Exploration of Parallel Algorithm with Heterogeneous Computing and Memory Architecture on NVIDIA Devices

Fan Shi
fans@andrew.cmu.edu

Yuwei Xiao
yuweix@andrew.cmu.edu

Summary

NVIDIA GPUs provides hierarchy of memory structure and Tensor Core Unit to speedup parallel algorithm. In this experimental study, we explore the mechanism and usage of Tensor Core unit and modify some frequently used parallel algorithm (GEMM, Reduction) to use Tensor Core Unit. We write additional CUDA implementations to explore and present Unified-memory usage, WARP-awareness CUDA programming and intra-WARP communication through Register Files. The experiments are conduct on Jetson Xavier NX platform and we perform algorithm evaluation using nv-profile.

Code: **https://github.com/Fan-Shi/15618-project**

1 Background

1.1 Hardware Features

Our team read through the Volta architecture white paper and summarize important features including Tensor Core units and memory hierarchy.

1.1.1 Tensor Core Unit

From Volta architecture, NVIDIA introduces Tensor Core Unit (TCU) into GPU's processing block (sub-structure of streaming multiprocessor). The TCUs can perform 4×4 matrix multiplication. TCUs can significantly improve the performance of General Matrix Multiplication with the cost of over-specialization. TCUs can only perform small accumulative matrix multiplication at the granularity of **Warp**.

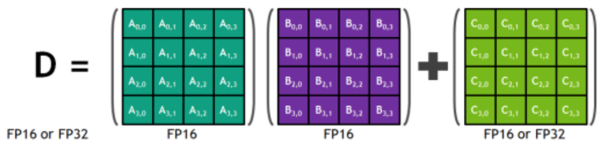


Figure 1: Tensor Core Operations

1.1.2 GPU memory hierarchy

NVIDIA GPUs have several level of memory hierarchy and in CUDA programming we need to explicitly allocate specific type of memories to guarantee performance. Different from CPU's memory hierarchy, GPU has limited amount of L1 Cache (shared memory) but higher number of register files per SM. Memory loading from Global Memory (DRAM) will be performed through memory banks which consecutive physical memories will be loaded at each load operation. Therefore, memory access pattern is essential for performance.

Shared memory can be accessed within thread blocks but the number of shared memory is limited. On the other hand, registers are available to thread at large amount and threads within a warp can exchange and communicate register values through **shuffle** operations. For TCUs, data are loaded, computed and stored through registers within a warp. Therefore, operation involved with TCUs are limited at warp level. [3]

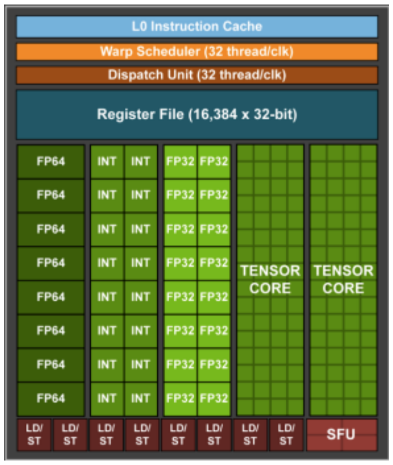

L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
TENSOR CORE
LD/ST
SFU

Figure 2: Structure of processing block

1.1.3 Main and GPU device memory of Xavier

On Jetson Xavier SOCs, CPU and GPU are gathered on one board where device memory is shared between them. In our experimental study, we both used traditional CudaMalloc & CudaMemcpy pattern and unified memory in profiling kernels.

1.2 Programming Interface

1.2.1 Tensor Core (WMMA)

We can directly program Tensor Cores in CUDA program through WMMA (WARP Matrix Multiply Accumulate) interface. The steps for usage are

1. Define input matrix A and B through wmma::fragment and specify the memory layout.
2. Define accumulator matrix to store computation result.
3. Load matrix A and B from memory location through wmma::load_matrix_sync. The function will load a square region across multiple rows.
4. accumulate result using wmma::mma_sync function to compute $D_{M\times N} = A_{M\times K}\cdot B_{K\times N} + C_{M\times N}$.

Notice that all the threads in a warp will call same sequence of functions. In kernel function, we should be aware of mapping from threads to warp.

1.2.2 Intra-Warp Communication

Safe intra-warp data communication is introduced in CUDA 9.0. By calling __shfl_sync() series of functions, threads within a warp can transmit value in registers. In addition, the shuffle sync function will synchronize threads in a warp to deal with the case of control divergence.

1.3 Parallel Algorithms

1.3.1 General Matrix Multiplication

General Matrix Multiplication will perform operation of

$$C = \alpha A \cdot B + \beta C$$

and many problems can be transformed into GEMM. In programming, matrix A, B and C are stored in consecutive array in row or column major layout

GEMM computation is memory intensive. Computation of one element in <M, N, K> setting will perform 3K memory load operation. To resolve this issue, tiled matrix multiplication is introduced where each thread block will load tiles into shared memory and produce out for a region. Using tiled matrix multiplication will increase arithmetic intensity by TILE_WIDTH directly.

Shared memory is limited and still result in access latency which stimulate us to come up with WARP matrix multiplication. With WARP matrix multiplication, each warp (instead

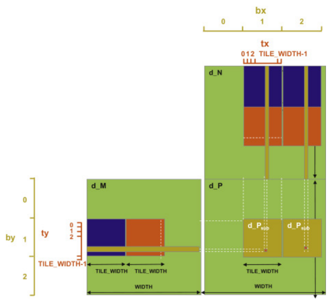


Figure 3: Tiled Matrix Multiplication

of thread block) will produce an output tile. Threads will load data into registers and compute several output elements. Data preloaded in registers are shared among threads in a warp using __shfl_sync() operations.

GEMM on Tensor Cores is an optimized WARP matrix multiplication which better memory access pattern and specialized processing unit to perform the computation.

1.3.2 Convolution

Convolution operation by be converted into a GEMM operation using image to column algorithm (im2col) [1]. The im2col algorithm will form input tensors and convolution filters into metrics separately. The reformed matrix has redundant data but can guarantee better memory access pattern for performance. Modern convolution kernels won't explicitly convert images into matrix but still apply GEMM computations based on input tiles.

1.3.3 Tensor Core Parallel Reduction

Reduction is a simple operation where we perform addition on all elements in an array. Tradition cuda kernels will accumulate values in several iterations at block level. With Tensor Cores, we can compute the sum of 256 input values in two cycles within a warp. We can form the accumulation of 256 values into two matrix multiplication.

$$\begin{bmatrix} a_{1,1} & a_{1,2}\dots & a_{1,15} & a_{1,16} \\ a_{2,1} & a_{2,2}\dots & a_{2,15} & a_{2,16} \\ \dots & & & \\ a_{16,1} & a_{16,2}\dots & a_{16,15} & a_{16,16} \end{bmatrix} \begin{bmatrix} 1 & 0 & \dots & 0 & 0 \\ 1 & 0 & \dots & 0 & 0 \\ \dots & & & & \\ 1 & 0 & \dots & 0 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} \sum a_{1,k} & 0 & \dots & 0 & 0 \\ \sum a_{2,k} & 0 & \dots & 0 & 0 \\ \dots & & & & \\ \sum a_{16,k} & 0 & \dots & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} \sum a_{1,k} & \sum a_{2,k} & \dots & \sum a_{15,k} & \sum a_{16,k} \\ 0 & 0 & \dots & 0 & 0 \\ \dots & & & & \\ 0 & 0 & \dots & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & \dots & 0 & 0 \\ 1 & 0 & \dots & 0 & 0 \\ \dots & & & & \\ 1 & 0 & \dots & 0 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} \sum a_{i,j} & 0 & ... & 0 & 0 \\ 0 & 0 & ... & 0 & 0 \\ ... \\ 0 & 0 & ... & 0 & 0 \end{bmatrix}$$

1.4 Required Questions

1.4.1 Key data structures

For GEMM, The key data structure is input tiles of matrix A and matrix B. By pre-loading data into shared memory or registers, we can significantly improve memory access pattern and increase performance.

1.4.2 Key operations

There are two major operations, memory load and multiplication. For non Tensor-Core implementation, data are loaded into shared memory or registers manually and perform computation on each thread. For Tensor-Core implementation, data are loaded into register using function load_matrix_sync and perform computation on Tensor Core

1.4.3 algorithm input and output

Both GEMM and parallel reduction will use random generated array. We will specify the layout (row major or column major) for input matrix of GEMM.

1.4.4 Part for Parallelization

The computation intensive section GEMM is to accumulate out tile values from input tiles. There are total TILE_WIDTH x TILE_WIDTH x TILE_WIDTH operations for each iterations. We can parallel the work using threads or directly apply tensor cores to perform the computation.

1.4.5 Dependency analysis

Both GEMM and parallel reduction have clear memory dependency pattern. In GEMM, computation of output tiles are fully independent from each other and we have full data-parallel pattern. In parallel reduction, each warp will work on specific region of input values then the work is also fully data parallel.

2 CUDA Algorithm Design

In this section, we will discuss and compare our algorithm design for WARP based GEMM and Tensor Core array reduction. The Tensor Core GEMM is similar to WARP based GEMM.

2.1 Warp based GEMM

For Warp based GEMM, we use register file instead shared memory for storing input tiles. For simplicity of programming, we put assign TILE_WIDTH element from input tile A (a partial row) and another TILE_width elements (a partial) from input tile B. In this way, we only need to exchange value once for every cycle of computation. Each warp will compute TILE_WIDTH 2 consecutive elements in output tile. The input are half value arrays for matrix A (column major) and matrix B (row major) allocated on GPU. The output value is array of half values for matrix C (row major) allocated on GPU.

The kernel for warp matrix multiplication is similar to shared memory based matrix multiplication excluding three major differences.

1. Input tiles will be directly loaded into registers pre allocated by each thread.
2. Each thread need to compute multiple output values depending on the ration of output tile size and warp size.
3. For each accumulative operation, each thread will need to acquire input value store in other threads through __shfl_sync() operation.

There are several key points to mention for performance during our implementation of algorithm.

1. Matrix A needs to be store in column major to allow coalesced memory access to global memory.
2. Matrix B needs to be store in row major to allow coalesced memory access.
3. Each thread will store TILE_WIDTH elements of matrix A to reduce data transmission.
4. Use unroll pragma for fixed loop to improve branch prediction.

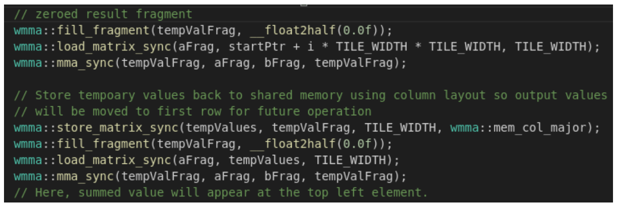
2.2 Tensor Core Reduction

We come up with this special reduction algorithm to show that Tensor Core can be used for parallel algorithm other than GEMM. The input data is half array allocated on GPU to sum up. Each warp will perform num_segment of partial array of length 256 (16 * 16). The sum result from each thread will be stored into another array allocated on GPU.

Also, we write this kernel show additional usage paradigm of Tensor Core Unit including

1. load_matrix_sync function can load data from either global memory or shared memory
2. store_matrix_sync function can store row major data into column major layout which automatically transpose the output tensor.
3. we could treat underlying storage of wmma::fragment as an array and access the elements through direct indexing.

The intuition based the algorithm is that we can sum 256 values in two rounds to tensor core multiplication and save as partial results in iterations. In addition, we can perform



```
// zeroed result fragment
wmma::fill_fragment(tempValFrag, __float2half(0.0f));
wmma::load_matrix_sync(aFrag, startPtr + i * TILE_WIDTH * TILE_WIDTH, TILE_WIDTH);
wmma::mma_sync(tempValFrag, aFrag, bFrag, tempValFrag);

// Store tempoary values back to shared memory using column layout so output values
// will be moved to first row for future operation
wmma::store_matrix_sync(tempValues, tempValFrag, TILE_WIDTH, wmma::mem_col_major);
wmma::fill_fragment(tempValFrag, __float2half(0.0f));
wmma::load_matrix_sync(aFrag, tempValues, TILE_WIDTH);
wmma::mma_sync(tempValFrag, aFrag, bFrag, tempValFrag);
// Here, summed value will appear at the top left element.
```

Figure 4: Core section of Wmma reduction

another two round of tensor core operations to sum up to 256 partial results. Each warp is able to process 65536 input elements.

In the implementation, we specifically use **heterogeneous** compute unit and memory hierarchy to improve performance. The cuda core will load and store data while tensor core is responsible for performing fast multiplication. In terms of memory, registers are used by tensor core fragments to perform computation while shared memory is used by us to cache intermediate results. The **heterogeneous** pattern can best utilize the resources and guarantee execution performance.

3 Approach and Experiment

We perform the implementation using CUDA language on Jetson Xavier NX IoT device with Linux system on aarch64 architecture. The device contain a 4-core CPU and GPU with 384 Cuda cores 48 Tensor Cores.

For all the algorithm implemented, we will use serial and openmp-cpu version for performance reference. Serial and openmp running time is measured through omp_get_wtime(). GPU kernel or whole workflow running is measured through cudaEvent_t and cudaEventRecord()

3.1 Serial version GEMM

For performance comparison, we implement GEMM on CPU with OpenMp. We only parallelize the outer loop to avoid critical section. Also, we implement a transposed version where matrix B is transposed to column major layout to improve memory access pattern and cache locality. The transposed version indicates better total execution time.

3.2 GPU GEMM and basic Tiled GEMM

We then comes to implement traditional GEMM algorithm on CUDA using fp32 input type and fp32 output type. The basic version version will map each thread to compute one output element and all threads will work independently. For example, for a <M, N, K> setting, we will launch with dimGrid(M x N / 256, 1, 1) and dimBlock(256, 1, 1) to compute the output.

For Basic Tiled GEMM, we use TILE_WIDTH of 16. Each thread block is mapped to a TILE_WIDTH x TILE_WIDTH output region. Threads will need to detect which output region it maps to. We follow the regular pattern to assign shared memory for input tiles (TileA and TileB) with shape TILE_WIDTH x TILE_WIDTH. The thread block will iterative on K dimension and store computed value in register. The launch configuration is same as basic GEMM

3.3 Porting kernels to fp16

We move above two kernels to use fp16 with same thread block mapping. The memory access pattern stay the same but all the math operations need to be done through half math operation provided through <cuda_fp16.h>

3.4 Warp based GEMM

Inspired by Tensor Core Unit's mechanism, we implement two kernels to assign a warp for each output tile. The first kernel will use shared memory as intermediate storage for input tiles. The second will use registers on thread to cache input tiles and communicates through __shfl_sync() operation to exchange values.

3.4.1 Shared memory based

As a intermediate implementation, I put input tiles into shared memory. Each warp will compute a output tile of 16x16 shape. Therefore, each thread will map to 8 output elements. The iteration process on K dimension is same as block level tiling. For threads mapping, I use thread block of 128 threads (4 warps) with gridDim(m / (2 * TILE_WIDTH), n / (2 * TILE_WIDTH), 1) and blockDim(2 * WARP_SIZE, 2, 1).

Each thread block will map to 4 (16x16) output tiles. Therefore we need to allocate 4 times of shared memory per block compared to thread-block tiling algorithm. __syncWarp() function is used to synchronize threads inside warp.

3.4.2 Register Tiling based

Register tiling warp GEMM on the other hand, load input tiles into registers for each thread. The thread mapping is same as shared memory based warp-gemm kernel. For each computation threads within a warp will exchange values through shuffle sync interface.

3.5 Simple Tensor Core GEMM

Then we upgrade the algorithm to use Tensor Core for GEMM computation. We use nvcuda::wmma:: functions to program the tensor core. The threads block and grid setting is the same as warp based gemm. However, with Tensor Core and wmma functions, i don't need to manually allocate registers and load data into registers. Data are loaded directly through load_matrix_sync and multiplication is performed using tensor cores.

Table 1: Exeuction time of GEMM implementations on CPU and GPU in ms

methods	cpu-serial	cpu-openmp	basic-fp32	shmem fp32	basic-fp16	warp-shmem fp16	tcus fp16
(64, 64, 64)	3	1	0.06	0.022	0.06	0.04	0.052
(128, 128, 128)	20	6	0.11	0.087	0.18	0.09	0.037
(256, 256, 256)	190	72	0.672	0.478	0.71	0.42	0.12
(512, 512, 512)	1475	403	5.16	3.52	5.03	3.1	0.73
(1024, 1024, 1024)	11742	2974	38.36	23.27	39.36	26.93	5.59

With Tensor Core, we can reach best performance compared to other implementations.

3.6 Tensor Core Reduction

We implement the Tensor Core reduction mentioned in the design section. The reduction algorithm will launch 48 warps with each process 256 * 256 elements. Each thread block will contain one warp for simplicity of programming.

The innovation point of this kernel is that we eliminate the number of manual memory operations (ex. transpose temporary result matrix) which guarantee a better memory access pattern. Additionally, we choose two kernel launch strategy. In first method, we use traditional cudaMalloc and cudaMemcpy pattern which impose notable delays for memory transfer. For the second method, we using unified memory (cudaMallocManaged) for the kernel. Jetson Nx have single memory space for bost CPU and GPU which result in significant improvement in whole workflow execution time.

4 Result and Analysis

4.1 Runtime result of GEMM

In the experiment we run and measure GEMM algorithm on CPU and GPUs on jetson nx device and summarize the result into table1. (notice register based warp GEMM is not included in evaluation study due to incomplete verification).

From the execution time tables we can clearly observe that GPU on jetson is significantly better than CPU in performing GEMM computation. We can notice that even the most naive CUDA kernel can perform 306 times better than single core CPU and 70 times better than basic openmp version. Although we didn't record memory transfer time and kernel launch time into the measurement, the speedup is still significant from cpu to gpu on gemm operations.

4.1.1 Performance of shared memory fp32

On the other hand, we can notice that the share memory based tilling algorithm doesn't provide significant improvement in terms of performance. The speedup we gain from using shared memory and tiling algorithm is about 2x speedup. For every experiment shape, we plot the speedup of shared memory tiling implementation.

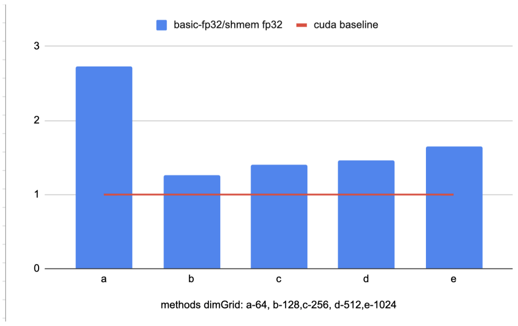


Figure 5: speedup result of shared memory tiling

This is expected due to Volta architecture's update where L1 cache has been improved to minimize the difference automatic data cache and manual shared memory setup. Therefore, we only observe 2x performance speedup [2].

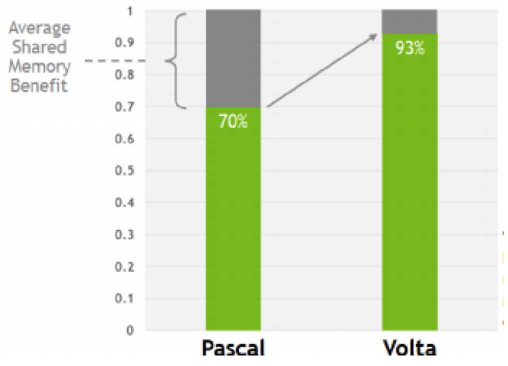


Figure 6: Volta Arch L1 Cache vs ShareMem

4.1.2 Performance comparison of Fp16 implementation

For the Fp16 implementations, we can observe that Tensor Core implementation can significantly improve the performance. We can observe that using Tensor Core can bring around 6 times of speedup in performance. On the one hand, using tensor core can significantly speedup the accumulation process output tiles. On the other hand, wmma:: memory load function optimize the memory access pattern from device memory so that it can significantly reduce memory wait time when compute GEMM.

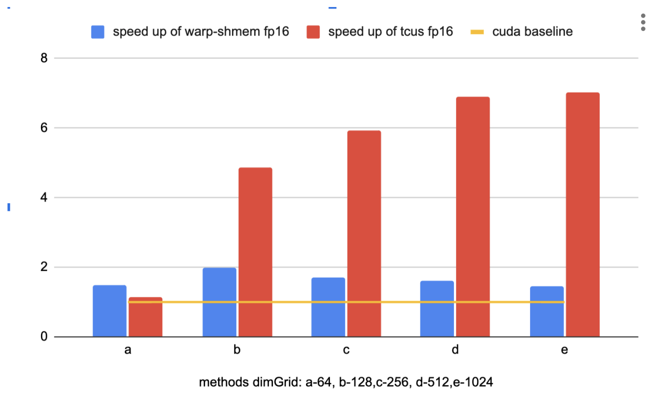


Figure 7: speedup result of fp16 kernels

4.2 Selected NvProf Analysis of GEMM kernels

4.2.1 Performance issue of fp32 vs fp16

I implemented basic GEMM kernels in both fp32 and fp16. Originally, I expect that fp16 mode will provide 2x performance. However, the result indicates that there is no performance improvement for this change. I examine the nvprof information and found the source of problem is: **Basic Gemm kernel issued too much memory requests.** The system needs to fullfill the memory requests in order and there are significantly queueing delay.

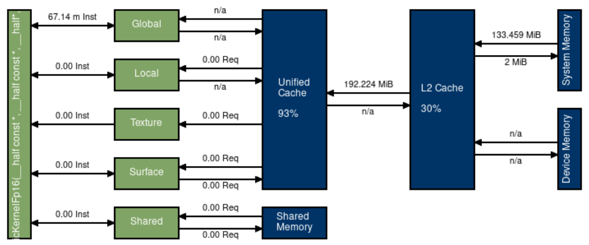


Figure 8: Memory Statistics of fp16 basic GEMM

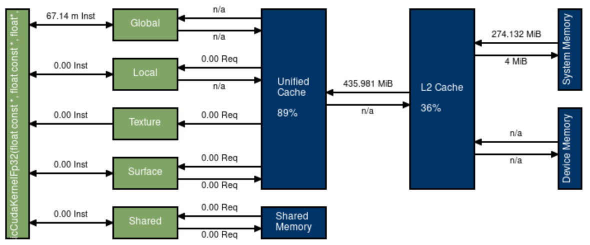


Figure 9: Memory Statistics of fp32 basic GEMM

From two memory statistics figures, we can compare the data transfer rate between system memory and L2 cache. The data transfer rate of fp32 is about twice as fp16. That is, although individual memory request size of fp16 is only half of fp32, there are two many memory request to execute which block the memory banks. The result is the data transfer rate of fp16 kernel is only half of fp32.

4.2.2 Limit factor of shared memory Warp GEMM

The first warp level GEMM kernel is based on shared memory. The kernel doesn't reach optimal performance because shared memory usage is too high. Each thread block will contain 128 threads but it needs to allocate 4 times more shared memory per block to store intermediate input tiles. Then the limiting factor becomes shared memory usage which lower the warp occupancy in computation.

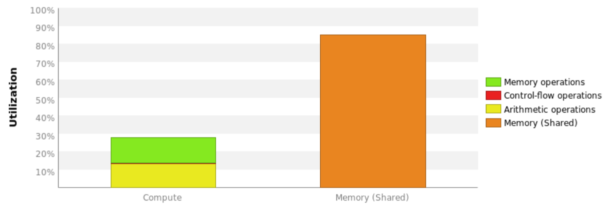


Figure 10: Utilization statistics of fp16 shareMem warp GEMM

4.2.3 Limit factor of Tensor Core implementation

The Tensor Core GEMM kernel has highly optimized memory access and execution pattern. The Tensor Core unit can significantly improve the computation speed than mmeory will become the prominent factor influencing the performance. From the stall reason graph we can clearly notice the impact of memory wait time.

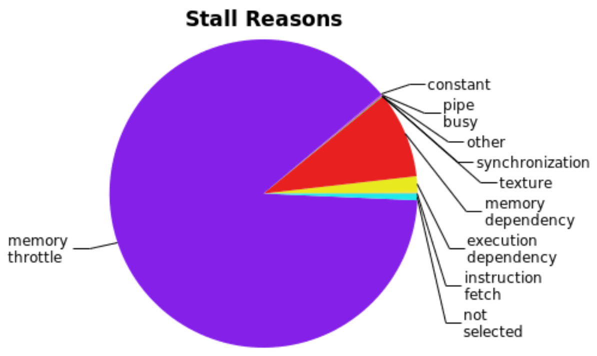


Figure 11: Stall Reason of Tensor Core GEMM kernel

4.3 Runtime result of Reduction

The runtime of GPU implementation is measured through the whole workflow including data transfer (existed for using normal device memory).

Table 2: Execution time for reduction in ms

method	cpu-ref	TCU redu	TCU redu (unified Mem)
time	24.6	6.7	2.4

From the performance result we can observe that GPU can not provide significant speedup compared to GEMM. The reason is that computation is not intensive for reduction operation. Also, we can notice that cudaMemcpy and related memory operation can significantly impcat the whole-workflow performance. However, with unified memory, the performance is significantly better.

The advantage of shared host and GPU memory space is notable in this reduction application.

References

[1] KIRK, D. B., AND HWU, W.-M. W. *Programming Massively Parallel Processors, Third Edition: A Hands-on Approach*, 3rd ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2016.

[2] NVIDIA. volta-architecture-whitepaper.

[3] YUAN LIN, V. G. Using cuda warp-level primitives.