1735688.1735702

[8] Tom Deakin, James Price, Matt Martineau, and Simon McIntosh-Smith. 2016. GPU-STREAM v2.0: Benchmarking the Achievable Memory Bandwidth of Many-Core Processors Across Diverse Parallel Programming Models. In *High Performance Computing*, Michela Taufer, Bernd Mohr, and Julian M. Kunkel (Eds.). Springer International Publishing, Cham, 489–507.

[9] Michael Garland. 2010. Parallel computing with CUDA. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, Atlanta, Georgia, United States of America, 1. https://doi.org/10.1109/ipdps.2010.5470378

[10] Mark Harris. 2016. Mixed-Precision Programming with CUDA 8. https://developer.nvidia.com/blog/mixed-precision-programming-cuda-8/

[11] Kenneth Hoste and Lieven Eeckhout. 2006. Comparing Benchmarks Using Key Microarchitecture-Independent Characteristics. In *2006 IEEE International Symposium on Workload Characterization*. IEEE, San Jose, California, United States of America, 83–92. https://doi.org/10.1109/iiswc.2006.302732

[12] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. 2019. Dissecting the NVidia Turing T4 GPU via Microbenchmarking. arXiv:1903.07486 [cs.DC]

[13] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P. Scarpazza. 2018. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. arXiv:1804.06826 [cs.DC]

[14] Guido Juckeland, William Brantley, Sunita Chandrasekaran, Barbara Chapman, Shuai Che, Mathew Colgrove, Huiyu Feng, Alexander Grund, Robert Henschel, Wen-Mei W. Hwu, Huian Li, Matthias S. Müller, Wolfgang E. Nagel, Maxim Perminov, Pavel Shelepugin, Kevin Skadron, John Stratton, Alexey Titov, Ke Wang, Matthijs van Waveren, Brian Whitney, Sandra Wienke, Rengan Xu, and Kalyan Kumaran. 2015. SPEC ACCEL: A Standard Application Suite for Measuring Hardware Accelerator Performance. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, Stephen A. Jarvis, Steven A. Wright, and Simon D. Hammond (Eds.). Springer International Publishing, Cham, 46–67.

[15] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. 2018. NVIDIA Tensor Core Programmability, Performance & Precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, Vancouver, British Columbia, Canada, 522–531. https://doi.org/10.1109/ipdpsw.2018.00091

[16] Xinxin Mei and Xiaowen Chu. 2017. Dissecting GPU Memory Hierarchy Through Microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems* 28, 1 (jan 2017), 72–86. https://doi.org/10.1109/tpds.2016.2549523

[17] Xinxin Mei, Kaiyong Zhao, Chengjian Liu, and Xiaowen Chu. 2014. Benchmarking the Memory Hierarchy of Modern GPUs. In *2014 IFIP International Conference on Network and Parallel Computing*. Springer Berlin Heidelberg, Yilan, Taiwan, 144–156. https://doi.org/10.1007/978-3-662-44917-2_13

[18] NVIDIA corporation. 2016. *Geforce GTX 1080 Whitepaper*. NVIDIA corporation. http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf

[19] NVIDIA corporation. 2016. *Nvidia Pascal Tuning Guide*. NVIDIA corporation. https://docs.nvidia.com/cuda/pascal-tuning-guide/index.html

[20] NVIDIA corporation. 2016. *Nvidia Tesla P100 Whitepaper*. NVIDIA corporation. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

[21] NVIDIA corporation. 2018. *Nvidia Turing GPU Architecture*. NVIDIA corporation. https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[22] NVIDIA corporation. 2020. *Nvidia A100 Tensor Core GPU Architecture*. NVIDIA corporation. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[23] NVIDIA corporation. 2020. *Nvidia Ampere GA102 GPU Architecture*. NVIDIA corporation. https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf

[24] NVIDIA corporation. 2021. *CUDA C++ Programming Guide*. NVIDIA corporation. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[25] NVIDIA corporation. 2021. *Parallel Thread Execution ISA Version 7.3*. NVIDIA corporation. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html

[26] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. 2007. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum* 26, 1 (mar 2007), 80–113. https://doi.org/10.1111/j.1467-8659.2007.01012.x

[27] Rico van Stigt. 2021. *Exposing GPU Architecture Characteristics using Microbenchmarking*. Master's thesis. University of Amsterdam.

[28] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. 2010. Demystifying GPU microarchitecture through microbenchmarking. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*. IEEE, White Plains, New York, United States of America, 235–246. https://doi.org/10.1109/ISPASS.2010.5452013

[29] Zhen Xu, Xuhao Chen, Jie Shen, Yang Zhang, Cheng Chen, and Canqun Yang. 2019. GARDENIA: A Graph Processing Benchmark Suite for Next-Generation Accelerators. *ACM Journal on Emerging Technologies in Computing Systems* 15, 1 (2019), 1–13. https://doi.org/10.1145/3283450