Demystifying the Nvidia Ampere Architecture through Microbenchmarking and Instruction-level Analysis

Hamdy Abdelkhalik
Klipsch School of ECE
New Mexico State University
enghamdy@nmsu.edu

Yehia Arafa*
Klipsch School of ECE
New Mexico State University
yarafa@nmsu.edu

Nandakishore Santhi
Information Sciences Group
Los Alamos National Laboratory
nsanthi@lanl.gov

Abdel-Hameed A. Badawy
Klipsch School of ECE
New Mexico State University
badawy@nmsu.edu

Abstract—Graphics Processing Units (GPUs) are now considered the leading hardware to accelerate general-purpose workloads such as AI, data analytics, and HPC. Over the last decade, researchers have focused on demystifying and evaluating the microarchitecture features of various GPU architectures beyond what vendors reveal. This line of work is necessary to understand the hardware better and build more efficient workloads and applications. Many works have studied the recent Nvidia architectures, such as Volta and Turing, comparing them to their successor, Ampere. However, some microarchitecture features, such as the clock cycles for the different instructions, have not been extensively studied for the Ampere architecture.

In this paper, we study the clock cycles per instruction with various data types found in the instruction-set architecture (ISA) of Nvidia GPUs. We measure the clock cycles for PTX ISA instructions and their SASS ISA counterparts using microbenchmarks. We further calculate the clock cycles needed to access each memory unit. Moreover, we demystify the new version of the tensor core unit found in the Ampere architecture using the WMMA API and measuring its clock cycles per instruction and throughput for the different data types and input shapes. Our results should guide software developers and hardware architects. Furthermore, the clock cycles per instructions are widely used by performance modeling simulators and tools to model and predict the performance of the hardware.

Index Terms—Instructions Latency, Tensor core throughput, PTX, SASS, Ampere.

I. INTRODUCTION

GPUs significantly accelerate general-purpose applications from neural networks to scientific computing. GPUs have become the main hardware component in any high-performance supercomputer. For instance, Meta built one of the fastest supercomputers based on Nvidia Ampere architecture GPUs (A100) [1], and they are extending it to be the most powerful supercomputer in the world by mid-2022. Besides, tens of the top500 supercomputers [2] are GPU-accelerated.

Nvidia provides a new architecture generation with updated features every two years with little micro-architecture information about these features, making it difficult to quantify. This raises the need to study the effect of new features on the performance of applications. Nvidia introduced the tensor core (TC) unit to accelerate deep neural networks with the introduction of Volta. This version of TC operates on FP16 and FP32 precision operations. Ampere architectures added a sparsity feature and new data precisions for the TC such as Int8, Int4, FP64, bf16, and TF32. Usually, there is little information beyond what the vendors choose to reveal in their whitepapers, which raises the need to quantify these features. Thus, researchers try to explain and demystify the new features of each GPU generation [3]–[8]. However, some areas still have not been fully covered in the literature.

In this work, we focus on demystifying the clock cycle latency at the granularity of the instructions in the Instruction-set architecture (ISA). Similar work has been proposed before. For instance, the authors in [9] adapted some microbenchmarks to demystify some hardware units such as the memory, the tensor cores, and the arithmetic units. However, they only calculated the latency for memories at the granularity of the warp and the block, not the instructions. In another work [10], the authors calculated the latencies for only the memory hierarchy of older generations.

This paper presents microbenchmark analyses to dissect the instruction clock cycles for the instructions of the Nvidia Ampere GPU architecture [11]. The microbenchmarks presented in this work are based on Parallel Thread Execution (PTX) [12]. PTX is an intermediate representation between the high-level language (CUDA) and the assembly language (SASS). Thus, it is portable among different architectures. PTX is open-source and well documented. However, its instructions do not directly execute on the hardware. It has to be converted to another architecture-dependent ISA. SASS, in this case, is closed-source and compatible only within each architecture family. This paper shows how each PTX instruction is mapped to a SASS instruction while measuring the clock cycles for both ISAs. Furthermore, we present the clock cycles needed to access each memory unit. The microbenchmarks are based on a previous work by Arafa et al. [13], which calculated the clock cycle latencies for various instructions on different Nvidia architectures. However, no such studies exist for the Ampere architecture. We also show the clock cycles and throughput for tensor core instructions on different data types.

Measuring the instructions clock cycles helps predict performance by GPU modeling tools. For instance, Arafa et al. [14] showed that the performance model can improve its predic-
tion accuracy by adopting the correct latencies for the GPU instructions. Furthermore, Andersch et al. [10] have proven the critical relationship between latencies and performance. This work is the first step in accurately modeling the Ampere GPU architecture.

The main contributions of this paper are as follows:

- We demystify the Nvidia Ampere [11] GPU architecture through microbenchmarking by measuring the clock cycles latency per instruction on different data types.
- We show the mapping of PTX instructions to SASS instructions while measuring the clock cycles for both.
- We calculate the Ampere architecture tensor core instructions (WMMA) clock cycles latency and throughput while clarifying their PTX and SASS instructions.
- We measure the access latency of the different memory units in the Ampere architecture.

II. BACKGROUND

Unlike multicore and manycore CPUs with powerful cores, GPUs have tens of simple cores/processors that can work simultaneously to perform a specific task efficiently. This is viable for applications requiring more parallelism, such as artificial intelligence and scientific computing.

An Nvidia GPU consists of several streaming multiprocessors (SMs). The number of SMs varies with the GPU generation. Older architectures, such as Kepler, have fewer SMs (15 or 24), while contemporary architectures, such as Ampere, have a more significant number (124). The computation resources inside the SMs also vary depending on the architecture generation. Each SM is divided into hundreds of small cores performing different operations. GPUs have different types of memory units. The global memory and L2 cache are shared with all SMs. Furthermore, it has L1 caches, which are private to each SM. Moreover, threads inside a block can communicate through the shared memory.

Nowadays, the need for GPUs in many essential fields forces vendors to enhance their GPU architecture to provide better performance. NVIDIA provides a new architecture every two years. New architectures not only have new hardware units but also may contain changes in the ISA boosting performance. For example, in the Ampere architecture, Nvidia introduced enhancements to the tensor cores, making them faster and running on larger matrices. Moreover, it introduced the new L2 cache residency control feature, which automatically manages data to keep or evict from the cache.

Although these features are well documented in the architecture whitepaper and online review websites, there are little information on the microarchitecture and the instruction-level enhancements of the recent Ampere architecture. This paper fills this gap by providing a detailed instruction-level characterization of the Ampere GPU instruction set architecture (ISA).

III. RELATED WORK

Researchers have conducted work to dissect every undisclosed microarchitecture characteristic of various GPUs [8], [13], [15–17] using microbenchmarks. Unlike the Nvidia Ampere architecture, the older architectures such as Fermi, Kepler, Turing, and Volta are heavily studied in the literature. Some work focused on the instruction level [6], [13], while others focused on the hardware units themselves [16], [18], [19]. In this section, we present some of this work in more detail.

Wong et al. [6] were the first to introduce microbenchmarks to measure the latency and the throughput of different types of memories and instructions. In [9], the authors modified some microbenchmarks to isolate the GPU features and study each separately. They studied the effect of the number of warps and blocks on the throughput for the memory, arithmetic, and tensor core operations. They calculated the latencies per block, not per instruction. Other works [7], [8], [20] calculated instructions and memory throughput and latencies for the Kepler, Volta, and Turing architectures. However, Bombieri et al. [20] added more details about energy consumption. In the same spirit, Mei et al. [16] presented a microbenchmark for calculating the throughput and latencies of different types of memory units on older architectures such as Fermi, Kepler, and Maxwell.

Other researchers focused on profiling the tensor cores in Volta and Turing architectures [18], [21], [22]. Recently, Sun et al. [17] tried to dissect the tensor cores in Ampere. The authors focused on investigating the matrix multiply-accumulate operations using the MMA API, which execute on the tensor cores. They did not provide results for the WMMA API. Kothiya et al. [23] demonstrated the mapping of PTX to SASS instructions for tensor core operations. Fasi et al. [19] developed a microbenchmark to investigate the tensor core numerical behavior and proved that the tensor core supports the subnormal number.

While the previous work presents good progress, none focused on the clock cycles latency for all data types of each instruction while demonstrating the PTX and SASS mapping for each instruction. Moreover, to the best of our knowledge, we are the first to investigate every WMMA Tensor Core instruction clock cycles and throughput with different data types for the Nvidia Ampere architecture. Finally, Our work can be easily extended for future architectures.

IV. METHODOLOGY

In this section, we introduce the design details of the microbenchmarks. Our work extends the microbenchmarks presented in [13] to calculate the clock cycles per instruction for the Nvidia Ampere A100 GPU. We modified the microbenchmarks to calculate the latencies for dependent and independent instructions. Furthermore, we extended the code to calculate the clock cycles latency for the different types of memory units and the tensor core instructions.

The microbenchmarks are directly written in PTX, a pseudo assembly intermediate and architecture-independent ISA across all Nvidia architectures. However, writing directly in PTX ISA can be tricky since the compiler translates the PTX code into another architecture-dependent ISA, SASS. Nvidia
does not disclose how the compiler maps PTX to SASS. For instance, the compiler can optimize multiple PTX instructions into one SASS instruction. In order to overcome these limitations and ensure the proper instructions get executed as we want, we dynamically read the SASS instruction trace at runtime for each PTX microbenchmark written. We use the **Tracing Tool** from **PPT-GPU** [24] for this purpose.

A. Instructions Clock Cycles Latency

We used only one thread per block to measure the instruction latency. There are two steps for this. **First**, we run code that calculates the clock cycles for the studied instruction with a specified data type. For instance, the code shown in Figure 1 calculates the latency of the add instruction where the operands are 32-bit registers. In general, measuring the latency can be done at runtime instructions and find that it is two cycles.

Finally, we return the latency value to the main CUDA function and divide it by three to calculate the number of cycles for each instruction. We use three instructions to overcome the first launch overhead. We found that executing only one instruction results in an unexpectedly higher number of cycles. Table I shows an example for this purpose.

| Instructions | Clock Cycles |
|--------------|--------------|
| add          | 3            |
| add          | 3            |
| add          | 3            |

Second, we inspect the SASS instructions using the dynamic **Tracing Tool** from **PPT-GPU** [14] to ensure that the mapping from PTX to SASS is correct and no additional overhead or instructions are added at runtime by the compiler. The PTX code shown in Figure 4(a) provides an inaccurate latency for the add instruction when storing the clock in 32-bit registers. The dynamic SASS instruction trace shows a barrier between the two clock readings, as shown in the second instruction of the SASS part. This barrier causes a considerable change in the results (around 33 cycles more). One method to overcome this barrier is to use the 64-bit registers to store the clocks, which removes the barrier and provides accurate measurement, as shown in Figure 4(b). The CPI for the first and second cases are 13 and 2 cycles, respectively. Finally, we calculate the clock overhead using two consecutive clock reading instructions and find that it is two cycles.
B. Memory Units Access Latencies

To calculate global, L2, and L1 cache memories latencies, we use a pointer chasing technique, in which each array element is dependent on the previous ones. This technique forces the reading operations to serialize to calculate the latency correctly. Otherwise, many reading operations can be issued simultaneously, which makes the latency measurement inaccurate. Figure 2 shows the PTX microbenchmark for memory latency calculation. Line 1 moves the array address to register %r19. Then, we start a counter with a zero value in register %r40. This counter is used to iterate over the array of elements. Lines 3 and 9-11 represent the loop instructions. Lines 4-7 store the array elements where each element depends on the previous one. After storing the results, we use the instructions from lines 14-24 to read the clocks while reading every element in the array. From 16-19, we have four load instructions to load four values, which repeat to read all the array elements.

The _ld_ instruction can be used with many operators such as _cv_, _ca_, and _cg_. Each operator has its usage. _ca_ is used to cache on all available levels (global-L1-L2) while the _cg_ caches only in L2. On the contrary, we use _cv_ because it bypasses the caches, which we need when we calculate the global memory latency. We use four instructions because we found that in many cases, the compiler unrolls the loops by four when we inspected the dynamic trace of some Cuda applications that use loops. The difference between the global memory and the L2 cache code is the operator used with the _ld_ instruction in the array and the number of elements in the array. For the L2 cache, we use the _cg_ operator, and the total size of the array elements must be less than the L2 size, while for the global memory, it must be larger than the L2 cache to avoid residency in the L2 cache. Likewise, we repeat the same methodology with the _ca_ operator to calculate the L1 cache latency.

For the **shared memory**, we load and store instructions between reading the clocks, as shown in lines 3-12 of Figure 3. However, we need to add another instruction that depends on the _ld_ or _st_ instructions to prevent the compiler from executing the clock reading instruction before finishing, as shown in lines 4-13.

C. Tensor core Instructions Latency and Throughput

The Tensor core (TC) unit is used mainly to accelerate machine learning applications. Each SM in the Ampere architecture contains four tensor cores that can run multiply-add operations on three matrices in the form \(D=A*B+C\) where A and B are the inputs and C and D are the accumulators. Unlike Volta, which supports only _fp16_ precision for the inputs, the Ampere architecture supports many types, such as _fp16_, _bf16_, _tf32_, _f64_, _u8_, _u4_, and _single bits_. General arithmetic instructions use one thread to execute and can communicate only through the global or shared memory. On the other hand, the TC instructions use all 32 instructions in the dedicated machine learning applications. Each SM in the Ampere architecture contains four tensor cores that can run multiply-add operations on three matrices in the form \(D=A*B+C\) where A and B are the inputs and C and D are the accumulators. Unlike Volta, which supports only _fp16_ precision for the inputs, the Ampere architecture supports many types, such as _fp16_, _bf16_, _tf32_, _f64_, _u8_, _u4_, and _single bits_. General arithmetic instructions use one thread to execute and can communicate only through the global or shared memory. On the other hand, the TC instructions use all 32 instructions in the dedicated warp. To demystify the Ampere TC instructions with the new data types, we designed a special microbenchmark written in Cuda. The microbenchmark is inspired by the work of Jia et al. [7], which focused on Volta architecture.

Some of the new data types introduced in the Ampere architecture are still in the experimental stage, as mentioned in PTX and CUDA documentation [25]. Moreover, because each data type has its shapes, stride, and layout, we use a different function to calculate the latency of each type. Figure 5 shows the code used to calculate the TC instruction latency of _U8_ data type. Lines 5-7 create a fragment in which the registers are prepared to get the matrix elements to be stored. We create four fragments; however, we do not...
write all to make the shape smaller. Then, we load the data from memory (lines 10-12), and the same goes for the other fragments. As explained previously, we read the clock before and after the TC WMMA executes (lines 15 to 22) and subtract before printing, as shown in lines 28-29. In lines 16-21, we run four TC instructions (1 per TC) several times. We used four instructions because calculating the latency from one TC with one instruction provides inaccurate measurement. For example, Figure 6 shows the dynamic SASS instructions of running one instruction on one TC. The NOP instructions refer to a warp synchronization in PTX, and we found that the latency here is not the same as mentioned in the white paper. This also happens when we run one instruction several times. Finally, we calculate the latency per instruction and print them in lines 28-29. Similarly, we calculate the TC WMMA throughput.

V. Results

In this section, we present our detailed setup and experimental results. We first show the instructions clock cycle latencies. Next, we explain the memory access latencies. Finally, We present the Tensor Core latency and throughput. We run all the microbenchmarks on the Nvidia Ampere A100 GPU.

A. Instructions Latency

We found that dependencies directly affect the instructions clock cycle latency. Hence, we rerun the microbenchmark with a sequence of dependent instructions (as in Figure 1), replacing the dependent sequence with another sequence of independent instructions. Table II shows the CPI for dependent and independent sequences for some of the instructions. For instance, single precision add instruction shows 4 and 2 cycles, respectively. We also found that with no dependency, the three add.u32 instructions of Figure 1 are mapped to the same SASS instruction (IADD) as in Figure 4(a). Nevertheless, PTX instructions may be converted to different instructions when we use three dependent instructions. For example, the add.u32 PTX instruction can be mapped to IADD3 or IMAD.IADD with the dependency case.

Table V depicts the various PTX-SASS instructions with their measured clock cycles latencies. We have a separate PTX kernel (microbenchmark) for each field in the table.

Next, we discuss some additional insights we found while generating the results:

1) The mad instruction runs on the floating point pipeline, not the integer pipeline, even if we use it with integer values. This can be proven by the following:
   - The PTX instruction mad.lo.u32 in Table V is mapped to the SASS FFMA (floating multiple-add).
   - We created a special code that runs two add instructions and two mad instructions using one thread, and we found that the total number of cycles is around four cycles. This does not mean that each instruction takes one cycle because according to the results in Table V, the minimum number of cycles for each of them individually is two cycles. Actually, it proves that each of the two types is executed simultaneously on different pipelines.

2) Except for bfind, min, and max instructions, there is no difference in clock cycles or mapping from PTX to SASS when using signed or unsigned instructions. For instance, add.u64 and add.s64 provide the same mapping and same latency.

3) Usually, a mov or add instruction is used to initialize a register with a value before using this register as an input operand to the instruction that we need to calculate its latency. However, in some cases, we found that the clock cycles and the PTX-SASS mapping change depending on how the inputs are initialized. For example, the PTX neg.f32 instruction is mapped to the SASS FADD instruction when we use “add” to initialize the inputs. On the other hand, it merges the mov and the neg instructions together in one SASS instruction (IMAD.MOV.u32) when we use mov for initializing. The same happens for the abs.f32 instruction.

4) Although many PTX instructions have a one-to-one mapping to SASS, others such as div, rem, sin, and cosf are translated to multiple different SASS instructions.

5) Not all instructions with the same data type have the same latency. More specifically, mad.lo.u64 is mapped to an IMAD SASS instruction which takes only 2 cycles. However, the double precision add, sub, & fma instructions take four cycles each.

6) For the testp instruction, the latency depends on the state.

B. Memory Access Latencies

The observed latencies of the different types of memories are shown in Table IV. The global memory latency is around 290 cycles. This value does not include any caching effects because we prevent caching at all levels. This number is improved compared to Turing architecture which was 434 cycles [13]. The L2 access latency is 200 cycles compared to 188 cycles for Turing architecture. Furthermore, the L1 cache hit for both Ampere and Truning architectures is 33 and 32 cycles, respectively. For the shared memory, we found that a store latency is smaller than a load instruction, 23 and 19 for load and store, respectively.

| # instrs | CPI for dependent | CPI for independent |
|---------|------------------|---------------------|
| add.f16 | 3                | 2                   |
| add.u32 | 4                | 2                   |
| add.f64 | 5                | 4                   |
| mul.lo.u32 | 3       | 2                   |
| mad.mn.f32 | 4        | 2                   |

TABLE II
THE CPI FOR DEPENDENT AND INDEPENDENT INSTRUCTIONS
C. Tensor Core Latencies and Throughput

The Ampere architecture ISA provides various SASS instructions that run on the Tensor Core, which supports the newly added data types. The Volta Architecture ISA has only the HMMA.884 SASS instruction, used for all Tensor Core operations (single and mixed-precision operations). For Turing, two kinds of the HMMA SASS instructions exist which run on different input shapes, HMMA.1688 and HMMA.884 [22].

Table III depicts the Ampere architecture TC instructions. More specifically, DMMA.884, IMMA.16816 and IMMA.8832 were added to handle the FP64, U8 and U4 data types, respectively. Each PTX instruction of each data type is translated to a different number of SASS instructions. For the FP16, BF16, and U8 inputs, the PTX is translated into two instructions. The TensorFloat-32 (TF32) precision is mapped to four SASS instructions, while the FP64 and U4 are mapped to only one instruction. These differences are related to the difference between the supported PTX shapes and the shapes that the SASS can work on. For example, in the first row of Table III, the PTX instruction can use many shapes such as 16x16x16, but the SASS can only work on 16x8x16. Therefore, two SASS instructions are needed to iterate over the PTX shape. However, the physical TC implementation can perform 8x4x8 [17]. While Raihan et al. [21] note that the TC latencies are shape-dependent for Turing. We found that different shapes for the same data type do not affect the calculated latency. It can vary from one type to another in Ampere architecture. Our observations for the TC throughput and latencies shown in the Table are consistent with the behavior mentioned in the whitepaper [11]. Finally, We noticed that for all half floating precision (fp16 and bf16) inputs, the SASS instruction MOVMM.16.MT88 is used for loading a matrix to the TC. In general, the MOVMM SASS instruction is used to move a matrix with a transpose. The number of issued MOVMM instructions depends on the matrix shape and the layout (row or column major). For example, if we used the A and B matrices as row major in the PTX code, then the MOVMM instructions are used to transpose the B matrix to multiply each row from A with each column from B. However, when we use both as column major, the MOVM instruction is used with the A and C matrices. It transposes A and C before execution and transposes C after the execution. Finally, if A is a row major and B is a column major, the MOVVM instruction does not exist in the trace. We used the above latency calculation methodology to calculate the memories throughput. Our observed values are consistent with the throughput values in the whitepaper.

| Supported shapes                  | Inputs | Accumulators | Cycles | Measured-theoretical | Instructions                                                                 |
|-----------------------------------|--------|--------------|--------|----------------------|-------------------------------------------------------------------------------|
| m16n16k16 - m8n32k16 - m32n8k16   | .f16   | .f16         | 16     | 311-312 GB/s         | PTX: wmma.mma.sync.aligned.row.row.m16n16k16.f16.f16                           |
|                                   |        |              |        |                      | SASS: 4*HMMA.16816.F32.BF16 – each inst. is 8 cycles                        |
| m16n16k16 - m8n32k16 - m32n8k16   | .f16   | .f32         | 16     | 310-312 GB/s         | PTX: wmma.mma.sync.aligned.row.row.m16n16k16.f16.f16                           |
|                                   |        |              |        |                      | SASS: 2*IMMA.8832 – each inst. is 4 cycles                                   |
| m16n16k16 - m8n32k16 - m32n8k16   | .bf16  | .f32         | 16     | 310-312 GB/s         | PTX: wmma.mma.sync.aligned.row.row.m16n16k16.f16.bf16.bf16                   |
|                                   |        |              |        |                      | SASS: 2*HMMA.16816.F32.BF16 – each inst. is 8 cycles                        |
| m16n16k8                          | .f32   | .f32         | 16     | 132-158 GB/s         | PTX: wmma.mma.sync.aligned.row.row.m16n16k8.f32.tf32.tf32                   |
|                                   |        |              |        |                      | SASS: 4*HMMA.16816.F32.TF32 – each inst. is 4 cycles                        |
| m8n8k4                            | .f64   | .f64         | 16     | 19-19.5 GB/s         | PTX: wmma.mma.sync.aligned.row.row.m8n8k4.f64.f64.f64.f64                   |
|                                   |        |              |        |                      | SASS: 1*HMMA.884 – each inst. is 16 cycles                                  |
| m16n16k16 - m32n8k16 - m8n32k16   | .u8    | .u32         | 8      | 594-624 GB/s         | PTX: wmma.mma.sync.aligned.row.row.m16n16k16.u32.u8.u32                     |
|                                   |        |              |        |                      | SASS: 2*IMMA.16816.U8.U8 – each inst. is 4 cycles                           |
| m8n8k32                           | .u4    | .u32         | 4      | 1229-1248 GB/s       | PTX: wmma.mma.sync.aligned.row.col.m8n8k32.x32.x32.x4.x4.x32                 |
|                                   |        |              |        |                      | SASS: 4*IMMA.8832.U4.U4 – each inst. is 4 cycles                            |

VI. Conclusion

This paper demystifies the instructions, memories, and tensor cores for the Nvidia Ampere architecture. We perform a detailed analysis of the PTX instructions latency while showing their SASS translation. The presented microbenchmarks are portable and can be extended for future architectures. In addition, we point out the microarchitecture instructions of the tensor cores and their latencies for all data types supported by the Ampere architecture. Finally, we calculate the memory latency while building a pointer chasing microbenchmark for both global memory and L2 cache. This work can help in understating the hardware from the microarchitecture point of view, leading to better optimized applications and workloads.

Acknowledgement

The authors would like to thank the anonymous reviewers. This work is partially supported by Triad National Security, LLC subcontract #581326. The views expressed in this paper do not necessarily represent the views of the DOE or the U.S. Government. Paper LAUR is LA-UR-22-28504.
| PTX | SAAS | cycles | PTX | SAAS | cycles |
|-----|------|--------|-----|------|--------|
| add.a16 | UIAAD3 | 2 | Min.a16 | ULOP3.LUT+UISETP.U32.AND+USEL | 8 |
| addc.a32 | IADDX.X | 2 | min.a32 | IMINMX.U32 | 2 |
| addc.a32 | IADDX | 2 | min.a64 | UISETP.U32.AND+2*USEL | 8 |
| add.a64 | UIAAD3.x+UIAAD3 | 4 | min.s16 | PRMT+IMINMX | 4 |
| add.a64 | UIAAD3+UIAAD3 | 4 | min.s32 | IMINMX | 2 |
| add.f16 | IADD | 2 | Min.s64 | UISETP.U32.AND+UISETP.LT.EX+2*USEL | 8 |
| add.f16 | FADD | 2 | min.f16 | HMMNX2+PRMT | 4 |
| add.f16 | DADD | 4 | min.f32 | FMNMX | 2 |
| mul instruction | min.f364 | | DSETP.MIN.AND+IMAD.MOV.U32†+UMOV+FSEL | 10 |
| mul.wide.u16 | LOP3.LUT+IMAD | 4 | | Neg instruction | |
| mul.wide.u32 | IMAD | 4 | neg.s16 | UIAAD3+0PRMT | 5 |
| mul.lo.a16 | LOP3.LUT+IMAD | 4 | neg.s32 | IADD | 2 |
| mul.lo.a16 | IMAD | 2 | neg.s64 | IMAD.MOV.U32+HMA2.MMA+MOV+UIAAD3 | 10 |
| mul.lo.u64 | IMAD | 2 | neg.l32 | FADD or IMAD.MOV.U32 | * |
| mul24.lo.a32 | PRMT+IMAD | 3 | neg.f64 | DADD+(UMOV) | 4 |
| mul24.hu.a32 | UPRMT+USHFR.U32.HI+IMAD.U32+PRMT | 9 | | FMA instruction | |
| mul.rn.f16 | HML2 | 2 | fma.rn.f16 | HFM2 | 2 |
| mul.rn.f32 | FMUL | 2 | fma.rn.f32 | FFMA | 2 |
| mul.rn.m64 | DMUL | 4 | fma.rn.m64 | DFMA | 4 |
| MAD instruction | | | | Sqrt Instruction | |
| mad.lo.a16 | LOP3.LUT+IMAD | 4 | sqrt.rn.f32 | [multiple instrs including MUFU.RSQ] | 190-235 |
| mad.lo.a32 | FMMA | 2 | sqrt.approx.f32 | [multiple instrs including MUFU.SQRT] | 2-18 |
| mad.lo.u64 | IMAD | 2 | sqrt.approx.f64 | [multiple instrs including MUFU.RSQ64] | 260-340 |
| mad24.lo.a32 | SGXT.U32 + IMAD | 4 | | Rcp Instruction | |
| mad24.hu.a32 | USHFR.U32.HI+UMINAD.WIDE.U32+2*UPRMT+IADD3 | 11 | rsqrt.approx.f32 | [multiple instrs including MUFU.RSQ] | 2-18 |
| mad.rn.f32 | FMMA | 2 | rsqrt.approx.f64 | 8-11 |
| mad.rn.m64 | DMUL | 4 | | |
| Div / Rem Instruction | | | | Pop Instruction | |
| rem/div.a16/a16 | multiple instructions | 290 | popc.b325 | POPC | 6 |
| rem/div.a32/a32 | multiple instructions | 66 | popc.b64 | 2*UPOPC + IADD3 | 7 |
| rem/div.a64/a64 | multiple instructions | 420 | clz.b32 | FLO.U32 | 7 |
| div.f32 | multiple instructions | 525 | clz.b64 | FLO.U32+U32 | 7 |
| div.f64 | multiple instructions | 426 | | |
| Abs Instruction | | | | Blind Instruction | |
| abs.a16 | PRMT+HABS+PRMT | 4 | bimd.a32 | FLO.U32 | 6 |
| abs.a32 | bimd.a64 | PLO.U32+IŠEPT.P.EUU32+IADD3+IBRA | 164 |
| abs.s64 | UISEPT.LT.AND+UIAAD3.X +UIAAD3+2*USEL | 11 | bimd.s64 | FLO | 6 |
| abs.f16 | PRMT | 1 | bimd.s16 | multiple instructions | 195 |
| abs.f32 | FADD+FTZ | 2 | bimd.s32 | |
| abs.f64 | DADD or (DADD+UMOV) | 4 | testp.normal.f32 | IMAD.MOV.U32+2*UISEPT.GE.U32.AND | 0 or 6 |
| Brev Instruction | | | | testp.normal.f32 | IMAD.MOV.U32+2*UISEPT.P.EEUU32 | 0 or 6 |
| brev.b32 | BREV+SGXT.U32 | 2 | testp.normal.f64 | 2*UISEPT.P.EEUU32+2*UISEPT.P.EEUU32.AND | 13 |
| brev.b64 | BREV+SGXT.U32 | 6 | testp.normal.f64 | 2*UISEPT.P.EEUU32+2*UISEPT.P.EEUU32.AND | 8 |
| copysign Instruction | | | | Other Instruction | |
| copysign.f32 | 2*LOP3.LUT or 1.5*LOP3.LUT | 4 | sin.approx.f32 | FMUL + MUFU.SIN | 8 |
| copysign.f64 | 2*LOP3.LUT+IMAD.U32+2*MOV | 6 | cos.approx.f32 | FLO.RZ+MUFU.COS | 8 |
| copysign.f64 | 2*LOP3.LUT+IMAD.U32+2*MOV | 6 | cos.approx.f32 | FLO.RZ+MUFU.COS | 8 |
| and/or/xor Instruction | lg2.approx.f32 | | FSETP.GEU.AND+FMUL+MUFU.LG2+FADD | 18 |
| and.b16 | LOP3.LUT or 1.5*LOP3.LUT | 2 | ex2.approx.f32 | FSETP.GEU.AND+2*FMUL+MUFUX2 | 18 |
| and.b32 | LOP3.LUT | 2 | ex2.approx.f16 | MUFU.EX.F16 | 6 |
| and.b64 | LOP3.LUT | 2-3 | tank.approx.f16 | MUFU.TANH | 6 |
| Not Instruction | tank.approx.f16 | | MUFU.TANH.F16 | 6 |
| not.b16 | LOP3.LUT | 2 | bar.warp.sync | NOP | changes |
| not.b32 | LOP3.LUT | 2 | fns.b32 | multiple instructions | 79 |
| not.b64 | 2*ULOP3.LUT | 4 | cvt.rz.s32.f32 | PT2.TRUNC.NTZ | 6 |
| lop3 Instruction | setp.nc.s32 | | IŠEPT.P.EEUU32.AND | 10 |
| lop3.b32 | IMAD.MOV.U32+2*LOP3.LUT | 4 | mov.u32 clock | CŠ2.R | 2 |
| bfe Instruction | | | | Bit Instruction | |
| cnot.b16 | ULOP3.LUT+IŠEFPQ.U32.AND+SEL | 5 | bit.b32 | 3*PRMT+2*IMAD.MOV+USHFL.U32+BMSK+LOP3.LUT | 11 |
| cnot.b32 | ULOP3.LUT+IŠEFPQ.U32.AND+SEL | 4 | bit.b64 | UMOV+USHFL.U32+(UIAAD3+ULOP3.LUT)* | 5 |
| cnot.b64 | multiple instructions | 11 | dp4.a.u32.632 | IMAD.MOV.U32+IDP3.A.U8.U8 | 135-170 |
| bfe.a32/a32 | 3*PRMT+2*IMAD.MOV+USHFR.U32.HI+SGXT.U32 | 11 | dp2.a.u32.532 | IMAD.MOV.U32+IDP2.A.U8.U8 | 135-170 |
| bfe.a64 | UMOV+USHFL.U32+UIAAD3+ULOP3.LUT* | 5 | dp2.a.lo.u32.a32 | IMAD.MOV.U32+IDP2.A.LO.U16.U8 | 135-170 |
REFERENCES

[1] Meta AI supercomputer, 2022. [Online]. Available: https://ai.facebook.com/blog/ai-rsc/

[2] Top500 List, 2022. [Online]. Available: https://www.top500.org/lists/top500/2022/6/

[3] X. Zhang, G. Tan, S. Xue, J. Li, K. Zhou, and M. Chen, “Understanding the gpu microarchitecture to achieve bare-metal performance tuning,” in Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2017, pp. 31–43.

[4] M. Ho and W.-F. Wong, “Exploiting half precision arithmetic in nvidia gpus,” in 2017 IEEE High Performance Extreme Computing Conference (HPEC). IEEE, 2017, pp. 1–7.

[5] D. Lustig, S. Sahasrabuddhe, and O. Giroux, “A formal analysis of the nvidia ptx memory consistency model,” in Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 2019, pp. 257–270.

[6] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moschovos, “Demystifying gpu microarchitecture through microbenchmarking,” in 2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS). IEEE, 2010, pp. 235–246.

[7] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, “Dissecting the nvidia volta gpu architecture via microbenchmarking,” arXiv preprint arXiv:1804.00526, 2018.

[8] Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza, “Dissecting the nvidia turing t4 gpu via microbenchmarking,” arXiv preprint arXiv:1903.07486, 2019.

[9] R. van Stigt, S. N. Satwam, and A.-L. Varbanescu, “Isolating gpu architectural features using parallelism-aware microbenchmarks,” in Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering, 2022, pp. 77–88.

[10] M. Andersch, J. Lucas, M. A. LvLvarez-Mesa, and B. Juurlink, “On latency in gpu throughput microarchitectures,” in 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). IEEE, 2015, pp. 169–170.

[11] NVIDIA A100 Tensor Core GPU Architecture, 2022. [Online]. Available: https://docs.nvidia.com/parallel-thread-execution/index.html#/instruction-set

[12] Y. Arafa, A.-H. A. Badawy, G. Chennupati, N. Santhi, and S. Eidenbenz, “Low overhead instruction latency characterization for nvidia gpgpus,” in 2019 IEEE High Performance Extreme Computing Conference (HPEC). IEEE, 2019, pp. 1–8.

[13] ——, “Ppt-gpu: Scalable gpu performance modeling,” IEEE Computer Architecture Letters, vol. 18, no. 1, pp. 55–58, 2019.

[14] Y. Arafa, A. ElWazir, A. ElKanishy, Y. Aly, A. Elsayed, A.-H. Badawy, G. Chennupati, S. Eidenbenz, and N. Santhi, “Verified instruction-level energy consumption measurement for nvidia gpus,” in Proceedings of the 17th ACM International Conference on Computing Frontiers, 2020, pp. 60–70.

[15] X. Mei and X. Chu, “Dissecting gpu memory hierarchy through microbenchmarking,” IEEE Transactions on Parallel and Distributed Systems, vol. 28, no. 1, pp. 72–86, 2016.

[16] W. Sun, A. Li, T. Geng, S. Stuijk, and H. Corporaal, “Dissecting tensor cores via microbenchmarks: Latency, throughput and numerical behaviors,” arXiv preprint arXiv:2206.02874, 2022.

[17] S. Markidis, S. W. Der Chien, E. Laure, I. B. Peng, and J. S. Vetter, “Nvidia tensor core programmability, performance & precision,” in 2018 IEEE International parallel and distributed processing symposium workshops (IPDPSW). IEEE, 2018, pp. 522–531.

[18] M. Fasi, N. J. Higham, M. Mikaitis, and S. Pranesh, “Numerical behavior of nvidia tensor cores,” PeerJ Computer Science, vol. 7, p. e330, 2021.

[19] N. Bomieri, F. Busato, F. Fummi, and M. Scala, “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, 2016, pp. 1–6.

[20] M. A. Raihan, N. Goli, and T. M. Aamodt, “Modeling deep learning accelerator enabled gpus,” in 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). IEEE, 2019, pp. 79–92.