<Undergraduate Thesis: EEE4610-01>

Modeling Tensor Core Microarchitecture for NVIDIA Turing Architecture with Experimental Features

Min-Young Shin
School of Electrical and Electronic Engineering
College of Engineering
Yonsei University

<Undergraduate Thesis: EEE4610-01>

Modeling Tensor Core Microarchitecture for NVIDIA Turing Architecture with Experimental Features

Thesis Advisor: Won Woo Ro

A thesis submitted in a partial fulfillment
for the senior Undergraduate Creative Independent Study's requirements

December 2020

Min-Young Shin
School of Electrical and Electronic Engineering
College of Engineering
Yonsei University

감사의 글

여름방학부터 시작하여 6개월 여간 공부와 연구를 지도해주신 연세대학교 esCaL 연구실의 노원우 교수님과 김민규 조교님께 진심을 다해 감사의 말씀드립니다. 시험기간을 제외하고 한 차례도 빠짐없이 매주 미팅을 한다는 것이 쉽지 않으셨을 텐데, 바쁘신 와중에도 좋은 가르침을 많이 주신 덕분에 무사히 연구를 마무리할 수 있었습니다. 또, 제 사수는 아니셨음에도 불구하고 제가 질문드린 것을 항상 해결해주신 안성우 조교님께도 감사의 말씀 전합니다.

대학에 와서 연구를 하는 것이 처음인 지라 미숙한 부분도 많았고, 스스로 답답했던 것도 많았습니다. 그럼에도 불구하고 항상 격려해주시고 잘한 점만 칭찬해주셨던 교수님과 조교님 덕분에 포기하지 않고 계속할 수 있었습니다. 연구의 힘든 부분도 당연히 많이 느꼈지만, 성취한 뒤의 느낄 수 있는 보람도 누릴 수 있어서 운이 좋았다고 생각합니다.

논문 서칭을 어디서 찾아야하는 지도 모른 채 연구를 시작했지만, 연구를 마무리하는 이 시점에 제 자신을 돌아보니 많이 성장한 것을 느낍니다. 시행착오를 겪으면서 스스로 깨닫게 된 것과 교수님, 조교님이 주신 피드백으로 배운 것들이 앞으로 많은 도움이 될 것입니다. 교수님이 보시기에는 아직 부족한 부분이 많을 테지만, 다음에 또 논문을 쓰게 된다면 이번 연구보다 더 나은 과정과 결과를 보일 수 있을 것 같습니다.

학부생으로써 대학원생 수준의 공부와 실험을 해볼 수 있는 기회가 있어서 정말 재밌었습니다. 이런 기회와 환경을 만들어주신 학교와 교수님, 조교님들께 다시 한번 감사드리며, 코로나 바이러스의 위협으로 힘들었던 2020년을 지나 2021년에는 항상 행복하고 건강한 한 해가 되시길 기원합니다.

Contents

Figure index ·····iii
Table index ·····vi
Abstract ·····vii
1. Introduction ·····1
2. Background ·····3
2.1. CUDA programming model····· ·····3
2.2. Instruction and Register data flow····· ·····5
2.3. Matrix multiplication····· ·····7
2.3.1. Naive matrix multiplication ·····7
2.3.2. tiled matrix multiplication ·····8
2.4. Tensor core····· ·····9
2.4.1. 1st-gen tensor core····· ·····9
2.4.2. WMMA API····· ·····10
2.4.3. PTX analysis····· ·····12
2.5. GPGPU-Sim····· ·····13
3. Microbenchmark····· ·····14
3.1. Fragment distribution··· ·····14
3.2. Computation pattern····· ·····20
3.3. Clock profiling····· ·····22
4. Proposed microarchitecture ····· ·····26
5. Evaluate····· ·····27
5.1. Building GPGPU-Sim ·····33
5.2. Evaluate····· ·····31
6. Conclusion··· ·····32
Reference ·····33
국문요약 ·····34

Figure index

Fig. 1.1. NVIDIA GPU architectures 2
Fig. 2.1. CUDA compilation flow 3
Fig. 2.2 comparison of two codes performing vector addition 4
Fig. 2.3. GPGPU pipeline 5
Fig. 2.4. naive matrix multiplication 6
Fig. 2.5. tiled matrix multiplication 7
Fig. 2.6. diagram for tensor core operation 8
Fig. 2.7. example code for WMMA kernel 11
Fig. 2.8. general PTX code for WMMA API 12
Fig. 3.1. PTX code for second generation tensor core 13
Fig. 3.2. PTX code for INT4 precision mode 14
Fig. 3.3. PTX code for INT8 precision mode 15
Fig. 3.4. PTX code for FP16 precision mode 15
Fig. 3.5. PTX code for mixed precision mode 15
Fig. 3.6. used microbenchmark for fragment distribution 16
Fig. 3.7. fragment distribution for matrix A and B 17
Fig. 3.8. threadgroup information 17
Fig. 3.9. fragment distribution for matrix C 17
Fig. 3.10. thread distribution inside a threadgroup 17
Fig. 3.11. SASS code for wmma::mma instruction 19
Fig. 3.12. computation pattern 20
Fig. 3.13. microbenchmark measuring clock cycle 21
Fig. 4.1. proposed microarchitecture 24
Fig. 5.1. consumed clock cycles 27
Fig. 5.2. comparison of clock cycle according to the precision mode 29

Table index

Table 1. supported precision and tile size·····10
Table 2. average cumulative clock cycle·····22
Table 3. clock interval between set and set·····22

ABSTRACT

Modeling Tensor Core Microarchitecture for NVIDIA Turing Architecture with Experimental Features

Since the launch of NVIDIA Corporation's Volta architecture in 2017, their GPU has included tensor cores which are the specialized unit for the matrix multiply-accumulate operation. The unit has brought drastic speed-up in Deep Neural Network (DNN) train and test time due to it consists mostly of convolution and matrix operations.

In this study, the *microarchitecture* of *Tensor Core* in *Turing architecture* is proposed. Since NVIDIA does not disclose the inside of the tensor core, it is necessary to profile through microbenchmarking. Dissecting the NVIDIA GPUs has also been done in previous studies[1]-[3]. However, it was not revealed about the experimental features of the Turing architecture, i.e. INT4(int 4-bit) operation mode and B1(binary 1-bit) operation mode. All of these functions were analyzed in this study.

Based on the results, the microarchitecture of the 2nd generation tensor core was modeled, and GPGPU-Sim was updated for the API of the 2nd-gen tensor core to evaluate its design. To evaluate the results, we compared the speed of the General Matrix Multiply(GEMM) kernel with newly reconstructed GPGPU-Sim based on the proposed microarchitecture and the actual GPU. Studies have shown that experimental features show efficiency in memory access and speed in calculating GEMM kernel, however the absolute computation amount per time is limited than the others.

Key words : 2nd-gen Tensor Core, microarchitecture, Experimental features, GPGPU-Sim

1. Introduction

NVIDIA's tensor core is one of the most widely used units that can process Deep Neural Network(DNN) with high throughput. The emergence of tensor cores has exponentially increased the speed of the GPU's DNN computation.

The Turing architecture was launched after Volta architecture and before ampere architecture(see fig 1.1). In particular, the second tensor core of the NVIDIA Turing architecture supports more functionality than the previous version. The INT8, INT4, and B1 precision mode with various tile sizes are enabled targeting each appropriate neural network.

The two experimental features, INT4 mode and B1, can be a key role in building neural networks. INT4 precision mode is used for inferencing workloads that can tolerate quantization[4]. B1 precision mode can accelerate Binarized Neural Network(BNN) and tensor core-accelerated BNN design can process ImageNet at a rate of 5.6K images per second, 77% faster than state-of-the-art[5].

In general, NIVIDIA does not disclose its internal structure, i.e. SASS, the real hardware ISA, and microarchitecture for GPU are unknown. However, there are some related studies. Jia, Zhe et al(2018) and Jia, Zhe, et al(2019) show the overall architecture for Volta and Turing architecture via microbenchmarking[2][3]. Raihan et al(2019) was focused on the 1st-gen tensor core and 2nd-gen tensor core microarchitecture without experimental features. Then, they prosed the microarchitecture for the 1st-gen tensor core. By their research, we could understand how GPU works and inspired how experimental features are constructed.

To the best of our knowledge, microarchitecture for 2nd-gen tensor core including experimental features has not been proposed by previous research. GPGPU-Sim, the most widely used GPGPU simulator, is also unable to simulate the 2nd-gen tensor cores API, due to the only API of Volta tensor core is supported.

Therefore, in this study, the following contributions were made.

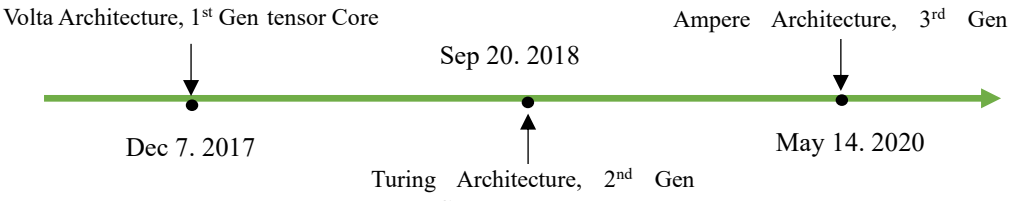


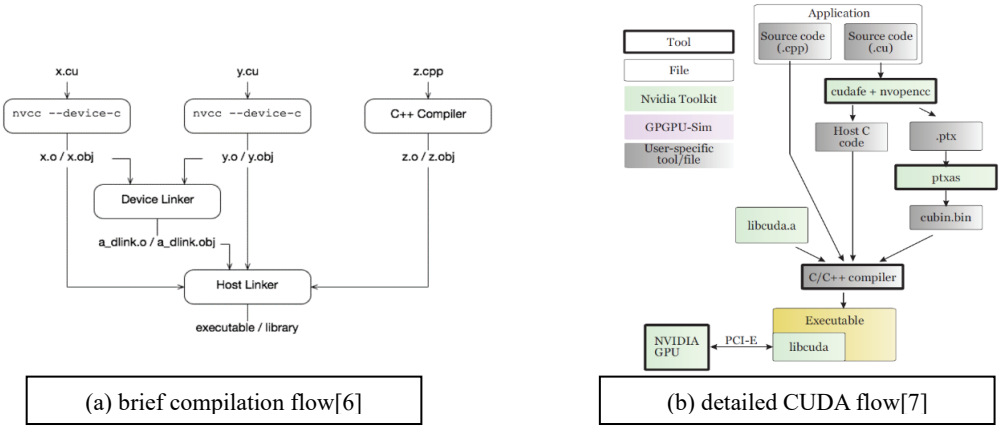
Fig. 1.1. NVIDIA GPU architectures.

1. We figure out the fragment distribution, computation pattern, clock cycle information for INT4 precision mode, and B1 precision mode via microbenchmarking.

2. It is possible to propose the 2nd-gen tensor core's microarchitecture based on the result from *contribution 1*.

3. Applying newly proposed 2nd-gen tensor core microarchitecture to GPGPU-Sim. Since now, it is possible to simulate the Turing tensor core API in GPGPU-Sim.

Through all these findings, it will be a good base knowledge to researching next-generation tensor cores. The paper will start from very basic backgrounds to give a kind explanation. Therefore, we first look up the basics of CUDA programming in the first part of the Background section. It describes the compilation flow of the CUDA compiler and coding style. Section 2.2 introduces the architecture of GPGPU and its data flow. The basic philosophy of GPGPU is SIMT(Single Instruction Multiple Threads). Thousands of cores execute operations for each thread in parallel. We will look at the architecture that makes this possible and will look at the problems with branch divergence. Section 2.3 shows two two matrix multiplication algorithms. The first approach is the naive manner and explains the problems that arise here. Then we can understand why "tiles" are needed. The fourth section in the background will look at the most important part, the tensor core. Analyze the Volta tensor core, which has already been studied a lot, WMMA API, and the PTX code. In section 2.5, there is an introduction of GPGPU-Sim *4.0* made by Khairy, Mahmoud, et al[6]. This tool can make it possible to simulate the architecture



(a) brief compilation flow[6] (b) detailed CUDA flow[7]

Fig. 2.1. CUDA compilation flow of brief version and detailed version

of GPGPU at the cycle level, and each component can be changed by users. Microbenchmarks are explained in section 3, and proposed microarchitecture is in section 4. Then, we evaluate the result in section 5 and the conclusion in section 6.

All code made from this study is found here, github.com/mayshin10/GPGPU-Sim_Enalbed_Turing_WMMA_API.

2. Background

2.1 CUDA programming model

NVIDIA CUDA API makes it easy to write parallel code for programmers who are familiar with the ANSI C language. You can see the compiling flow in Fig. 2.1(a) when the codes include CUDA language or C/C++ language. Depending on the processor in which the code is performed, the code that executes in the GPU is called the *device kernel*, or the code performed by the CPU is called *host code*. The programmer does not need to attend to whether .cu files contain C/C++ code or not. The *nvcc* compiler recognizes CUDA language and C/C++ language separately. Each file is compiled according to the language it is written and an executable is created through the device linker or host linker.

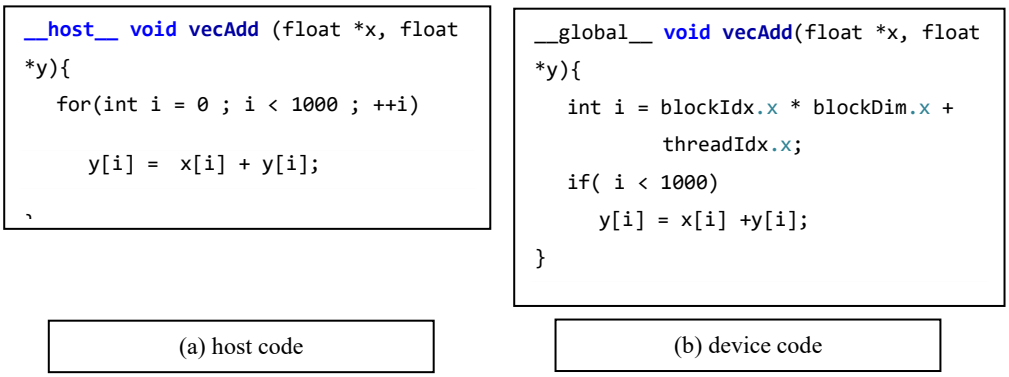


Fig. 2.2. comparison of two codes performing vector addition

Fig. 2.1(b) shows a detailed CUDA flow with GPU hardware. CUDA source code is compiled to the high-level virtual assembly language, PTX(Parallel Thread Execution ISA). Then, it would be translated into its actual hardware assembly language to be executable. This actual level language is called SASS(Streaming ASSembler) and the executable program communicates with GPU hardware via PCI-E connection. Although it is important to know how SASS constructs to understand the hardware's actual behavior, it is not visible for general programmers since it is being kept private by NVIDIA Corporation. Since there have been some

efforts to unveil SASS through the indirect method[1], we now have some information about the SASS.

GPU executes threads in a parallel manner. Threads are grouped into 32 units to form *warp* in terms of NVIDIA and 64 units of *wavefronts* in terms of AMD. These groups have ideally the same control flow and perform the same instruction in a cycle. GPU hardware consists of several *Streaming Multiprocessor*(SM)s. Users can physically bind threads that will fit into the same SM. This is called *thread block*, or *CTA*(Cooperative Thread Array). Then compute kernel launches a single grid which contains thread blocks with 3 dimensions. For example, suppose that there are 16 SMs in a GPU, up to 8 thread blocks per SM, and up to 1024 threads per SM.

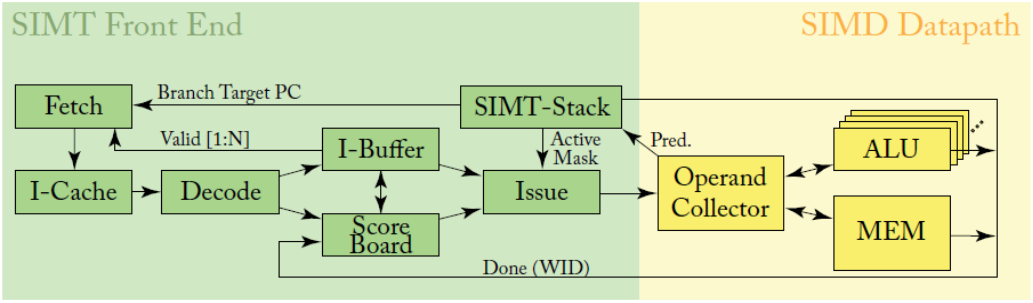


Fig. 2.3. GPGPU pipeline [8]

A programmer can make a total of 128 thread blocks in one grid, and each one can make SM's occupancy 100% if it has 128 threads.

The grid can have dimensions of x, y, z. Each of these can be used in code as gridDim.x, gridDim.y, and gridDim.z. Likewise, a thread block can define their three dimensions, blockDim.x, blockDim.y, and blockDim.z. Each thread in thread blocks has an index number in the block, which can be written as threadIdx. Fig. 2.2(a) is a common C code for calculating the sum of the two vectors. The function is prefixed with "*__host__*" to indicate that it is a host code, however, the compiler recognizes it as a host code even without the keyword "__host__" since it is the default. Fig. 2.2(b) implements the vector addition in parallel. An integer variable *i* is defined as "blockIdx.x * blockDim.x + threadIdx.x". This has the effect of giving their index number to the thread out of the whole threads that are running. Unlike the host code, which

requires 1000 iterations, CUDA is doing vector addition in parallel with 1000 threads. As the number of loops increases, the performance speed of GPU compare to CPU will increase even more.

2.2 Instruction and Register data flow

Fig. 2.3 shows the pipeline of the GPU. The GPU is a device for running multiple threads in one instruction. This is called the SIMT(Single Instruction Multiple Threads) models. To achieve this goal, the hardware makes a schedule by putting several threads together as a warp or wavefront. When it arrives at the exe stage, it is handled by SIMD(Single Instruction Multiple Data) manners by CUDA cores.4

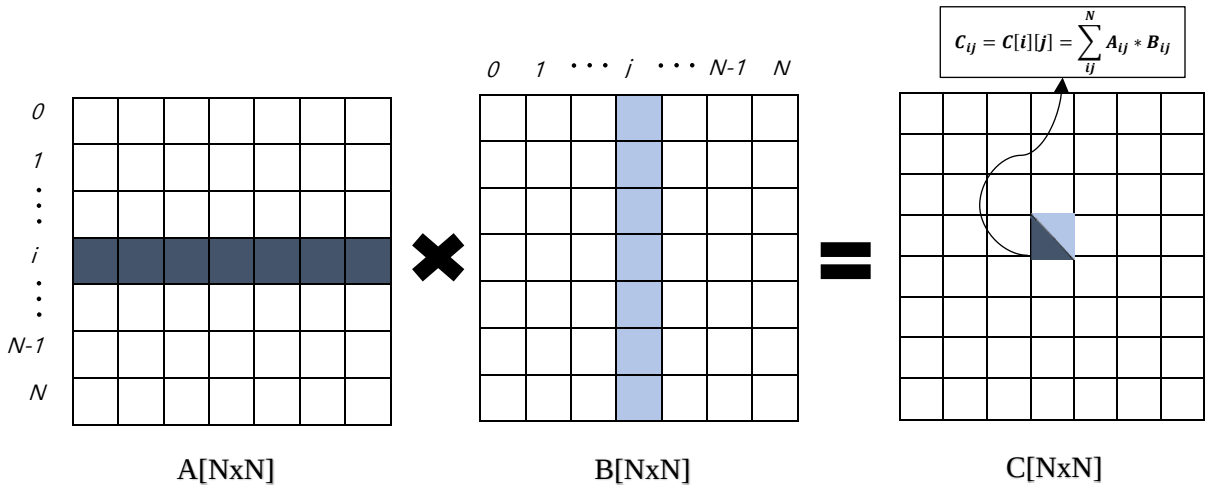


Fig. 2.4. naive matrix multiplication concept. For a C element, it needs N times multiplication and N times addition between A′s elements and B′s elements.

There are three important loops, instruction fetch loop, instruction issue loop, and register access loop. In the instruction fetch loop, the Fetch, I-cache, Decode, I-Buffer units are making this loop to fetch instructions. It is scheduled together in a single pipeline.

The Instruction issue loop consists of I-buffer, scoreboard, issue, SIMT stack units. Each register has one bit for the scoreboard. When one instruction comes, which contains write-back to arbitrary register, the register's scoreboard bit would be set to 1. Consequently, the instruction that comes to the next and wants to read/write the register the scoreboard bit 1 will a cause stall. SIMT stack handles branch divergence problems. Although threads with the same warp should follow the same control flow, they must execute different instructions when they face a conditional statement. Therefore, the execution order must be changed into serializing rather than parallelizing. Threads satisfying branch condition would be performed, and threads that do not become masking off. Masked off threads skip the computations and wait for the control flow to merge again. The SIMT stack, consisting of 4 entries, Re-converge PC (RPC), Next PC, and Active Mask, finds the Re-convergence point through the immediate post-dominator manner. If the RPC and Next PC are the same, the threads will pop in the SIMT stack. Although it is a very

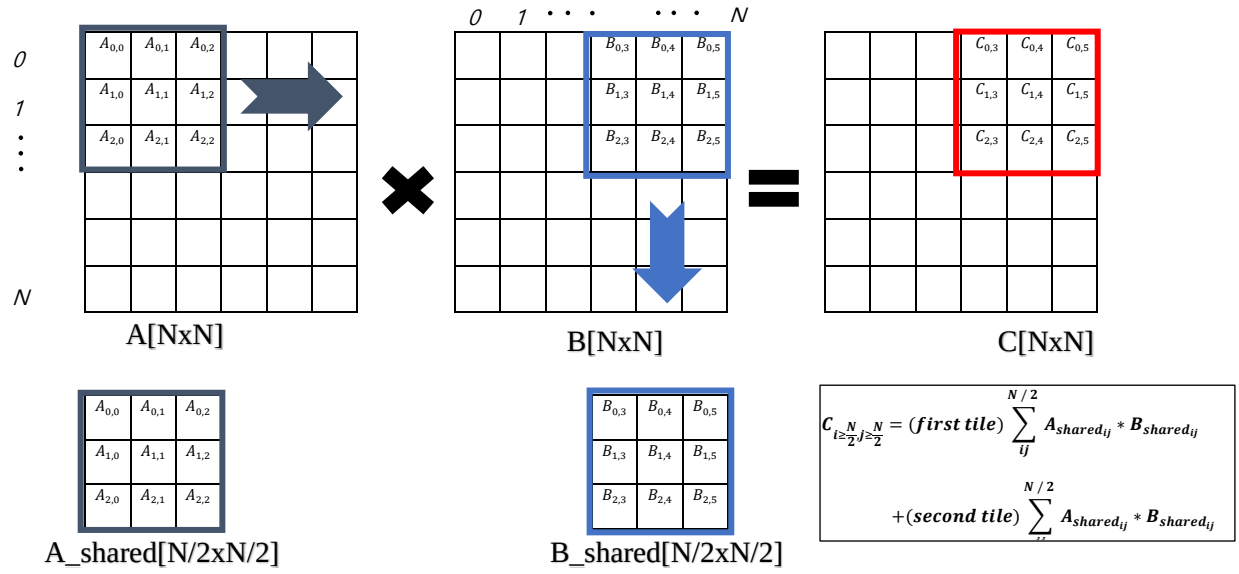


Fig. 2.5. matrix multiplication with 2 x 2 tiled manner. To compute the elements in a red box, black and blue tiles move twice.

intuitive way to handle branch divergence problems, there are some problems and better ways to control the problems according to advanced researches.

The register loop is comprised of operand collector, ALU, MEM units. An operand collector is a unit that has come up to solve the problem of conflict in the register bank. Because in-flight warp probably accesses the same registers, the compositions of the naive register file configuration cause many bank conflicts and stall. Therefore, the operand collector solves this problem by making the bank of the register file accessed by warp different.

2.3 Matrix Multiplication techniques

This section will introduce a total of two matrix multiplication techniques. The latter has better performance, making up for shortcomings of the former.

2.3.1 Naive matrix multiplication

What I introduce first is *Naive matrix multiplication*. It is the most intuitive form that utilizes the advantages of CUDA parallel programming. Fig. 2.4 shows the NxN square matrix.

Multiplying the A(NxN) and B(NxN) matrices results in a C(NxN) matrix. The strategy taken in a naive matrix multiplication is that one thread is responsible for the calculation of one matrix element. In other words, this thread will perform N times multiplication and N times addition among two operand matrix. In the meantime, perform 2N load commands and 1 store command. CGMA is 2N/(2N+1) * 4B ≈ 1 / 4, and since these operations are performed a total of NxN times, the total time complexity is O(N^3), and the memory access is also O(N^3). One thing to be careful about when performing this method, SM processor should be kept at 100% thread occupancy since CGMA is low.

2.3.2 tiled matrix multiplication

The global memory speed, which is lower than the GPU's computational processing speed, binds the entire computing speed to global memory latency. Therefore, when A and B matrices access global memory, caching data with user-defined shared memory has the effect of increasing CGMA. Fig. 2.5 shows an example of the size N/2 x N/2 of the tile. It calculates the NxN matrix by dividing factor 2. The tiles form a total grid of 2x2. The tiles start at (0,0), the tiles of A matrix move in a row, and the tiles of B matrix move in a column, sharing data between threads in the tiles.

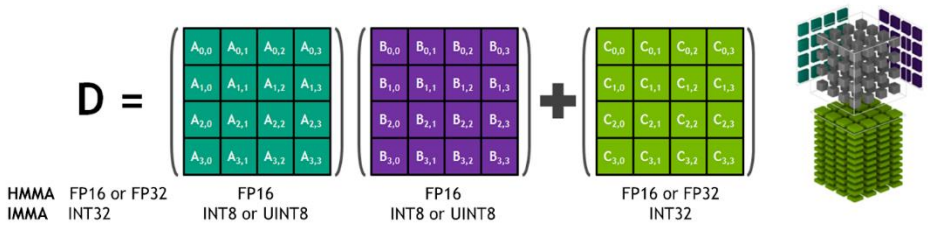


Fig. **2.6**. diagram for tensor operation from NVIDIA whitepaper[4]

2.4 Tensor core

As shown in Fig. 2.6, the tensor core is a unit capable of performing matrix multiply-accumulate operation of 4x4x4 tile size in one cycle when it handles FP16 precision mode. The operation performed by the tensor core is called, D = A x B + C, where each matrix represents a tile-sized matrix. In other words, the tensor core can only calculate the tile-sized matrix rather than calculates the larger matrix directly. Tile size is marked with M x N x K. Matrix A has the size of M x K, and the matrix B has the size of K x N. Of course, D and C should be matrices of size M x N. Three generations of tensor core have been released, and the supported precision and tile sizes are expanding over the generations.

Two tensor cores in Volta architecture or Turing architecture are assigned to one sub-SMs. Both the Volta architecture and the Turing architecture consists of a total of four sub-SMs in one SM, each sub-SM contains one warp-scheduler. Each sub-SM utilizes two tensor cores, so it can be inferred that each tensor core will be scheduled for 16 threads. Ampere's tensor core is not to follow the microarchitecture with them, but to have its original microarchitecture.

2.4.1 1st-gen tensor core

Table 1 shows the tile sizes and precision mode according to the generation of the tensor core. The tensor core of the Volta architecture supports only two precision modes. Both their tile sizes are 16 x 16 x 16. For both precision mode, tile size is 16 x 16 x 16, and precision for matrix A and matrix B is FP16. Multiplying the two floating points results in a precision of FP32. In

Table 1. supported precision and tile size according to the various tensor core generations. The written red color indicates it is supported in the 1st-gen tensor core. The written bold fonts indicate it is supported by the 2nd-gen tensor core. The whole precision and tile size modes in Table 1 are supported by the 3rd-gen tensor core.

precision	tile size(MxNxK)
double	8x8x4
TF32	16x16x8
BFLOAT16	16x16x16
	32x8x16
	8x32x16
FP16(ACC: FP32)	**16x16x16**
	32x8x16
	8x32x16
FP16(ACC: FP16)	**16x16x16**
	32x8x16
	8x32x16
INT8	**16x16x16**
	32x8x16
	8x32x16
INT4	**8x8x32**
INT1	**8x8x128**

FP16 mode, cast it with FP16, and accumulate it to matrix C. In mixed precision mode, the precision of matrix C and D is 32bits, as the accumulate process is conducted with FP32.

2.4.2 WMMA API

WMMA API is an API that connects CUDA programs and tensor core so that users can exploit tensor core as they want. It has the name WMMA(Warp Matrix Multiply-Accumulate) because

```
#include <mma.h>
using namespace nvcuda;
__global__ void wmma_example (atype* a, btype* b, ctype* c, dtype* d, A_STRIDE,
B_STRIDE, C_STRIDE, D_STRIDE){

wmma::fragment<wmma::matrix_a, 16, 16, 16, atype, LAYOUT_A> a_frag;
wmma::fragment<wmma::matrix_b, 16, 16, 16, btype, LAYOUT_B> b_frag;
wmma::fragment<wmma::accumulator, 16, 16, 16, ctype> c_frag;

wmma::load_matrix_sync(a_frag, a, A_STRIDE);
wmma::load_matrix_sync(b_frag, a, B_STRIDE);
wmma::load_matrix_sync(c_frag, a, C_STRIDE, LAYOUT_C);

wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);
wmma::store_matrix_sync(d, c_frag, D_STRIDE, LAYOUT_D);}
```

Fig. 2.7. example code for wmma kernel

the API assigns a warp to the tensor cores. As mentioned in 2.4 tensor core, since one warp-scheduler can schedule warp on two tensor cores, WMMA API uses two tensor cores per warp.

Warp first declares a fragment for matrix A, matrix B, and accumulator(matrix C). From the perspective of warp, one warp should have all the elements of matrix A and matrix B in their register to calculate. Therefore, the threads in the warp have to cooperate to store the whole matrix by dividing up the elements. This is called a fragment and is determined by the thread index. When it is declared, it is needed to pass an argument for LAYOUT, which specifies whether the matrix is stored in a row-major manner or col-major manner. Then the matrix elements are loaded from global memory through the wmma::load operation and all elements in one matrix will be filled in each fragment. STRIDE indicates its original matrix(before dividing into the tile-sized matrix) stride. After synchronized loading is completed, the tensor cores do wmma::mma operation. The calculated values are stored in the accumulator and the calculated results will be stored in global memory when the wmma::store operation is called. At the end of all these steps, matrix multiplication of one tile size is completed.

```
wmma.load.a.sync.alayout.shape.atype {ra_set}, ra_address, ra_stride;
wmma.load.b.sync.blayout.shpae.btype {rb_set}, rb_address, rb_stride;
wmma.load.c.sync.clayout.shpae.ctype {rc_set}, rc_address, rc_stride;
wmma.mma.sync.alayout.blayout.shape.dtype.ctype {rd_set},{rd_set},{rd_set},{rd_set}
wmma.store.d.sync.layout.shape.type {rd_set}, rd_address, rd_stride
```

Fig. 2.8. general PTX code for WMMA API

2.4.3 Assembly language

The CUDA language can be disassembled into two different assembly languages. One is the PTX code, which is a virtual assembly language. It is not one to one mapped with binary code, thus, it is needed more work to convert the PTX code to its real machinery code. It is called SASS code, the actual hardware assembly language. The description of the PTX code is well documented by the vendor[6], but the SASS code is undocumented.

The disassembled WMMA code is shown in Fig 2.8. The ‘sync’ infers that the instruction executes warp in a synchronized manner. layout for oriented direction and type for precision is required to specify the instruction. On the side of the register, the register set can be differed according to the precision and tile size mode. And the second register tells the global memory address storing matrix elements. The stride register contains the stride of the original matrix before slice into tile sized matrix.

2.4 GPGPU-Sim

Since the GPGPU-Sim was first announced at ispass-2009, upgrades have continued to this day. The higher version, the better performance, and now reports a 15%(error %) accuracy for a current version called GPGPU-Sim 4.0 or Accel-Sim. The characteristic of GPGPU-sim is that it is a cycle-level simulator and allows users to change the configuration of hardware. Also, tools such as AerialVision and GPUWattch[9] allow additional functions. In this study, GPGPU-Sim 4.0 was used, and the configuration of RTX 2060 chip which is newly added with

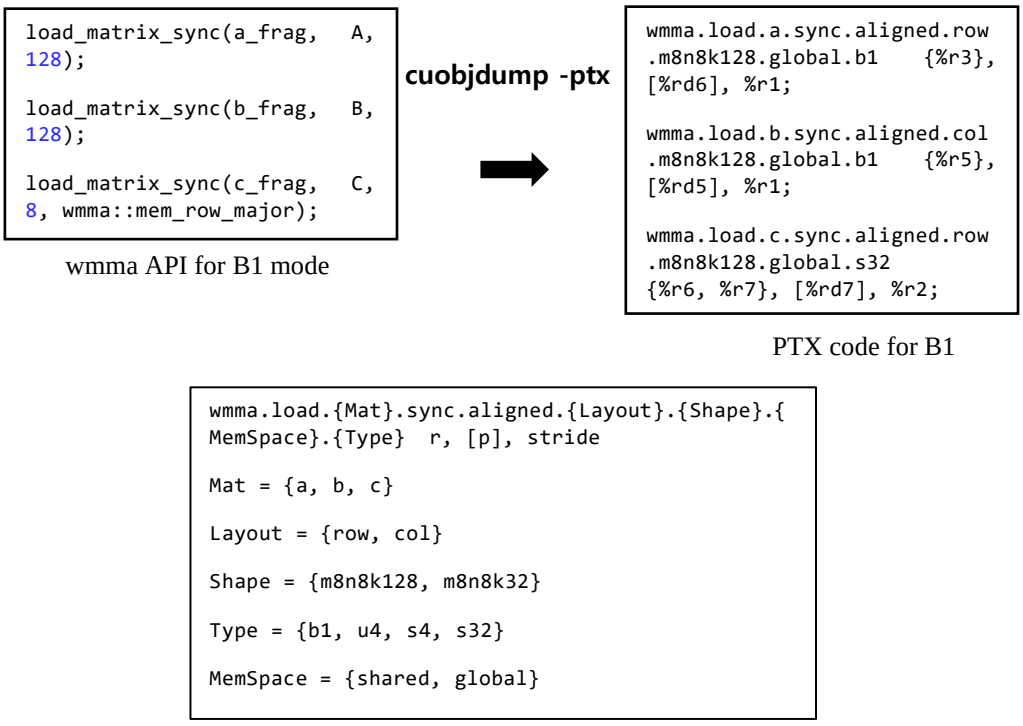


Fig. 3.1. PTX code for second generation tensor core

compute capability 7.5 was used. We will reconfigure GPU hardware for matrix multiplication using GPGPU-Sim.

3. Microbenchmark

3.1. Fragment distribution

3.1.1. PTX analysis for 2nd-gen tensor core

It is not documented which elements of matrix A, B, C, and D would be fetched by a thread within a warp when they use tensor core operation. Thus, we will get the idea of fragment distribution through PTX code analysis. The WMMA API written for B1 precision mode is in

```
wmma.load.a.sync.aligned.row.m8n8k32.global.u4 {%r3}, [%rd6], %r1;
wmma.load.b.sync.aligned.col.m8n8k32.global.u4 {%r5}, [%rd5], %r1;
wmma.load.c.sync.aligned.row.m8n8k32.global.s32 {%r6, %r7}, [%rd7], %r2;
```

Fig. 3.2. PTX code for INT4 precision mode

Fig. 3.1 To investigate the fragment distribution, we used tile-sized matrix multiplication to generate only one warp. The usage of *cuobjdump* tool is similar to *objdump* tool which is used for non-CUDA languages. The CUDA code can be disassembled into PTX code using *cuobjdump -ptx* command.

The generated PTX code in Fig. 3.2 is for the wmma::load command. If look closely, it can be seen that this is slightly different from Fig 3.1. It appears that *.aligned* and *.global* options have been added to the command. The aligned option indicates that the base address and stride should be aligned to the size of the fragment. The size of the fragment is expressed in bytes. For example, for the FP16 precision mode, fragment size in bytes is 32. And actual stride = 2 * stride because stride is not expressed in bytes, but the size of the FP16 elements is 2 bytes. Base address and 2*stride must be aligned to 32 bytes. The global option points to the memory space where the element is located, and there are two possible choices; global and shared.

The right side of Fig 3.1 is the case of B1 precision mode. In the PTX command for matrix A and matrix B, there is only one register in the register set. On register {%r} has a size of 32 bits. Therefore, the number of elements fetched at matrix A and matrix B in B1 precision mode is 32 bits/1 bit = 32. There are two registers in the register set of matrix C. Since the element of matrix C is an integer type, the number of elements that the PTX command fetches is 64 bits / 32 bits =2.

Another experiment feature, INT4 precision mode, is very similar to B1 precision mode. There is only one register in the set of registers for Matrix. One register is 32 bits sized, so 32 bits / 4 bits = 8 elements have been fetched. Matrix C has two registers in the register set, and

```
wmma.load.a.sync.aligned.row.m16n16k16.global.u8 {%r2, %r3}, [%rd6], %r1;

wmma.load.b.sync.aligned.col.m16n16k16.global.u8 {%r4, %r5}, [%rd7], %r1;

wmma.load.c.sync.aligned.row.m16n16k16.global.s32{%r6, %r7, %r8, %r9, %r10, %r1
                                                  1, %r12, %r13}, [%rd8], %r1;
```

Fig. 3.3. PTX code for INT8 precision mode

```
wmma.load.a.sync.aligned.row.m16n16k16.global.f16
{%r2, %r3, %r4, %r5, %r6, %r7, %r8, %r9}, [%rd5], %r1;

wmma.load.b.sync.aligned.col.m16n16k16.global.f16
{%r10, %r11, %r12, %r13, %r14, %r15, %r16, %r17}, [%rd6], %r1;

wmma.load.c.sync.aligned.row.m16n16k16.global.f16

{%r18, %r19, %r20, %r21}, [%rd7], %r1;
```

Fig. 3.4. PTX code for FP16 precision mode

```
wmma.load.a.sync.aligned.row.m16n16k16.global.f16
{%r2, %r3, %r4, %r5, %r6, %r7, %r8, %r9}, [%rd7], %r1;

wmma.load.b.sync.aligned.col.m16n16k16.global.f16
{%r10, %r11, %r12, %r13, %r14, %r15, %r16, %r17}, [%rd8], %r1;

wmma.load.c.sync.aligned.row.m16n16k16.global.f32

{%f1, %f2, %f3, %f4, %f5, %f6, %f7, %f8}, [%rd6], %r1;
```

Fig. 3.5. PTX code for mixed precision mode

the precision of the accumulator in integer type. Therefore, one fragment for accumulator has 64 bits / 32 bits = 2 integer type elements.

From INT8 precision mode, the size of the fragment varies depending on the size of the tile. For the tile size of 16 x 16 x 16, only two register sets are for matrix A and matrix B. INT8 precision has an 8-bit size, so fragment contains 64 bits / 8 bits = 8 matrix elements. The accumulator is an integer type in the size of 32 bits. The number of elements that accumulator consists of is 8 * 32 bits / 32 bit and the set of register consists of eight.

For FP16 precision mode, matrix A and matrix B have eight registers in a register set. Fragments have 8*32 bits / 16 bits = 16 elements because it is 16 bit size, and accumulator has 4 * 32 bits / 16 bits = 8 elements because it is as set of four registers.

```
__device__ void itialization( … ) {
          for(int i = 0 ; i < matrix_width * matrix_height ; i++)
                    matrix[i] = i;
}
__global__ void fragment_microbenchmark( … ){

<Fragment_declaration> matrix_frag;
wmma::load_matrix_sync(matrix_frag, mem_addr, stride);
for(int i = 0 ; I < matrix_frag.num_elements; i++){
         float t=static_cast<float>(matrix_frag.x[i]);
         printf("THREAD%d CONTAINS %.2f\n",threadIdx.x,t);
}
```

Fig. 3.6. microbenchmark for fragment distribution(reproduced from [1])

The mixed precision mode is identical to matrix A and matrix B to FP16. Since the accumulator has four registers, it has 4*32 bits / 32 bits = 4 elements. The register marked {%f} is for soring a float type value.

3.1.2. Fragment distribution Microbenchmark

Based on the suggestion from section 3.1.1, the microbenchmark identified exactly where the element is fetched. Microbenchmark for fragment distribution is illustrated in Fig 3.6. At the beginning of the code, the matrix is initialized with an increasing integer value to specify its own index. After initialization, run the wmma::load kernel for one warp. As described WMMA API usage, declaring the fragment of matrix A, B and accumulators produce arrays that will fit elements as many as num_elements. when wmma::load is performed, the array is filled with elements, and since the matrix has already been initialized into an increasing sequence, the output of each thread indicates the index of elements that have been fetched. This microbenchmark revealed the fragment distribution as shown in Fig. 3.7-Fig. 3.10.

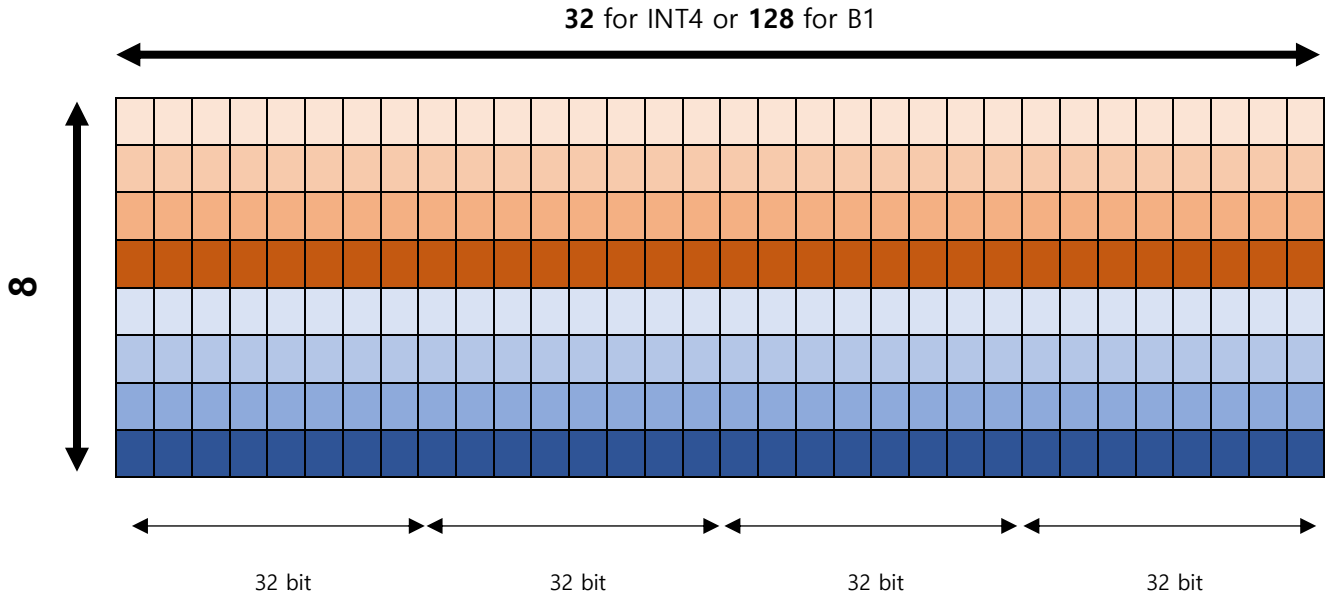


Fig. 3.7. fragment distribution for Matrix A and B

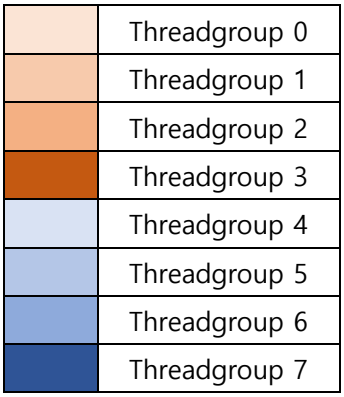


Fig. 3.8. threadgroup information according to the color of the row

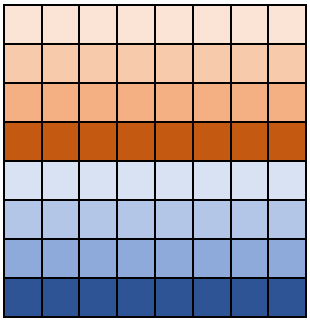


Fig. 3.9. fragment distribution for Matrix C

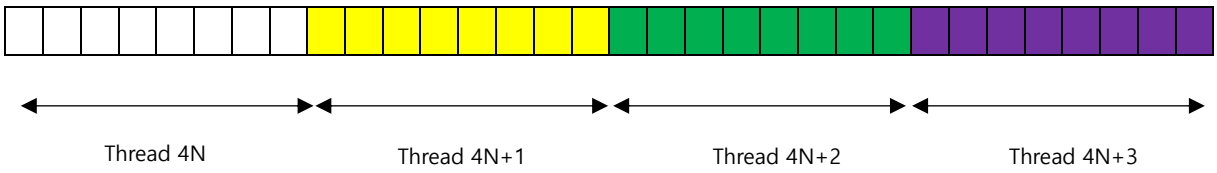


Fig. 3.10. thread distribution inside a threadgroup

3.1.3. Fragment distribution results

Fig. 3.7 is a fragment distribution that has been identified for INT4 precision mode, and B1 precision mode, the two experimental features of 2nd-gen tensor core. In INT4 mode, the width of the matrix is 32 and in B1 mode is 128. One row is all fetched by one thread group. Thus, elements are fetched by a total of eight thread groups, each element is fetched by only one thread without duplication. This is different from the 1st-gen tensor core of the Volta architecture. And also it is different from the FP16 precision mode and Mixed precision mode of the second generation tensor core.

Fig. 3.7 is for row-major matrix A. The col-major matrix B has also the same fragment distribution and it is transposed in shape with Fig. 3.7. Because INT4 precision mode and B1 precision mode, which are experimental features, are allowed only row-major for matrix A and col-major for matrix B. It is not necessary to think about fragment distribution other than Fig. 3.7. One thread brings 32-bit sized memory as we have seen in section 3.1.1. Therefore, eight elements are fetched into one thread in INT4 precision mode, and 32 elements are fetched into a thread in B1 mode.

Fig. 3.9 is a fragment distribution that has been identified for Matrix C, named accumulator. Likewise, one threadgroup fetches all elements in a row. Since the accumulator of the experimental feature is common at 8x8 size, it has brought two elements per thread both in INT4 precision mode and B1 precision mode.

Fig. 3.10 represents the fetching pattern of each thread in the threadgroup. It is not complicated, but the thread index is just increased by 1 in order. For example, the fragment to fetch the elements of row 3 of matrix A is threadgroup2. And thread8, thread9, thread10 and, thread11 has fetched elements. When it works for matrix C, thread8, thread9, thread10 and thread11 fetch two elements per thread to completely fetch all elements in a row.

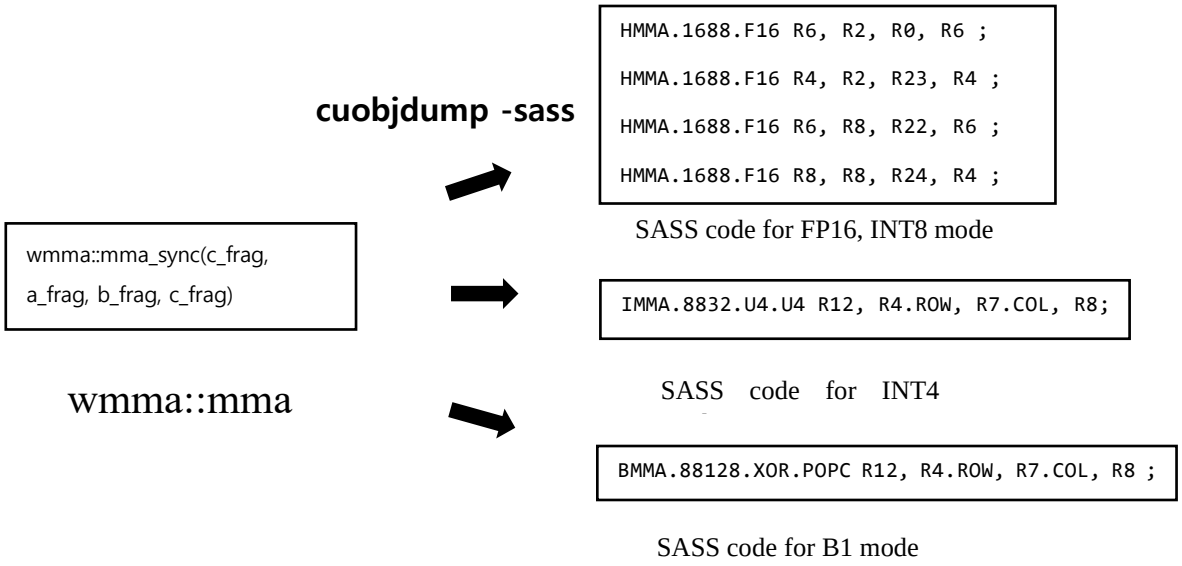


Fig. 3. 11. SASS code for wmma::mma instruction

3.2 Computation pattern

3.2.1 SASS analysis for wmma::mma

After the elements have been loaded in each fragment, the wmma::mma command leads to the tensor core to carry out the matrix multiply-accumulate operation. When it is disassembled in PTX code, it looks like atomic instruction. However, it is disassembled in SASS code, it will look as shown in Fig. 3.11. As we looked at before, the SASS code is the language of actual hardware. Because the GPU hardware changes according to the architecture generation, the capability to perform instruction is varied. Therefore, all CUDA codes perform instructions at the SASS level.

The wmma::mma command is disassembled into different types of SASS codes depending on the precision mode. In Mixed, FP16, and INT8 precision mode, a total of four HMMA or IMMA operations are grouped for a wmma::mma command. Thus, a total of four sets of computation patter are made.

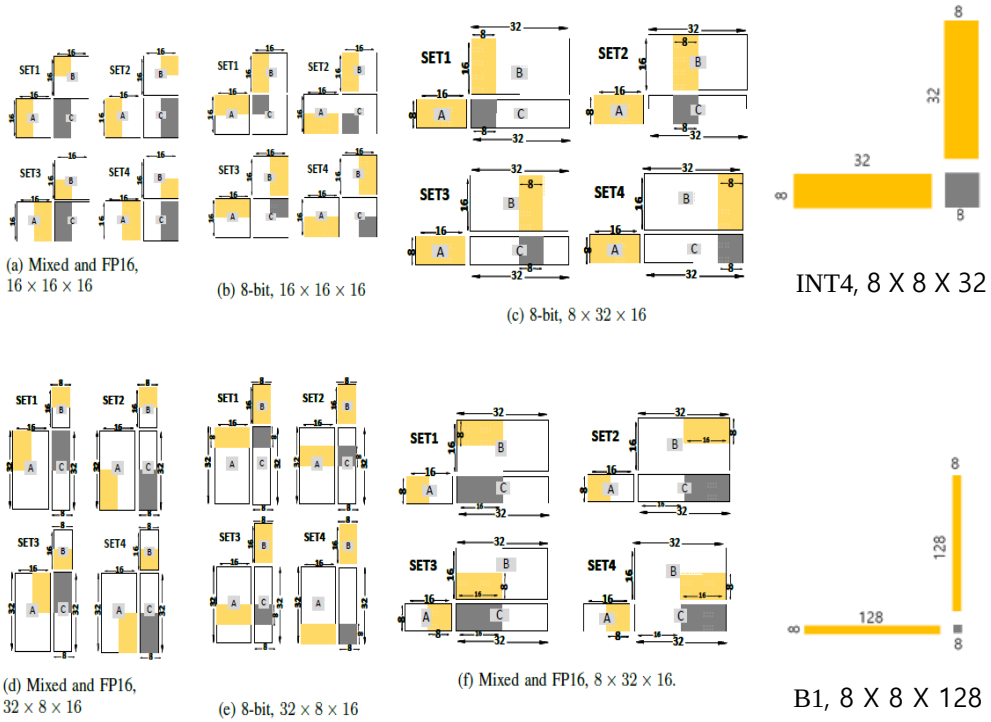


Fig. 3.12. computation patter result for second generation tensor core. left side figures are from Raihan et al(2019), and right side are implemented by our study.

In INT 4 mode, IMMA commands are performed with atomic instruction. Similarly, in B1 precision mode, only one BMMA command is generated. Note that unlike other operations, the XOR operation is carried out and the option POPC is to count number 1 as a result of XOR.

3.2.2 computation pattern results

Fig. 3.12 shows the computation pattern through microbenchmark. As the research on microbenchmark and computation pattern for Mixed precision, FP16, INT8 precision mode are conducted by Raihan et al(2019), the figure contains the result of prior studies. The results for the right experimental features are found in this study and are performed as shown in the graphic. Since it is an atomic instruction, both INT4 precision mode and B1 mode can be conducted in one instruction resulting in an 8x8 integer matrix.

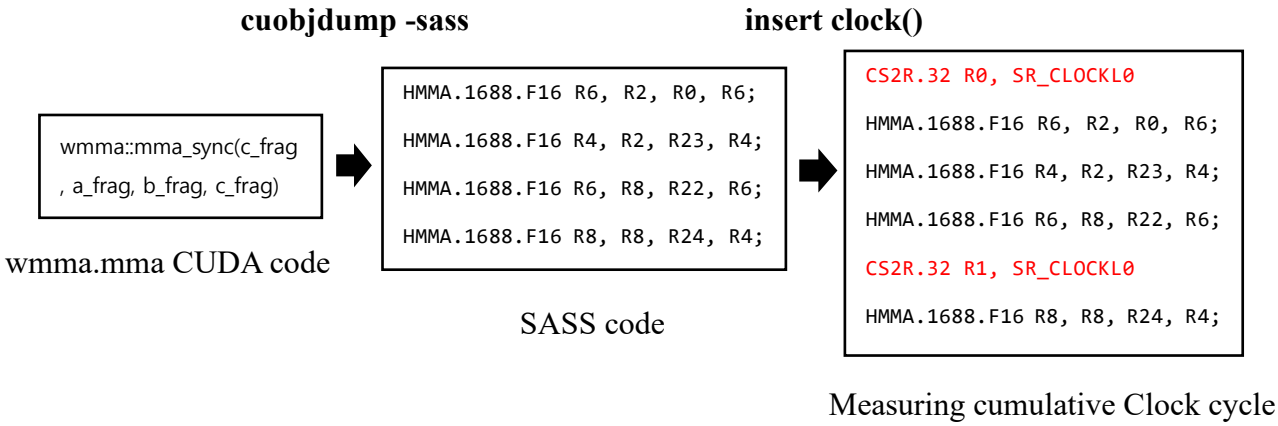


Fig. 3.13. microbenchmark measuring clock cycle

3.3 Clock profiling.

3.3.1 Microbenchmark measuring clock cycle

For the case that wmma::mma command is not disassembled into atomic instruction, we show wmma.mma PTX code is divided into multiple SASS codes. Microbenchmark is constructed to profile the clock cycles consumed by SASS instruction in an accumulative manner. In Fig. 3.13 , the wmma::mma command is divided into four HMMA instructions through the *cuobjdump -sass* command. The shown example is a case of FP16 precision mode. Its computation pattern consists of a total of four sets. When the clock() function is inserted and disassembled into SASS code, it is converted into *CS2R* SASS code.

Before adjusting the binary file, wmma.mma instruction is located between two clock instructions. It could not measure the one HMMA instruction but measure four HMMA instructions. the two clock() function locations could be adjusted on the binary file to measure the cumulative clock cycle. In this example, a CS2R instruction adjusted its position from the bottom of the instructions to one step up. Therefore, it can measure three HMMA instructions' accumulative clock cycle. Table 2 shows the results measured by RTX 2080 Ti.

Table 2. average cumulative clock cycle measured by RTX 2080 Ti

precision	tile size(MxNxK)	Average Cumulative Clock Cycles			
		SET1	SET2	SET3	SET4
Mixed precision	16x16x16	45	59	80	102
	32x8x16	60	75	96	118
	8x32x16	42	56	77	99
FP16	16x16x16	42	50	60	74
	32x8x16	44	52	58	72
	8x32x16	42	50	58	72
INT8	16x16x16	47	51	55	65
	32x8x16	58	62	66	76
	8x32x16	38	42	46	56
INT4	8x8x32	365	-	-	-
B1	8x8x128	365	-	-	-

Table 3. clock interval between set and set

precision	tile size(MxNxK)	Average Cumulative Clock Cycles			
		SET1-SET2	SET2-SET3	SET3-SET4	Avg.
Mixed precision	16x16x16	14	21	22	19
	32x8x16	15	21	22	19.3
	8x32x16	14	19	22	18.3
FP16	16x16x16	8	10	10	9.3
	32x8x16	8	8	14	10
	8x32x16	8	8	14	10
INT8	16x16x16	4	4	10	6
	32x8x16	4	4	10	6
	8x32x16	4	4	10	6
INT4	8x8x32	-	-	-	-
B1	8x8x128	-	-	-	-

3.3.2 Clock cycle results

In Table 2, all supported precision mode and tile size by Turing architecture's 2nd-gen tensor core were measured. Each result is reported in average value with 10 times iterations and it could be varied depending on the hardware which has Turing architecture chipset.

Comparing the mixed precision mode with the FP16 precision mode, one can see that the mixed precision mode consumes approximately twice as many as the FP16 precision mode does. This is clear by comparing values in Table 3, which is presented the clock cycle interval between set and set. It has the advantage to compare the clock cycle without instruction launch time. As a result, the clock interval between the set and the set is 18-19 in mixed precision mode and 9-10 in the FP16 mode. Therefore, we can get a clue about the how microarchitecture of the tensor core is constructed.

For INT8 precision mode, it consumes approximately half of the clock cycle than FP16 precision mode. Table 3 shows that INT8's clock interval average is 6 cycles, and all of them are the same regardless of the tile size. This is because tensor core handles matrix multiply-accumulate operation as an identical four set of computation pattern like we have seen before. The computation speed-up of the INT8 precision mode is as much as less throughput than FP16 precision mode; twice.

Only a single value exists for INT4 precision mode and B1 precision mode since their mma operations are atomic instructions. Therefore, unlike other precision modes in Table 2, only one clock cycle value exists, and for the same reason, the value cannot be displayed in Table 3. Note that the consumed clock cycle in INT4 precision mode and B1 precision mode is the same. This gives the number of units supporting INT4 matrix multiply-accumulate operations and the number of units supporting XOR-POPC operations within the tensor core.

Combining Fragment distribution, computation pattern, and clock cycle information, we could propose the microarchitecture for Turing tensor core.

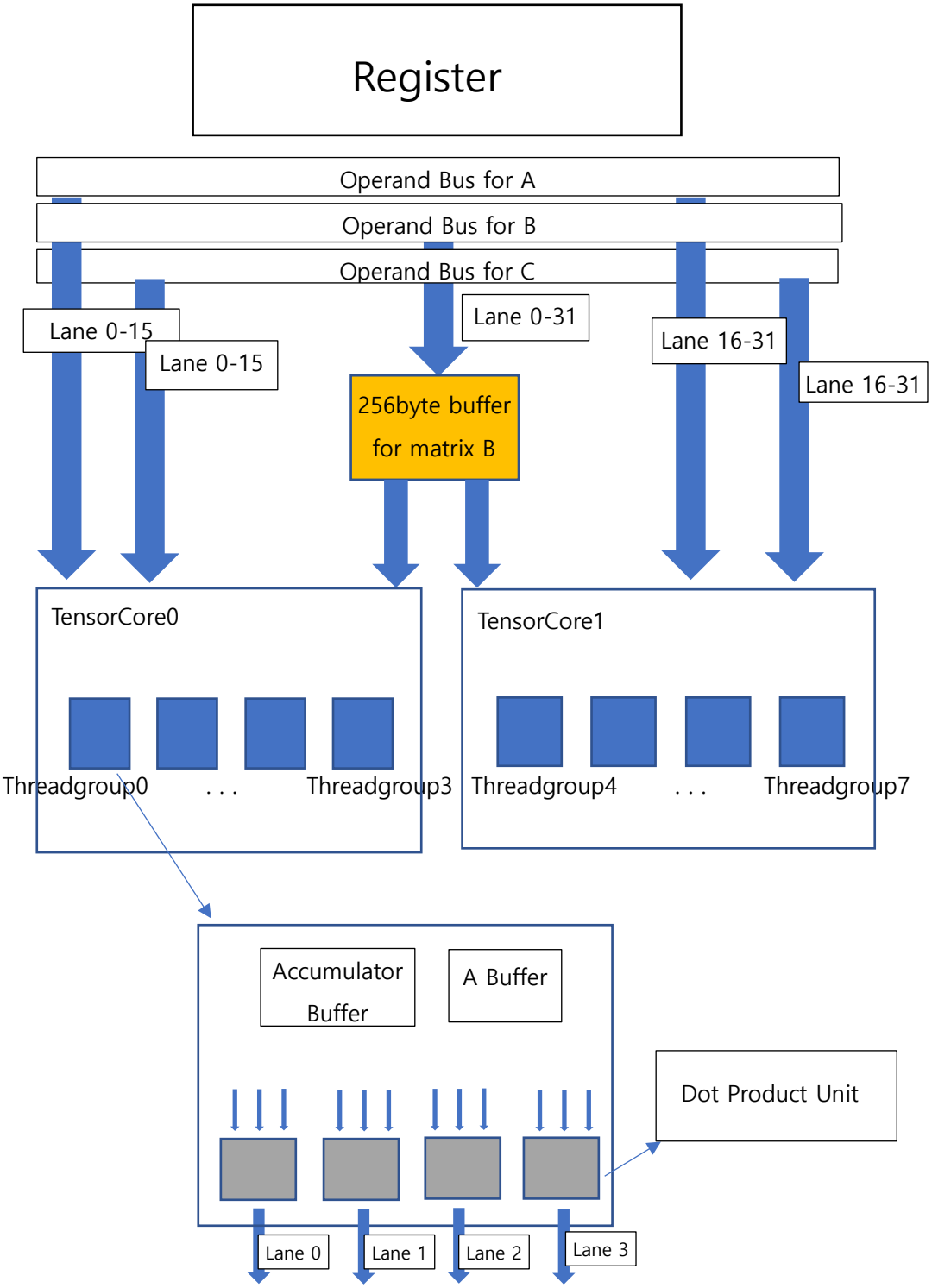


Fig. 4.1. proposed microarchitecture for second generation tensor core

4. Proposed microarchitecture

Fig. 4.1 is the proposed microarchitecture of the 2nd-gen tensor core of the Turing architecture. It can be inferred from information obtained through microbenchmarks.

There are two tensor cores in sub-SM with one warp scheduler. The operand buses are connecting registers and the tensor cores by 32 SIMD lanes. The SIMD lane is arranged from index 0 to 31 in sequence. Thus, when the values in the fragment are fetched from registers, the bank conflict will be reduced.

Unlike the proposed tensor core microarchitecture of Volta architecture, two tensor cores of the Turing architecture cannot operate independently. This is because in the fragment distribution we looked at, one element was fetched only by one thread. Therefore, there should be a buffer shared by the two tensor cores between the tensor cores.

This buffer may only share elements of matrix A or only elements of matrix B. The microarchitecture we proposed assumes that elements of matrix B are shared by the buffer. Matrix A and accumulator have buffers inside the threadgroup. Through these buffers, matrix A buffer, matrix B buffer, and accumulator, each threadgroup computes their one output without any elements from outside the threadgroup.

The dot product units in the threadgroup produce a total of four outputs. One dot product unit consists of several FEDP(four element dot product) units according to the precision mode. There are one FEDP units for FP16 precision mode inside the dot product unit. And this FEDP takes one more cycle when it accumulates FP32 precision values so that mixed precision mode takes twice as long as the FP16 precision mode.

FEDP units for INT8 precision mode are twice as much as the FP16 precision mode has. Therefore, it takes twice less than FP16 precision mode. It makes possible making two outputs per one cycle.

The number of FEDP units for INT4 can be inferred from the peak throughput of INT4 precision mode described from the NVIDIA whitepaper[4]. The whitepaper states that the peak throughput of INT4 precision mode is twice that of INT8. Therefore, we supposed that the number of FEDP units in INT4 is twice that of INT8 and four times that of FP16.

We have seen that the consumed clock cycle by INT4 precision mode and B1 precision mode were the same in Table 2. However, INT4 has an 8x8x32 tile size and B1 has an 8x8x128 tile size, so the number of B1's XOR-POPC units is four times greater. To sum up, the dot product unit contains sixteen XOR-POPC units for B1 precision mode, four FEDP units for INT4 precision mode, eight FEDP units for INT8 precision mode, and one FP16 FEDP unit which takes one more cycle when it accumulates FP32 precision values.

5. Evaluate

5.1. Building GPGPU-Sim

To evaluate the proposed microarchitecture, we made simulations for checking the correlation between the proposed model and actual hardware. GPGPU-Sim currently supports only the WMMA API of the Volta architecture. Therefore, the simulator was modified so that the API of Turing architecture's tensor core could be implemented.

The PTX code of Turing tensor core's API is slightly different from the PTX code of the Volta architecture's tensor core API as we have seen in section 3. First of all, ptx.l and ptx.y files should be modified to include the new PTX code. The functionality of the simulator should then be mounted through instruction.cc which specifies functional behavior according to the incoming PTX code. The code must be configured in consideration of the Fragment distribution we have disclosed. By adding Load, MMA, Store instructions, the GPGPU-Sim can operate second-generation tensor core's API functionally,

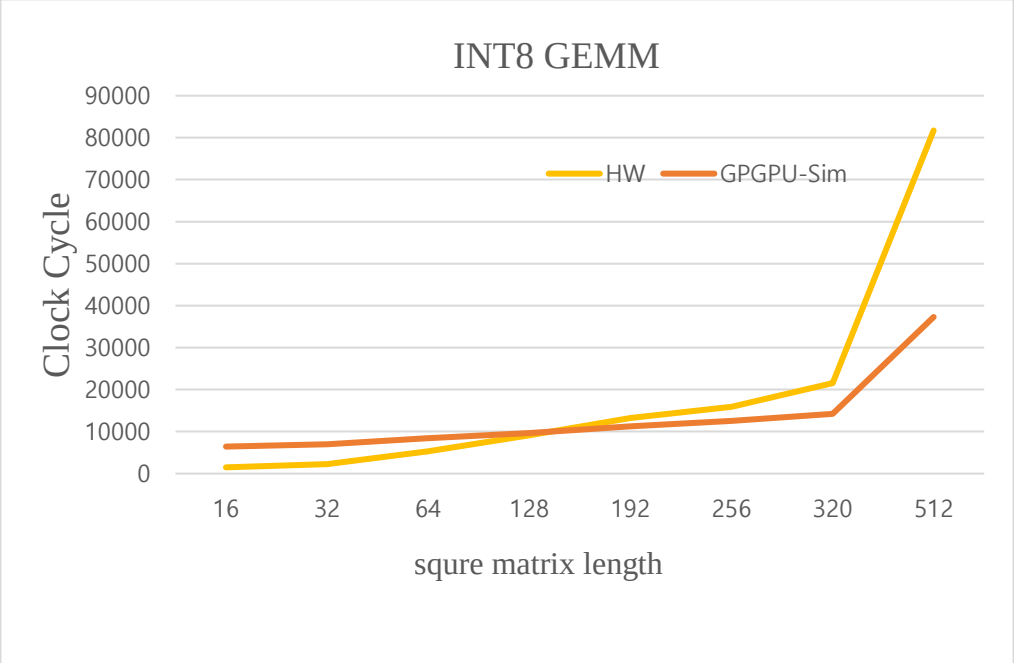


Fig. 5.1.1. consumed clock cycle for INT8 GEMM kernel

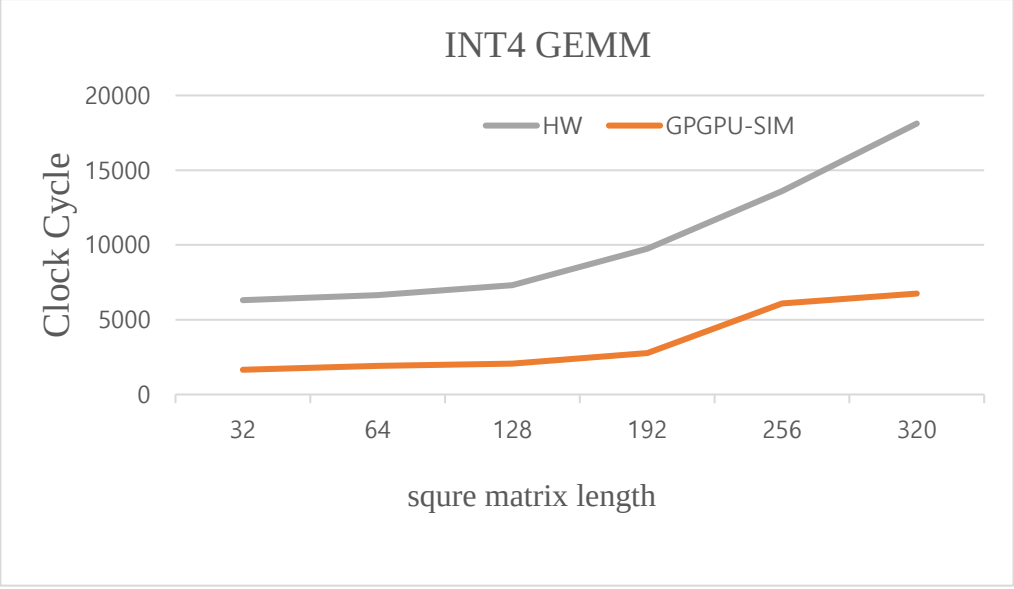


Fig. 5.1.2. consumed clock cycle for INT4 GEMM kernel

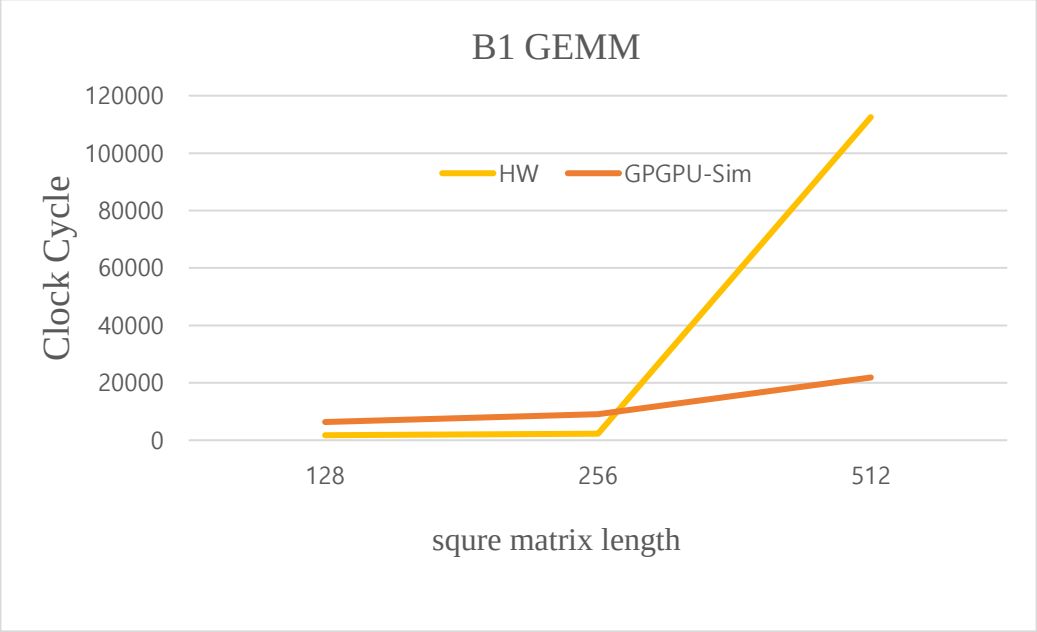


Fig. 5.1.3. consumed clock cycle for B1 GEMM kernel

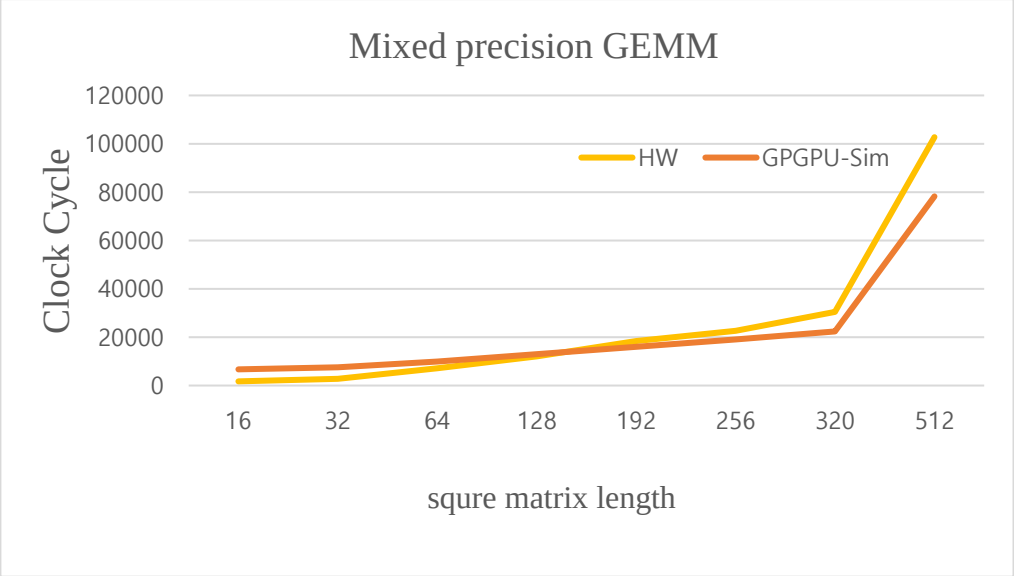


Fig. 5.1.4. consumed clock cycle for mixed precision GEMM kernel

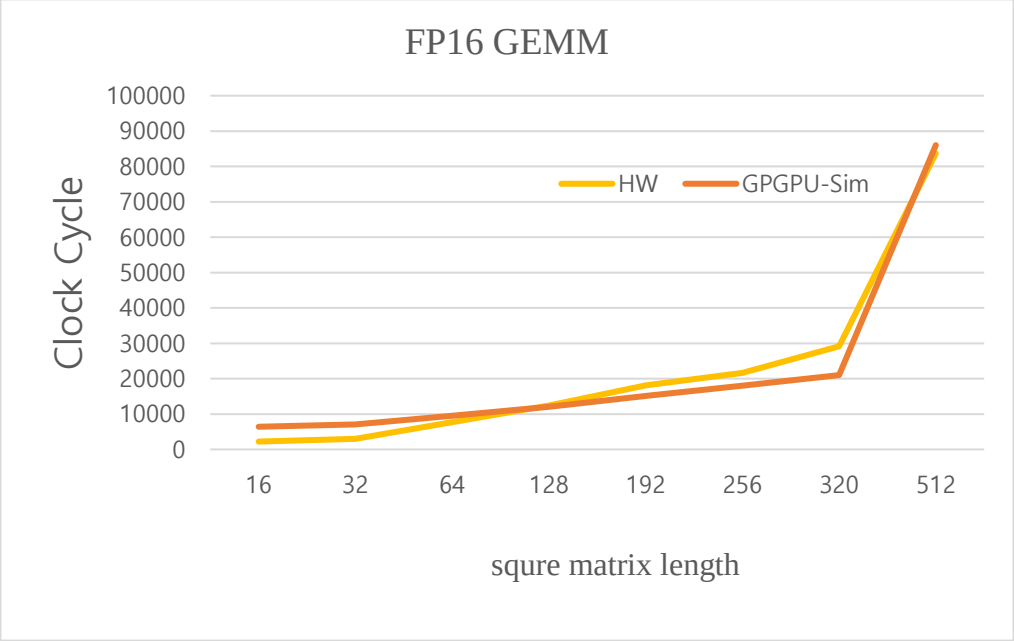


Fig. 5.1.5. consumed clock cycle for FP16 GEMM kernel

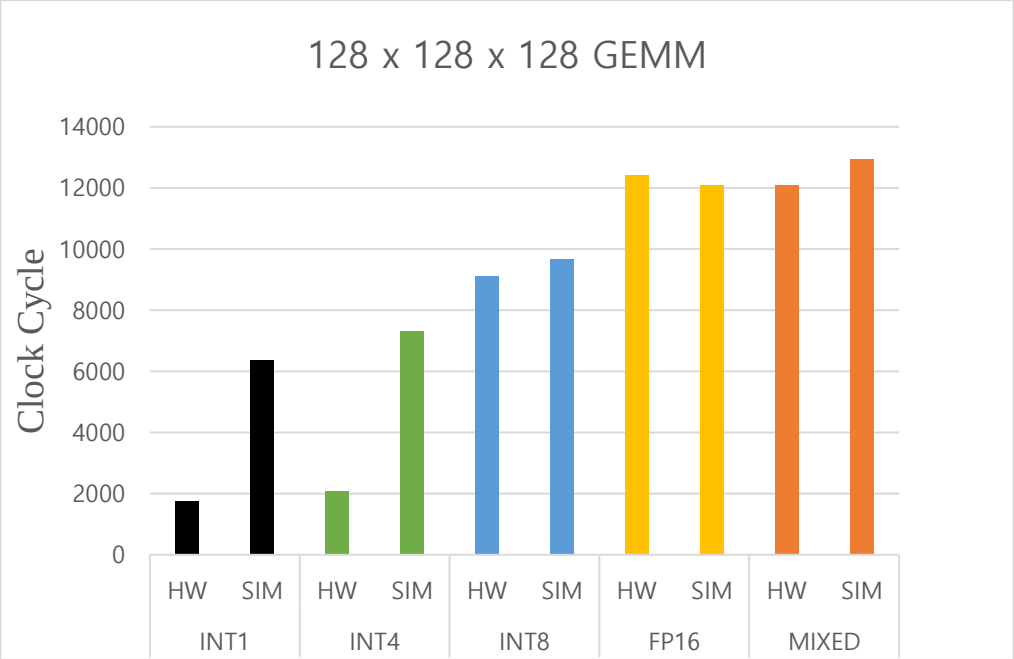


Fig. 5.2. comparison of clock cycle according to the precision mode

5.2. Evaluate

We launched GEMM(General Matrix Multiplication) kernel with square matrix operands(w x w x w) and compared the clock cycle required by GPGPU-Sim and actual hardware.

Fig. 5.1.1 is the result of INT8 precision mode. The x-axis represents the length of a row or column of a square matrix. The larger the size of the hardware and simulator, the greater the clock cycles. The cycle increases exponentially at the matrix size is 320 x 320 x 320. This result is shown in both hardware and simulator.

Fig. 5.1.2 is the result of INT4 precision mode. Correlation with actual hardware and simulation results is consistent, but the gap of clock cycles between hardware and simulation result. This is because the clock delay and latency constants according to the computation do not fit well. It can be handled by adjusting instruction launching latency and delay through fine-tuning.

Fig. 5.1.3 is the result of the B1 precision mode. B1 precision mode has only three experimental samples. There are only 128x128x128, 256x256x256, and 512x512x512. This is because the default tile size for B1 precision mode is 8x8x128 so that the GEMM matrix can have its row or column as multiple of a 128. In addition, when the matrix size exceeded 512, the measurement was practically impossible since simulation time would increase exponentially. For this reason, there are many differences between the actual hardware and the simulator in the experimental results. To correct this, more experiments should be conducted to determine the latency and delay constants for tensor operation. For the mixed precision mode and FP16 mode, the correlation and similarity between the actual hardware and simulator are remarkably accurate.

Fig. 5.2 compared the correction of hardware and simulator according to the precision. In common, 128x128x128 GEMM kernel was launched. The gradual increase in the number of clock cycles according to the precision mode comes out very well in both the actual hardware

and simulator. Without B1 and INT4 precision modes, the correction of hardware and simulator is also very good. B1 and INT4 precision modes should be matched more accurately by fine tuning the latency and delay constants.

6. Conclusion

Through this study, tensor core microarchitecture of Turing architecture could be modeled. As a result of microbenchmarks, we knew the fragment distribution, computation pattern and clock information. Base on the result, we could model reasonable microarchitecture.

Then we modified GPGPU-Sim to evaluate the proposed microarchitecture, and we found that it behaves the same as the actual hardware. Therefore, we could confirm that our proposed microarchitecture is valid.

With this research, we are expecting that there will be many advances in microarchitecture research of deep learning accelerators. We hope the result of the study will be used to develop microarchitecture of the Ampere architecture, the next generation of the tensor core by NVIDIA Corporation and deep learning accelerators similar to the tensor core.

Reference

[1] Raihan, Md Aamir, Negar Goli, and Tor M. Aamodt. "Modeling deep learning accelerator enabled GPUs." *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2019.

[2] Jia, Zhe, et al. "Dissecting the nvidia volta gpu architecture via microbenchmarking." arXiv preprint arXiv:1804.06826 (2018).

[3] Jia, Zhe, et al. "Dissecting the NVidia Turing T4 GPU via microbenchmarking." arXiv preprint arXiv:1903.07486 (2019).

[4] NVIDIA Turing Architecture Whitepaper. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[5] Li, Ang, and Simon Su. "Accelerating Binarized Neural Networks via Bit-Tensor-Cores in Turing GPUs." arXiv preprint arXiv:2006.16578 (2020).

[6] NVIDIA Corporation CUDA toolkit documentation v11.1.0. https://docs.nvidia.com/cuda/index.html

[7] Bakhoda, Ali, et al. "Analyzing CUDA workloads using a detailed GPU simulator." 2009 IEEE International Symposium on Performance Analysis of Systems and Software. IEEE, 2009.

[8] Aamodt, Tor M., Wilson Wai Lun Fung, and Timothy G. Rogers. "General-purpose graphics processor architectures." Synthesis Lectures on Computer Architecture 13.2 (2018): 1-140.

[9] Leng, Jingwen, et al. "GPUWattch: enabling energy optimizations in GPGPUs." ACM SIGARCH Computer Architecture News 41.3 (2013): 487-498.

[10] Kirk, David. "NVIDIA CUDA software and GPU parallel computing architecture." ISMM. Vol. 7. 2007.

[11] NVIDIA Corporation. CUDA C++ programming guide https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[12] Khairy, Mahmoud, et al. "Accel-sim: an extensible simulation framework for validated GPU modeling." 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2020.

[13] T. M. Aamodt, W. W. Fung, I. Singh, A. El-Shafiey, J. Kwa, T. Hetherington, A. Gubran, A. Boktor, T. Rogers, and A. Bakhoda, “GPGPU-Sim3. x manual,” http://gpgpu-sim.org/manual/index.php/Main Page

[14] Leng, Jingwen, et al. "GPUWattch: enabling energy optimizations in GPGPUs." ACM SIGARCH Computer Architecture News 41.3 (2013): 487-498.

[15] COLEMAN, CODY, and DANIEL KANG. "Efficient CUDA." (2017).

국 문 요 약

실험 기능을 포함한 튜링 아키텍처의 텐서 코어 마이크로아키텍처 모델링

2017년 발표된 볼타 아키텍처를 기점으로 NVIDIA에서는 텐서 코어라는 새로운 연산장치를 GPU에 탑재하기 시작했다. 텐서 코어는 행렬곱과 콘볼루션이 계산양의 대부분을 차지하는 딥러닝 학습에 특화된 연산 장치이기 때문에, 이전 세대의 GPU에 비해 행렬 연산에서 놀라운 속도 향상을 보인다.

본 연구에서는 2018년 발표된 튜링 아키텍처에 탑재된 2세대 텐서 코어의 마이크로아키텍처를 분석하였다. NVIDIA에서 텐서 코어의 내부를 공개하지 않기 때문에, 마이크로벤치마킹을 통해 프로파일링하는 과정을 거쳐야만 한다. 그 결과를 토대로 2세대 텐서코어의 마이크로아키텍처를 제안하고, GPGPU-Sim을 최신화하여 이것의 적합성을 평가하였다.

2세대 텐서 코어는 실험적 기능들이 추가되어있고, 사용자는 wmma API를 통해서 이 기능들을 사용할 수 있다. 본 연구에서는 텐서 코어의 실험적 기능들을 모두 포함하고 있는 마이크로아키텍처를 제안하고 있고, GPGPU-Sim 역시 새로운 텐서 코어의 실험적 기능까지 지원할 수 있게끔 재구성되어 있다.

핵심되는 말 : 2세대 텐서 코어, 마이크로아키텍처, gpgpu-sim