Matrix Multiplication Background

User's Guide | NVIDIA Docs

DU-09799-001_v001 | February 2023

Table of Contents

List of Figures

List of Tables

Chapter 1. Background: Matrix-Matrix Multiplication

GEMMs (General Matrix Multiplications) are a fundamental building block for many operations in neural networks, for example fully-connected layers, recurrent layers such as RNNs, LSTMs or GRUs, and convolutional layers. In this guide, we describe GEMM performance fundamentals common to understanding the performance of such layers.

GEMM is defined as the operation $C = \alpha \mathrm{AB} + \beta C$, with *A* and *B* as matrix inputs, α and β as scalar inputs, and *C* as a pre-existing matrix which is overwritten by the output. A plain matrix product *AB* is a GEMM with α equal to one and β equal to zero. For example, in the forward pass of a fully-connected layer, the weight matrix would be argument *A*, incoming activations would be argument *B*, and α and β would typically be 1 and 0, respectively. β can be 1 in some cases, for example, if we're combining the addition of a skip-connection with a linear operation.

Chapter 2. Math And Memory Bounds

Following the convention of various linear algebra libraries (such as BLAS), we will say that matrix A is an `M x K` matrix, meaning that it has M rows and K columns. Similarly, B and C will be assumed to be `K x N` and `M x N` matrices, respectively.

The product of A and B has `M x N` values, each of which is a dot-product of K-element vectors. Thus, a total of `M * N * K` fused multiply-adds (FMAs) are needed to compute the product. Each FMA is 2 operations, a multiply and an add, so a total of `2 * M * N * K` FLOPS are required. For simplicity, we are ignoring the α and β parameters for now; as long as K is sufficiently large, their contribution to arithmetic intensity is negligible.

To estimate if a particular matrix multiply is math or memory limited, we compare its arithmetic intensity to the `ops:byte` ratio of the GPU, as described in Understanding Performance. Assuming an NVIDIA® V100 GPU and Tensor Core operations on FP16 inputs with FP32 accumulation, the `FLOPS:B` ratio is 138.9 if data is loaded from the GPU's memory.

$$\text{Arithmetic Intensity} = \frac{\text{number of FLOPS}}{\text{number of byte accesses}} = \frac{2\cdot(M\cdot N\cdot K)}{2\cdot(M\cdot K+N\cdot K+M\cdot N)} = \frac{M\cdot N\cdot K}{M\cdot K+N\cdot K+M\cdot N}$$

As an example, let's consider a `M x N x K = 8192 x 128 x 8192` GEMM. For this specific case, the arithmetic intensity is `124.1 FLOPS/B`, lower than V100's `138.9 FLOPS:B`, thus this operation would be memory limited. If we increase the GEMM size to 8192 x 8192 x 8192 arithmetic intensity increases to 2730, much higher than `FLOPS:B` of V100 and therefore the operation is math limited. In particular, it follows from this analysis that matrix-vector products (general matrix-vector product or GEMV), where either `M=1` or `N=1`, are always memory limited; their arithmetic intensity is less than 1.

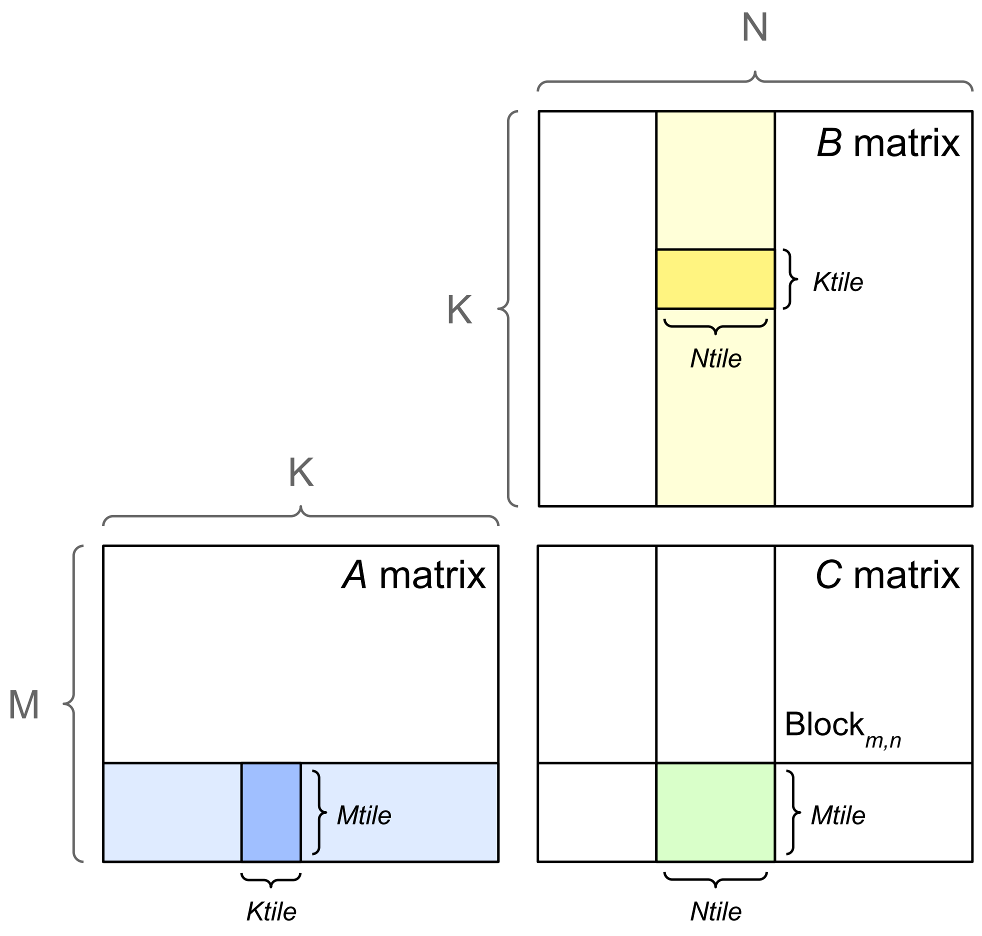
It is worth keeping in mind that the comparison of arithmetic intensity with the ops:byte ratio is a simplified rule of thumb, and does not consider many practical aspects of implementing this computation (such as non-algorithm instructions like pointer arithmetic, or the contribution of the GPU's on-chip memory hierarchy).

2.1. GPU Implementation

GPUs implement GEMMs by partitioning the output matrix into tiles, which are then assigned to thread blocks.

Tile size, in this guide, usually refers to the dimensions of these tiles (*Mtile* x *Ntile* in Figure 1). Each thread block computes its output tile by stepping through the K dimension in tiles, loading the required values from the A and B matrices, and multiplying and accumulating them into the output.

Figure 1. Tiled outer product approach to GEMMs



2.2. Tensor Core Requirements

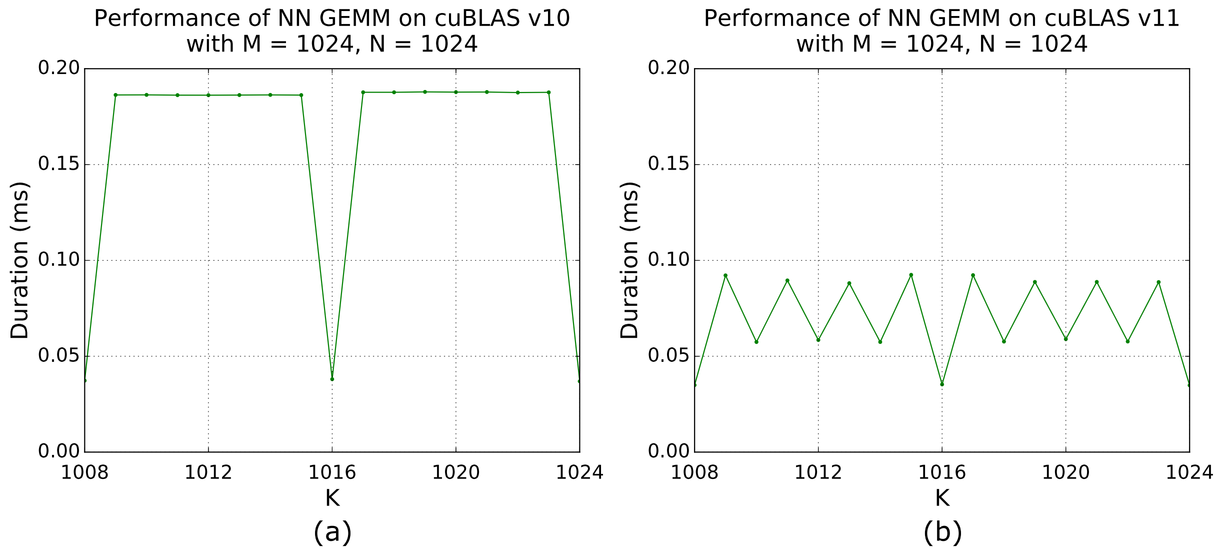
As we discussed in GPU Architecture Fundamentals, the latest NVIDIA GPUs have introduced Tensor Cores to maximize the speed of tensor multiplies. Requirements to use Tensor Cores depend on NVIDIA library versions. Performance is better when equivalent matrix dimensions M, N, and K are aligned to multiples of 16 bytes (or 128 bytes on A100). With NVIDIA cuBLAS versions before 11.0 or NVIDIA cuDNN versions before 7.6.3, this is a requirement to use Tensor Cores; as of cuBLAS 11.0 and cuDNN 7.6.3, Tensor Cores may be used regardless, but efficiency is better when matrix dimensions are multiples of 16 bytes. For example, when using FP16 data, each FP16 element is represented by 2 bytes, so matrix dimensions would need to be multiples of 8 elements for best efficiency (or 64 elements on A100).

Table 1. Tensor Core requirements by cuBLAS or cuDNN version for some common data precisions. These requirements apply to matrix dimensions M, N, and K.

Tensor Cores can be used for...	**cuBLAS version < 11.0** **cuDNN version < 7.6.3**	**cuBLAS version ≥ 11.0** **cuDNN version ≥ 7.6.3**
INT8	Multiples of 16	Always but most efficient with multiples of 16; on A100, multiples of 128.
FP16	Multiples of 8	Always but most efficient with multiples of 8; on A100, multiples of 64.
TF32	N/A	Always but most efficient with multiples of 4; on A100, multiples of 32.
FP64	N/A	Always but most efficient with multiples of 2; on A100, multiples of 16.

The requirement is in fact more relaxed - only the fastest varying dimensions in memory are required to obey this rule - but it is easiest to just think of all three dimensions the same way. Following these alignments for all dimensions ensures Tensor Cores will be enabled and run efficiently. This effect can be seen in Figure 5 - calculations are fastest (durations are lowest) when K is divisible by 8. When K is not divisible by 8, switching from cuBLAS 10.2 to cuBLAS 11.0 allows Tensor Cores to be used and results in 2-4x speedup. It is also worth noting that with cuBLAS 11.0, among values of K that are not divisible by 8, even values still result in faster calculation than odd values. We recommend choosing matrix dimensions to be multiples of 16 bytes (8 for FP16 as in Table 1); if this is not possible, choosing multiples of a smaller power of two (such as 8 or 4 bytes) often still helps performance with cuBLAS 11.0 and higher. On A100, choosing multiples of larger powers of two up to 128 bytes (64 for FP16) can further improve efficiency.

Figure 2. Comparison of GEMM execution times with (a) cuBLAS 10.1 and (b) cuBLAS 11.0, both with FP16 data. Calculation is fastest (duration is lowest) when K is divisible by 8. “NN” means A and B matrices are both accessed non-transposed. NVIDIA V100-DGXS-16GB GPU.



2.3. Typical Tile Dimensions In cuBLAS And Performance

The cuBLAS library contains NVIDIA’s optimized GPU GEMM implementations (refer to here for documentation).

While multiple tiling strategies are available, larger tiles have more data reuse, allowing them to use less bandwidth and be more efficient than smaller tiles. On the other hand, for a problem of a given size, using larger tiles will generate fewer tiles to run in parallel, which can potentially lead to under-utilization of the GPU. When frameworks like TensorFlow or PyTorch call into cuBLAS with specific GEMM dimensions, a heuristic inside cuBLAS is used to select one of the tiling options expected to perform the best. Alternatively, some frameworks provide a “benchmark” mode, where prior to the training they time all implementation choices and pick the fastest one (this constitutes a once per training session overhead).

This tradeoff between tile efficiency and tile parallelism suggests that the larger the GEMM, the less important this tradeoff is: at some point, a GEMM has enough work to use the largest available tiles and still fill the GPU. Conversely, if a GEMM is too small, the reduction in either tile efficiency or tile parallelism will likely prevent the GPU from running at peak math utilization. Figure 3 and Figure 4 illustrate this general trend; larger GEMMs achieve higher throughput.

Figure 3. Performance improves as the M-N footprint of the GEMM increases. Duration also increases, but not as quickly as the M-N dimensions themselves; it is sometimes possible to increase the GEMM size (use more weights) for only a small increase in duration. NVIDIA A100-SXM4-80GB, CUDA 11.2, cuBLAS 11.4.

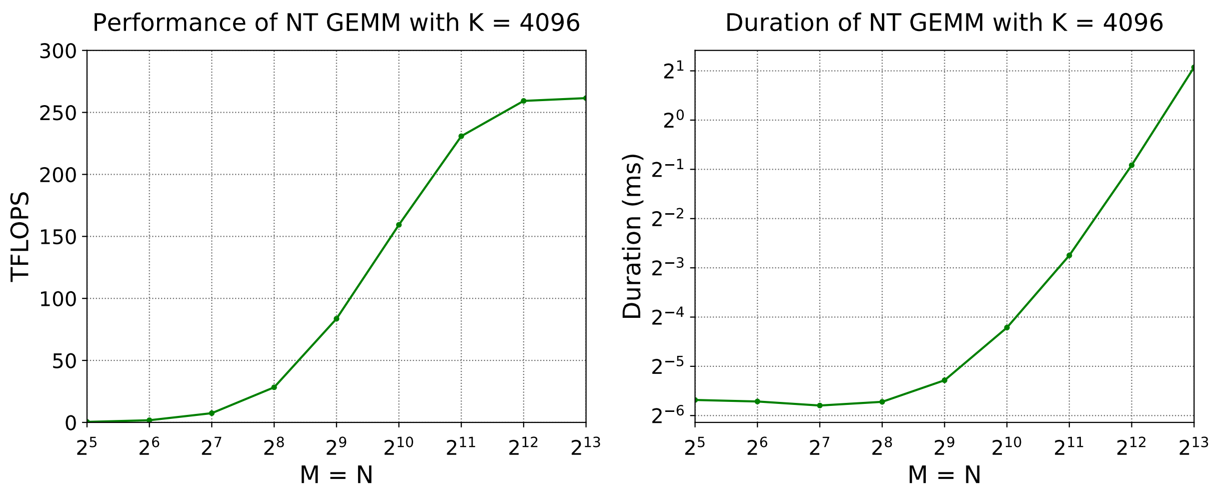
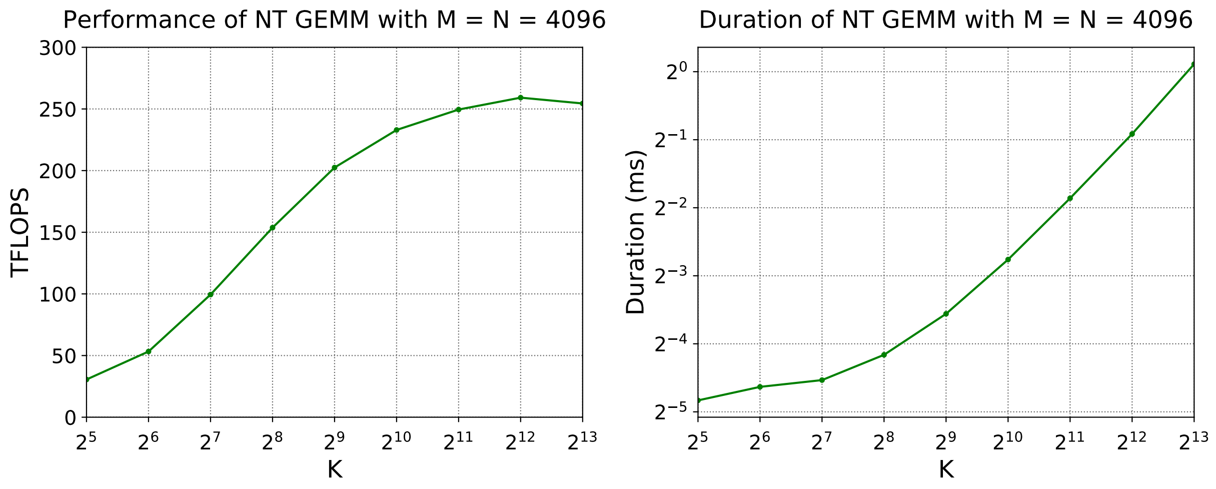


Figure 4. Performance improves as the K dimension increases, even when M=N is relatively large, as setup and tear-down overheads for the computation are amortized better when the dot product is longer. NVIDIA A100-SXM4-80GB, CUDA 11.2, cuBLAS 11.4.



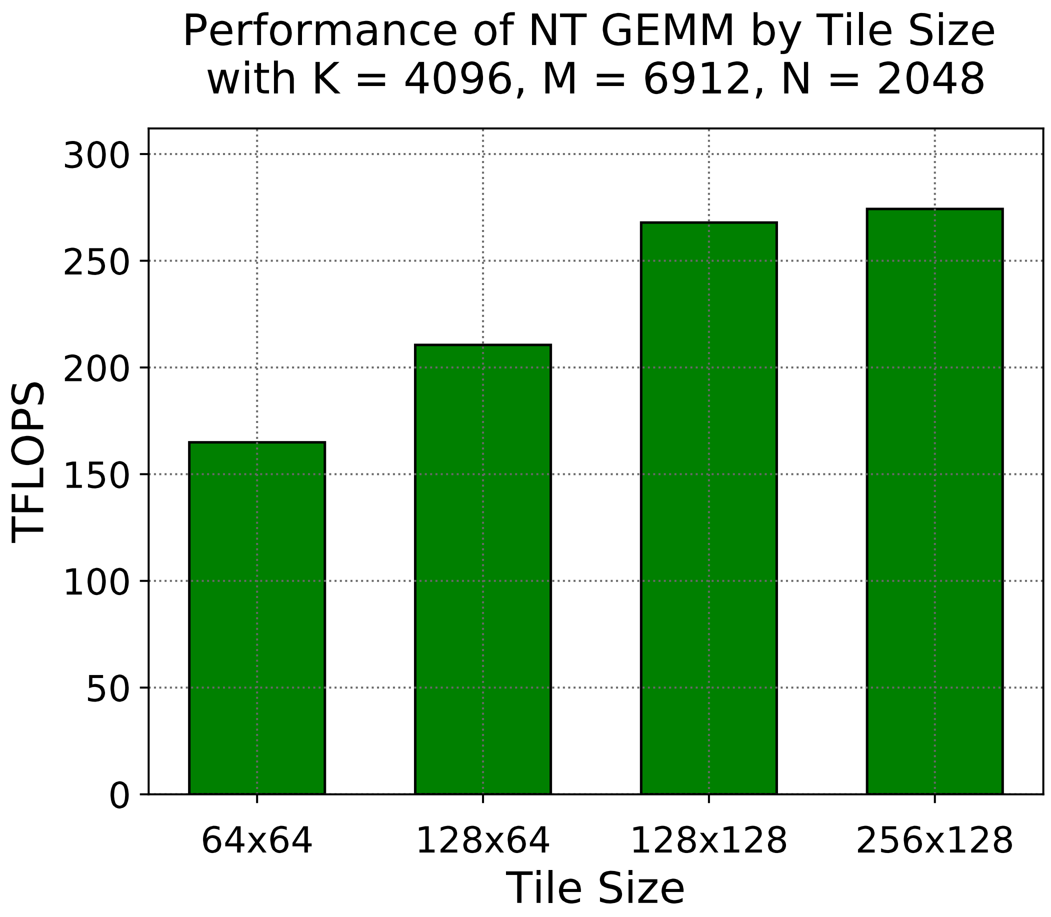
For cuBLAS GEMMs, thread block tile sizes typically but not necessarily use power-of-two dimensions. Different tile sizes might be used for different use cases, but as a starting point, the following tiles are available:

- 256x128 and 128x256 (most efficient)

- 128x128
- 256x64 and 64x256
- 128x64 and 64x128
- 64x64 (least efficient)

Figure 5 shows an example of the efficiency difference between a few of these tile sizes:

Figure 5. Larger tiles run more efficiently. The 256x128-based GEMM runs exactly one tile per SM, the other GEMMs generate more tiles based on their respective tile sizes. NVIDIA A100-SXM4-80GB, CUDA 11.2, cuBLAS 11.4.



The chart shows the performance of a `MxNxK = 6912x2048x4096` GEMM with different tile sizes. It demonstrates that the increased tile parallelism with smaller tiles (64x64 enables 8x more parallelism than 256x128) comes at a notable efficiency cost. In practice, cuBLAS will avoid using small tiles for GEMMs that are large enough to have sufficient parallelism with larger tiles and will resort to the smaller ones only when substantially smaller GEMMs than the one in this example are being run. As a side note, NVIDIA libraries also have the ability to "tile" along the K dimension in case both M and N are small but K is large. Because K is the direction of the dot product, tiling in K requires a reduction at the end, which can limit achievable performance. For simplicity, most of this guide assumes no K tiling.

Chapter 3. Dimension Quantization Effects

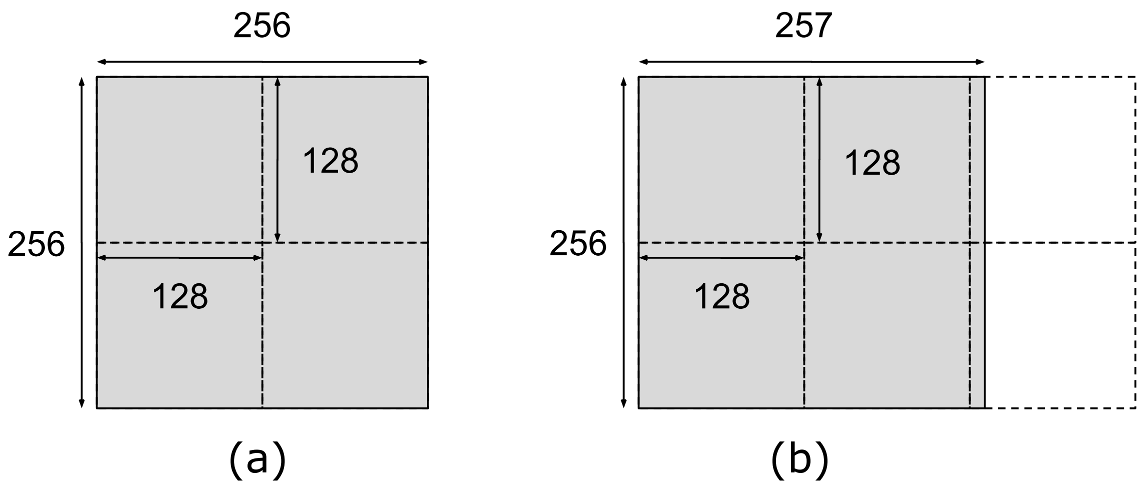
As described in GPU Execution Model, a GPU function is executed by launching a number of thread blocks, each with the same number of threads. This introduces two potential effects on execution efficiency - tile and wave quantization.

3.1. Tile Quantization

Tile quantization occurs when matrix dimensions are not divisible by the thread block tile size.

The number of thread block tiles is large enough to make sure all output elements are covered, however, some tiles have very little actual work as illustrated in Figure 6, which assumes 128x128 tiles and two matrix dimensions.

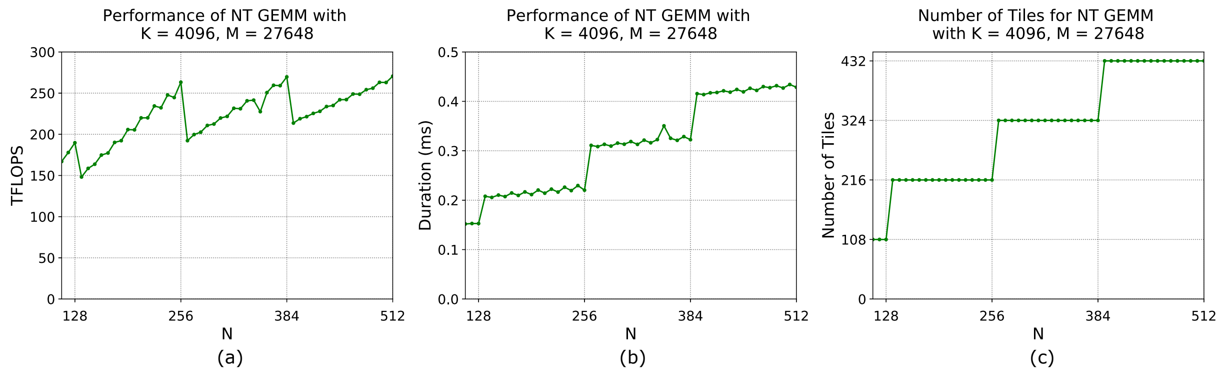
Figure 6. Example of tiling with 128x128 thread block tiles. (a) Best case - matrix dimensions are divisible by tile dimensions (b) Worse case - tile quantization results in six thread blocks being launched, two of which waste most of their work.



While libraries ensure that invalid memory accesses are not performed by any of the tiles, all tiles will perform the same amount of math. Thus, due to tile quantization, the case in Figure 6 (b) executes 1.5x as many arithmetic operations as Figure 6 (a) despite needing only 0.39% more operations algorithmically. As this shows, the highest utilization is achieved when output matrix dimensions are divisible by tile dimensions.

For another example of this effect, let's consider GEMM for various choices of N, with `M = 27648`, `K = 4096`, and a library function that uses 256x128 tiles. As N increases from 136 to 256 in increments of 8, the Tensor Core accelerated GEMM always runs the same number of tiles, meaning the N dimension is always divided into 2 tiles. While the number of tiles remains constant, the fraction of those tiles containing useful data and hence the number of useful FLOPS performed an increase with N, as reflected by the GFLOPS in Figure 7 below. Notice that throughput reduces significantly between `N = 128` (where the single tile per row is filled with useful data) and `N = 136` (where a second tile is added per row but contains only 8/128 = 6.25% useful data). Also, note how the duration is constant whenever the number of tiles is constant.

Figure 7. Tile quantization effect on (a) achieved FLOPS throughput and (b) elapsed time, alongside (c) the number of tiles created. Measured with a function that forces the use of 256x128 tiles over the MxN output matrix. In practice, cuBLAS would select narrower tiles (for example, 64-wide) to reduce the quantization effect. NVIDIA A100-SXM4-80GB, CUDA 11.2, cuBLAS 11.4.



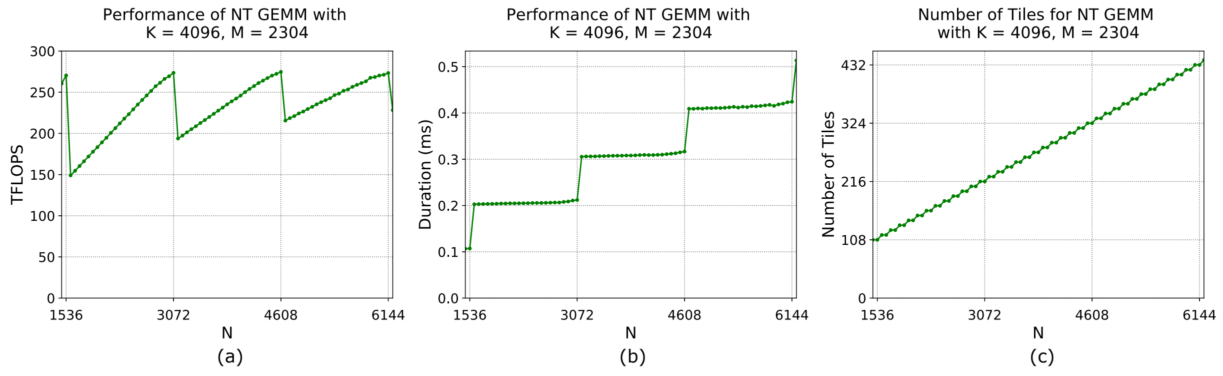
3.2. Wave Quantization

While tile quantization means the problem size is quantized to the size of each tile, there is a second quantization effect where the total number of tiles is quantized to the number of multiprocessors on the GPU: Wave quantization.

Let's consider a related example to the one before, again varying N and with `K = 4096`, but with a smaller `M = 2304`. An NVIDIA A100 GPU has 108 SMs; in the particular case of 256x128 thread block tiles, it can execute one thread block per SM, leading to a wave size of 108 tiles that can execute simultaneously. Thus, GPU utilization will be highest when the number of tiles is an integer multiple of 108 or just below.

The M dimension will always be divided into 2304/256 = 9 tiles per column. When `N = 1536`, the N dimension is divided into 1536/128 = 12 tiles per row, and a total of 9*12 = 108 tiles are created, comprising one full wave. When `1536 < N <= 1664`, an additional tile per row is created for a total of 9*13 = 117 tiles, leading to one full wave and a 'tail' wave of only 9 tiles. The tail wave takes nearly the same time to execute as the full 108-tile wave in this example but uses only 9/108 = 8.33% of A100's SMs during that time. Consequently, GFLOPS roughly halve and duration roughly doubles from `N = 1536` to `N = 1544` (Figure 8). Similar jumps can be seen after `N = 3072`, `N = 4608`, and `N = 6144`, which also map to an integer number of full waves.

Figure 8. The effects of wave quantization in terms of (a) achieved FLOPS throughput and (b) elapsed time, as well as (c) the number of tiles created. Measured with a function that uses 256x128 tiles over the MxN output matrix. Note that the quantization effect occurs when the number of tiles passes a multiple of 108. NVIDIA A100-SXM4-80GB, CUDA 11.2, cuBLAS 11.4.



It is worth noting that the throughput and duration graphs for wave quantization look very similar to those for tile quantization, except with a different scale on the horizontal axis. Because both phenomena are quantization effects, this is expected. The difference lies in where the quantization occurs: tile quantization means work is quantized to the size of the tile, whereas wave quantization means work is quantized to the size of the GPU. Figure 7 (c) and Figure 8 (c) in both the tile and wave quantization illustrations show this difference.

NVIDIA Corporation | 2788 San Tomas Expressway, Santa Clara, CA 95051
www.nvidia.com