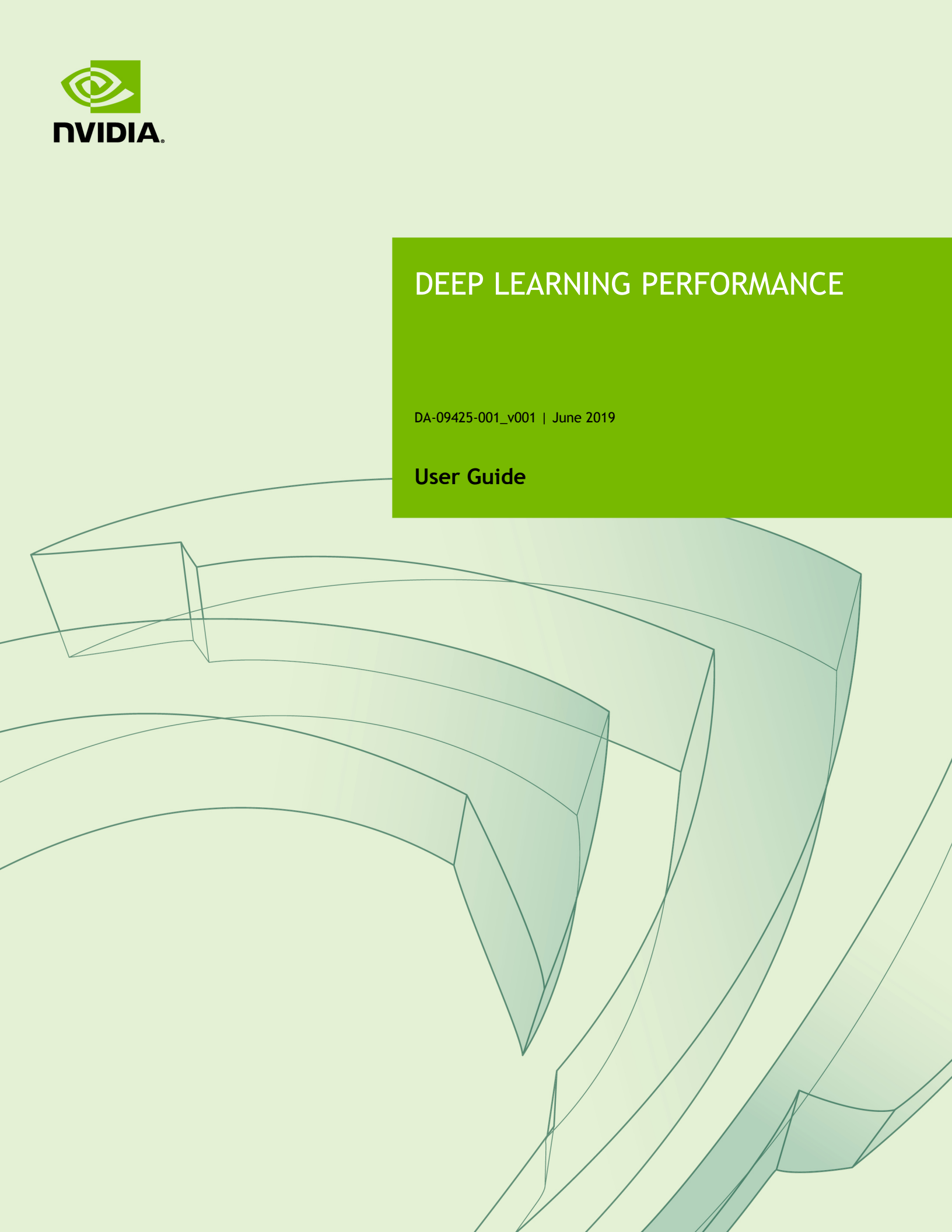

DEEP LEARNING PERFORMANCE

DA-09425-001_v001 | June 2019

User Guide

TABLE OF CONTENTS

Chapter 1. INTRODUCTION

This guide describes and explains the impact of parameter choice on the performance of various types of neural network layers commonly used in state-of-the-art deep learning applications. It focuses on GPUs that provide Tensor Core acceleration for deep learning (NVIDIA Volta architecture or more recent).

First, we provide a Performance Guidelines: Summary you can read in 5 minutes that, if followed, will set you up for great Tensor Core performance. Following the checklist, we dive into the detailed sections of the guide, starting with an introduction to the high-level basics of GPU performance. Then, we cover various common neural network operations - for example fully-connected layers, convolutions, or batch normalization - in their own sections. Each operation section briefly describes the basics of the operation's performance, as well as the impact of each configuration parameter on performance, providing data to illustrate the performance behavior.

The guide is intended to be used as a reference; sections link to background as necessary, but are standalone wherever possible. An understanding of some underlying principles, including matrix multiplication, is assumed.

1.1. How We Measure Performance

In this guide, when we talk about *performance* we mean operation speed, as opposed to model accuracy (which some refer to as task performance). We compare operation performance with two metrics: duration (in milliseconds) and math processing rate (or *throughput*), which we simply refer to as 'performance' (in floating point operations per second, or FLOPS). Graphs throughout the guide use the metric that most clearly represents a trend.

The simplest measurement of performance is duration: how long it takes to complete a given operation. While measuring duration is intuitive, it is difficult to make a meaningful performance comparison between operations of very different sizes, and that we consequently expect to have very different durations. An alternative way to think about performance is in terms of throughput: how many operations can be done per unit time. Performing one inference step on a layer with a batch size of 1024 will naturally take longer than doing the same for a layer with a batch size of 1. However, using a larger batch size means that more calculations are being done; usually, this also

means that there are more opportunities to parallelize the operation. Consequently, while the larger operation takes more time to complete, it is usually also more efficient: one step with a batch size of 1024 will be faster than 1024 steps with a batch size of 1. This improvement is difficult to see in duration, but immediately visible in throughput.

Throughout this guide, unless otherwise stated, all performance examples assume 16-bit input and output data with 32-bit accumulation and Tensor Cores enabled for fast processing. Tensor inputs and outputs are assumed to be in NHWC format unless otherwise stated. All performance data has been averaged over multiple runs to reduce variance. For convolutions specifically, data represents cross-correlation performance.

GPUs use dynamic voltage/frequency scaling to adjust clock frequencies based on the power consumed by a workload. For all performance data in this guide, input data and weights were generated randomly from a normal distribution, which leads to high toggling rates in the GPU's hardware units, consumes the most power, and results in conservative clock frequencies. While recording data, the GPU frequency was locked to the maximum frequency (while still obeying the power cap) using **`nvidia-smi --lock-gpu-clocks`** to ensure result consistency.

1.2. How This Guide Fits In

NVIDIA's GPU deep learning platform comes with a rich set of other resources you can use to learn more about NVIDIA's Tensor Core GPU architectures as well as the fundamentals of mixed-precision training and how to enable it in your favorite framework.

The Tesla V100 GPU architecture whitepaper provides an introduction to Volta, the first NVIDIA GPU architecture to introduce Tensor Cores to accelerate Deep Learning operations. The equivalent whitepaper for the Turing architecture expands on this by introducing Turing Tensor Cores, which add additional low-precision modes.

The Training With Mixed Precision Guide describes the basics of training neural networks with reduced precision such as algorithmic considerations following from the numerical formats used. It also details how to enable mixed precision training in your framework of choice, including TensorFlow, PyTorch, and MxNet. The easiest and safest way to turn on mixed precision training and use Tensor Cores is through Automatic Mixed Precision, which is supported in PyTorch, TensorFlow, and MxNet.

Chapter 2. PERFORMANCE GUIDELINES: SUMMARY

This section presents a quick summary of the most important guidelines you can follow to improve performance. If you want to jump straight to optimizing a network, read this section! The rest of the document goes into more depth about the details of these guidelines and why they are important; we link to sections with additional information where appropriate.

When discussing routines commonly used for deep learning, we make a distinction between math-limited and bandwidth-limited operations. **Math-limited** routines are ones where performance is limited by calculation rate rather than memory bandwidth. Some examples typically in this category are fully-connected layers, convolutional layers, and recurrent layers, which involve many calculations per input and output value. Tensor Cores are designed to accelerate these routines. Conversely, **bandwidth-limited** routines involve relatively few calculations per input and output value, and their speed is consequently nearly always limited by memory bandwidth. Most other types of operations, including activation functions, pooling, and batch normalization, are in this category. Tensor Cores are not used for these routines, as speeding up calculation rate will not improve their performance.

Which operations are in each category depends on the parameters of the operation and on the accelerator device used. If calculation rate is high enough, any routine will become limited by memory bandwidth. Additionally, routines that are typically math-limited can be defined with parameters that cause them to be bandwidth-limited instead. We discuss this in more detail in GPU Performance Background, and more specifically, Understanding Performance.

The following suggestions apply primarily to math-limited deep learning routines. More specific recommendations for individual types of layers, both math-limited and bandwidth limited, follow at the end of the section.

2.1. Enable Tensor Cores

Tensor Cores are designed to significantly accelerate deep learning operations; we recommend choosing parameters that enable them whenever possible.

Tensor Cores are used if key parameters of the operation are divisible by 8 if operating on FP16 data or 16 if using INT8 data. For fully-connected layers, this means choosing batch size and the number of inputs and outputs to be multiples of 8 (or 16); for convolutional layers, only the number of input and output channels need to be divisible by 8 (or 16). This requirement is based on how data is stored and accessed in memory; further details can be found in Tensor Core Requirements.

2.2. Choose Parameters To Maximize Execution Efficiency

GPUs perform operations efficiently by dividing the work between many parallel processes. Consequently, using parameters that make it easier to break up the operation evenly will lead to the best efficiency.

This means choosing parameters (including batch size, input size, output size, and channel counts) to be divisible by larger powers of two, at least 64, and up to 256. There is no downside to using values divisible by powers of two above 256, but there is less additional benefit. More specific requirements for different routines can be found at the end of this section. Background on why this matters can be found in GPU Architecture Fundamentals and Typical Tile Dimensions In cuBLAS And Performance.

2.3. Operate In Math-Limited Regime Where Possible

GPUs excel at performing calculations in parallel, but data movement is expensive and has a strict maximum rate. If the speed of a routine is limited by calculation rate, performance can be improved by enabling Tensor Cores and following our other recommendations. On the other hand, if the speed of a routine is limited by the rate of data movement, these tweaks are ineffective; performance will be determined by how long it takes to load inputs and store outputs.

For fully-connected and convolutional layers, this occurs mostly when one or more parameters of a layer are small; see Understanding Performance and Math And Memory Bounds for background and details.

2.4. Routine-Specific Recommendations

The following general recommendations can be applied to particular routines.

For fully-connected layers:

- Choose batch size and the number of inputs and outputs to be divisible by 8 (FP16) / 16 (INT8) to enable Tensor Cores; see Tensor Core Requirements.
- Especially when one or more parameters are small, choosing batch size and the number of inputs and outputs to be divisible by at least 64 and ideally 256 can streamline tiling and reduce overhead; see Dimension Quantization Effects.
- Larger values for batch size and the number of inputs and outputs improve parallelization, thereby improving GPU efficiency; see Performance and subsections.
- As a rough guideline, batch sizes or neuron counts less than 128 will lead to the layer being limited by memory bandwidth, not computation rate (Tesla V100); see Batch Size.

For convolutional layers:

- Choose the number of input and output channels to be divisible by 8 to enable Tensor Cores. For the first convolutional layer in most CNNs where the input tensor consists of 3-channel images, padding to 4 channels is sufficient if a stride of 2 is used; see Channels In And Out.
- Choose parameters (batch size, number of input and output channels) to be divisible by at least 64 and ideally 256 to enable efficient tiling and reduce overhead; see Quantization Effects.
- Larger values for size-related parameters (batch size, input and output height and width, and the number of input and output channels) can improve parallelization. As with fully-connected layers, this speeds up an operation's efficiency, but does not reduce its absolute duration; see How Convolution Parameters Affect Performance and subsections.
- NVIDIA libraries offer a set of different convolution algorithms with different performance behaviors, dependent on the convolution's parameters. When the size of the input processed by the network is the same in each iteration, autotuning is an efficient method to ensure the selection of the ideal algorithm for each convolution in the network. For TensorFlow, autotuning is enabled by default. For PyTorch, enable autotuning by adding `torch.backends.cudnn.benchmark = True` to your code.
- Choose tensor layouts in memory to avoid transposing input and output data. There are two major conventions, each named for the order of dimensions: NHWC and NCHW. We recommend using the NHWC format where possible. This is supported natively in MXNet and available through XLA in TensorFlow; native support for PyTorch is in development. Additional details can be found in Tensor Layouts In Memory: NCHW vs NHWC.

For activation functions, pooling, and similar operations:

- The speed of these routines is limited by data movement speed (as opposed to calculation speed), and hence Tensor Core acceleration is not used; see Memory-Limited Layers.

Chapter 3.
GPU PERFORMANCE BACKGROUND

It is helpful to understand the basics of GPU execution when reasoning about how efficiently particular layers or neural networks are utilizing a given GPU.

3.1. GPU Architecture Fundamentals

The GPU is a highly parallel processor architecture, composed of processing elements and a memory hierarchy. At a high level, NVIDIA GPUs consist of a number of Streaming Multiprocessors (SMs), on-chip L2 cache, and high-bandwidth DRAM. Arithmetic and other instructions are executed by the SMs, data and code are accessed from DRAM via the L2 cache. As an example, a Volta V100 GPU contains 80 SMs, 6 MB L2 cache, and up to 32 GB of HBM2 memory delivering approximately 900 GB/s bandwidth.

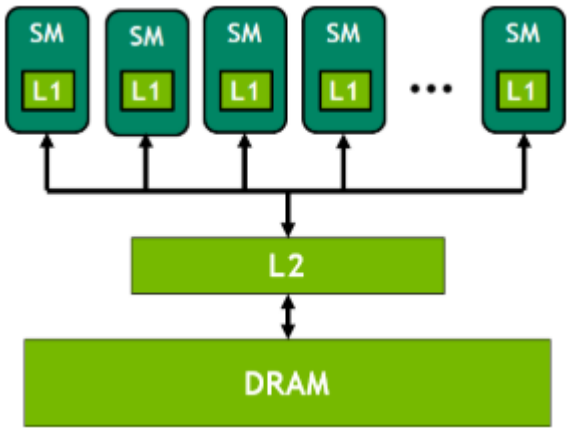

Figure 1 Simplified view of the GPU architecture

Each SM has its own instruction schedulers and various instruction execution pipelines. Multiply-add is the most frequent operation in modern neural networks, acting as a building block for both fully-connected and convolutional layers, both of which can be viewed as a collection of vector dot-products. The following table shows single SM's multiply-add operations per clock for various data types on NVIDIA's two most recent

GPU architectures – Volta and Turing. Each multiply-add comprises two operations, thus one would multiply the throughputs in the table by 2 to get flop counts per clock. To get the flops rate for GPU one would then multiply these by the number of SMs and SM clock rate. For example, Tesla V100 SXM2 GPU with 80 SMs and 1.53 GHz clock rate has peak throughputs of 15.7 FP32 Tflops and 125 FP16 Tflops (throughputs achieved by applications depend on a number of factors discussed throughout this document).

	CUDA Cores				Tensor Cores			
GPU	FP64	FP32	FP16	INT8	FP16	INT8	INT4	INT1
Volta	32	64	128	256	512			
Turing	2	64	128	256	512	1024	2048	8192

Figure 2 Multiply-add operations per clock per SM

As shown in Figure 2, FP16 operations can be executed in either Tensor Cores or CUDA cores. Furthermore, the Turing architecture can execute INT8 operations in either Tensor Cores or CUDA cores. Tensor Cores were introduced in the Volta GPU architecture specifically for accelerating matrix multiply and accumulate operations for machine learning and scientific applications. These instructions operate on small matrix blocks (for example, 4x4 blocks). Note that Tensor Cores compute and accumulate products in higher precision than the inputs. For example, when inputs are FP16 Tensor Cores will compute products without loss of precision and accumulate in FP32. When math operations cannot be formulated in terms of matrix blocks they are executed in other CUDA cores. For example, element-wise addition of two half-precision tensors would be performed by CUDA cores, rather than Tensor Cores.

3.2. GPU Execution Model

To utilize their parallel resources, GPUs execute many threads concurrently.

There are two concepts critical to understanding how thread count relates to GPU performance:

1. GPUs execute functions using a 2-level hierarchy of threads. A given function's threads are grouped into equally-sized *thread blocks*, and a set of thread blocks are launched to execute the function.
2. GPUs hide dependent instruction latency by switching to execution of other threads. Thus, the number of threads needed to effectively utilize a GPU is much higher than the number of cores or instruction pipelines.

The 2-level thread hierarchy is a result of GPUs having many SMs, each of which in turn has pipelines for executing many threads and enables its threads to communicate via shared memory and synchronization. At runtime, a thread block is placed on an SM for execution, enabling all threads in a thread block to communicate and synchronize efficiently. Launching a function with a single thread block would only give work to a single SM, therefore to fully utilize a GPU with multiple SMs one needs to launch many thread blocks. Since an SM can execute multiple thread blocks concurrently, typically

one wants the number of thread blocks to be several times higher than the number of SMs. The reason for this is to minimize the "tail" effect, where at the end of a function execution only a few active thread blocks remain, thus underutilizing the GPU for that period of time as illustrated in Figure 3.

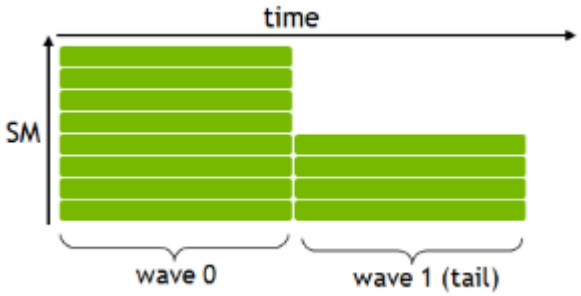

Figure 3 Utilization of an 8-SM GPU when 12 thread blocks with an occupancy of 1 block/SM at a time are launched for execution. Here, the blocks execute in 2 waves, the first wave utilizes 100% of GPU, while the 2nd wave utilizes only 50%.

We use the term *wave* to refer to a set of thread blocks that run concurrently. It is most efficient to launch functions that execute in several waves of thread blocks - a smaller percentage of time is spent in the tail wave, minimizing the tail effect and thus the need to do anything about it. For the higher-end GPUs, typically only launches with fewer than 300 thread blocks should be examined for tail effects.

3.3. Understanding Performance

Performance of a function on a given processor is limited by one of the following three factors; memory bandwidth, math bandwidth and latency.

Consider a simplified model where a function reads its input from memory, performs math operations, then writes its output to memory. Say T_{mem} time is spent in accessing memory, T_{math} time is spent performing math operations. If we further assume that memory and math portions of different threads can be overlapped, the total time for the function is max(T_{mem}, T_{math}). The longer of the two times demonstrates what limits performance: If math time is longer we say that a function is *math limited*, if memory time is longer then it is *memory limited*.

How much time is spent in memory or math operations depends on both the algorithm and its implementation, as well as the processor's bandwidths. Memory time is equal to the number of bytes accessed in memory divided by processor's memory bandwidth. Math time is equal to the number of operations divided by the processor's math bandwidth. Thus, on a given processor a given algorithm is math limited if $T_{math} > T_{mem}$ which can be expressed as:

$$\#\ ops/BW_{math} > \#\ bytes/BW_{mem}$$

By simple algebra, the inequality can be rearranged to:

$$\#\ ops/\#\ bytes > BW_{math}/BW_{mem}$$

The left-hand side, the ratio of algorithm implementation operations and the number of bytes accessed, is known as algorithm's *arithmetic intensity*. The right-hand side, the ratio of a processor's math and memory bandwidths, is sometimes called *ops:byte* ratio. Thus, an algorithm is math limited on a given processor if the algorithm's arithmetic intensity is higher than the processor's ops:byte ratio. Conversely, an algorithm is memory limited if its arithmetic intensity is lower than the processor's ops:byte ratio.

Let's consider some concrete examples from deep neural networks, listed in Table 1 below. For these examples, we will compare the algorithm's arithmetic intensity to the ops:byte ratio on an NVIDIA Volta V100 GPU. V100 has a peak math rate of 125 FP16 Tensor TFLOPS, an off-chip memory bandwidth of approx. 900 GB/s, and an on-chip L2 bandwidth of 3.1 TB/s, giving it a ops:byte ratio between 40 and 139, depending on the source of an operation's data (on-chip or off-chip memory).

Table 1 Examples of neural network operations with their arithmetic intensities. Limiters assume FP16 data and a NVIDIA Tesla V100 GPU.

Operation	Arithmetic Intensity	Usually limited by...
Linear layer (4096 outputs, 1024 inputs, batch size 512)	315 FLOPS/B	arithmetic
Linear layer (4096 outputs, 1024 inputs, batch size 1)	1 FLOPS/B	memory
Max pooling with 3x3 window and unit stride	2.25 FLOPS/B	memory
ReLU activation	0.25 FLOPS/B	memory
Layer normalization	< 10 FLOPS/B	memory

As the table shows, many common operations have low arithmetic intensities - sometimes only performing a single operations per two-byte element read from and written to memory. Note that this type of analysis is a simplification, as we're counting only the *algorithmic* operations used. In practice, functions also contain instructions for operations not explicitly expressed in the algorithm, such as memory access instructions, address calculation instructions, control flow instructions, and so on.

The arithmetic intensity and ops:byte ratio analysis assumes that a workload is sufficiently large to saturate a given processor's math and memory pipelines. However, if the workload is not large enough, or does not have sufficient parallelism, the processor will be underutilized and performance will be limited by latency. For example, consider the launch of a single thread that will access 16 bytes and perform 16000 math operations. While the arithmetic intensity is 1000 FLOPS/B and the execution should be math-limited on a V100 GPU, a single thread grossly under-utilizes the GPU, leaving nearly all of its math pipelines and execution resources idle. Furthermore, the arithmetic intensity calculation assumes that inputs and outputs are accessed from memory exactly once. It is not unusual for algorithm implementations to read input elements multiple times, which would effectively reduce arithmetic intensity. Thus, arithmetic intensity is a first-order approximation; profiler information should be used if more accurate analysis is needed.

3.4. Summary

To roughly approximate what limits performance of a particular function on a given GPU, one can take the following steps:

- Look up the number of SMs on the GPU, and determine the ops:bytes ratio for the GPU.
- Compute the arithmetic intensity for the algorithm.
- Determine if there is sufficient parallelism to saturate the GPU by estimating the number and size of thread blocks. If the number of thread blocks is at least roughly 4x higher than the number of SMs, and thread blocks consist of a few hundred threads each, then there is likely sufficient parallelism.
  - For fully-connected layers, section Fully-Connected Layer provides intuition on parallelization.
  - For convolutional layers, section Convolutions provides intuition on parallelization.
- The most likely performance limiter is:
  - Latency if there is not sufficient parallelism
  - Math if there is sufficient parallelism and algorithm arithmetic intensity is higher than the GPU ops:byte ratio.
  - Memory if there is sufficient parallelism and algorithm arithmetic intensity is lower than the GPU ops:byte ratio.

Chapter 4. DNN OPERATION CATEGORIES

While modern neural networks are built from a variety of layers, their operations fall into three main categories according to the nature of computation.

4.1. Elementwise Operations

Elementwise operations may be unary or binary operations; the key is that layers in this category perform mathematical operations on each element independently of all other elements in the tensor.

For example, a ReLU layer returns **`max(0, x)`** for each ***x*** in the input tensor. Similarly, element-wise addition of two tensors computes each output sum value independently of other sums. Layers in this category include most non-linearities (sigmoid, Tanh, etc.), scale, bias, add, and others. These layers tend to be memory-limited, as they perform few operations per bytes accessed.

4.2. Reduction Operations

Reduction operations produce values computed over a range of input tensor values.

For example, pooling layers compute values over some neighborhood in the input tensor. Batch normalization computes the mean and standard deviation over a tensor before using them in operations for each output element. In addition to pooling and normalization layers, SoftMax also falls into the reduction category. Typical reduction operations have low arithmetic intensity and thus are memory limited.

4.3. Dot-Product Operations

Operations in this category can be expressed as dot-products of elements from two tensors, usually a weight (learned parameter) tensor and an activation tensor.

These include fully-connected layers, occurring on their own and as building blocks of recurrent and attention cells. Fully-connected layers are naturally expressed as matrix-vector and matrix-matrix multiplies. Convolutions can also be expressed as collections

of dot-products - one vector is the set of parameters for a given filter, the other is an "unrolled" activation region to which that filter is being applied. Since filters are applied in multiple locations, convolutions too can be viewed as matrix-vector or matrix-matrix multiply operations (see section on Convolution Algorithms for more details).

Operations in the dot-product category can be math-limited if the corresponding matrices are large enough. However, for the smaller sizes these operations too end up being memory-limited. For example, a fully-connected layer applied to a single vector (a tensor for minibatch of size 1) is memory limited.

In the following section, we explain the fundamentals of matrix-matrix multiplication performance, and provide more details how various parameters, including minibatch size, filter size, and others affect the size of the corresponding matrix operations. Afterwards, we will come back to element-wise and reduction operations, briefly describing their performance as well.

Chapter 5. 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 section we describe GEMM performance fundamentals common to understanding the performance of layers described in subsequent sections.

GEMM is defined as the operation $C=aAB+\beta C$, with A and B as matrix inputs, a 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 a 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 a 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.

5.1. 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, in other words 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 a 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 a Tesla 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.

$$Arithmetic\ Intensity = \frac{\#\ ops}{\#\ bytes} = \frac{2 \cdot (M \cdot N \cdot K)}{2 \cdot (M \cdot K + N \cdot K + M \cdot N)} = \frac{M \cdot N \cdot K}{M \cdot K + N \cdot K + M \cdot N}$$

As an example, let's consider a M x N x K = 8192 x 128 x 8192 GEMM. For this specific case, 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).

5.1.1. 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 4). 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.

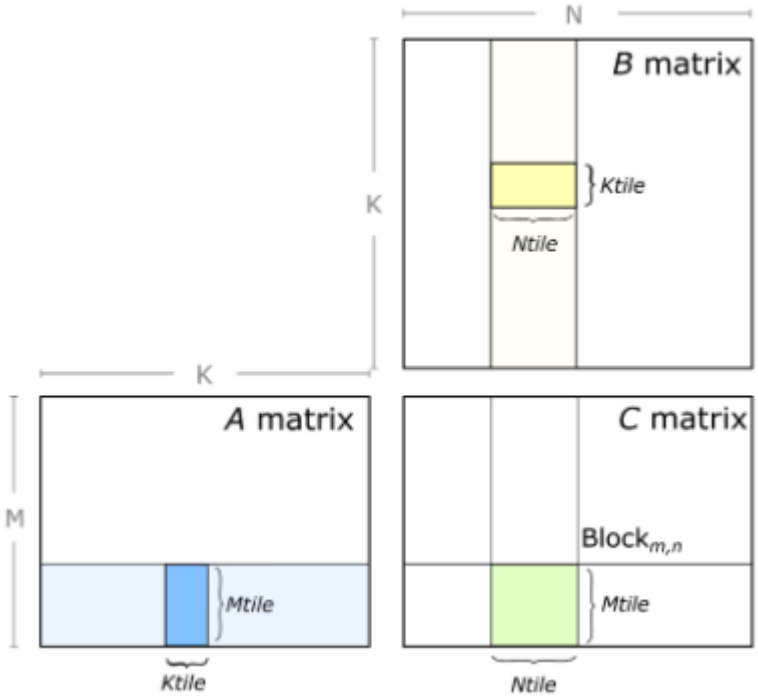

Figure 4 Tiled outer product approach to GEMMs

5.1.2. Tensor Core Requirements

As was shown in the GPU Architecture Fundamentals section, the latest NVIDIA GPUs have introduced Tensor Cores to maximize the speed of tensor multiplies. In order to use Tensor Cores, NVIDIA libraries require that matrix dimensions M, N, and K are multiples of 8 (with FP16 data) or 16 (with INT8 data).

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. When the dimensions are not multiples of 8 (or 16), libraries will revert to a slower implementation without Tensor Cores. This effect can be seen in Figure 5 - as we change the M dimension, cases that are multiples of 8 are executed on Tensor Cores, resulting in a speedup of about 6x. For this reason, we recommend padding dimensions where necessary to enable Tensor Cores.

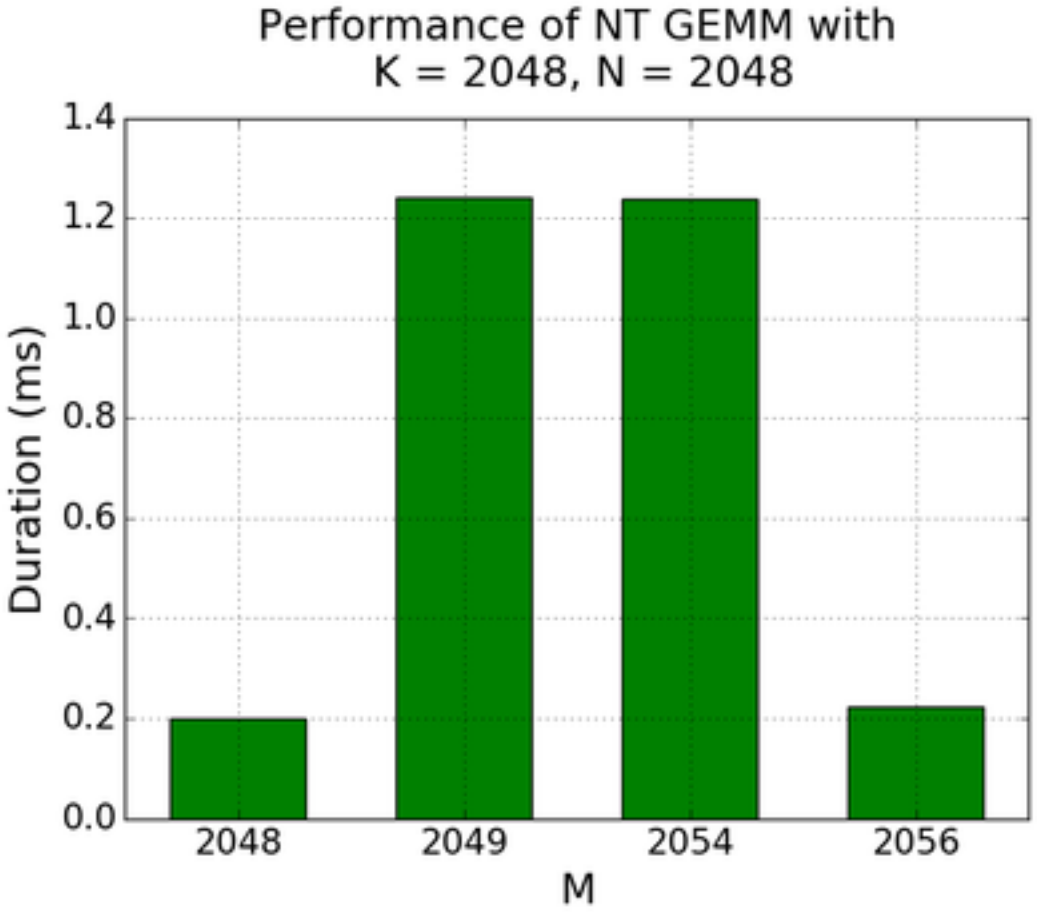

Figure 5 GEMM speedup when Tensor Core use is triggered by using dimensions that are multiples of 8. “NT” means A is accessed non-transposed, B is accessed transposed. Tesla V100-SXM3-32GB GPU, cuBLAS v10.1.

5.1.3. 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 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 GEMMs are too small, the reduction in either tile efficiency or tile parallelism will likely prevent the GPU from running at peak math utilization. Figure 6 and Figure 7 illustrate this general trend; larger GEMMs achieve higher throughput.

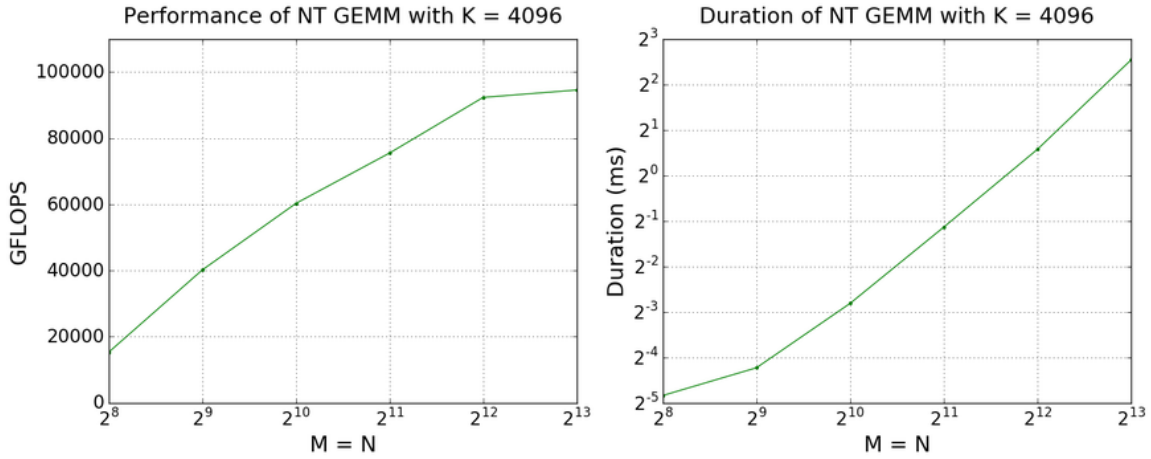

Figure 6 Performance improves as the M-N footprint of the GEMM increases. Duration also increases, but not as quickly as the M-N dimensions themselves; it is sometimes possible to increase the GEMM size (e.g. use more weights) for only a small increase in duration. Tesla V100-SXM3-32GB GPU, cuBLAS v10.1.

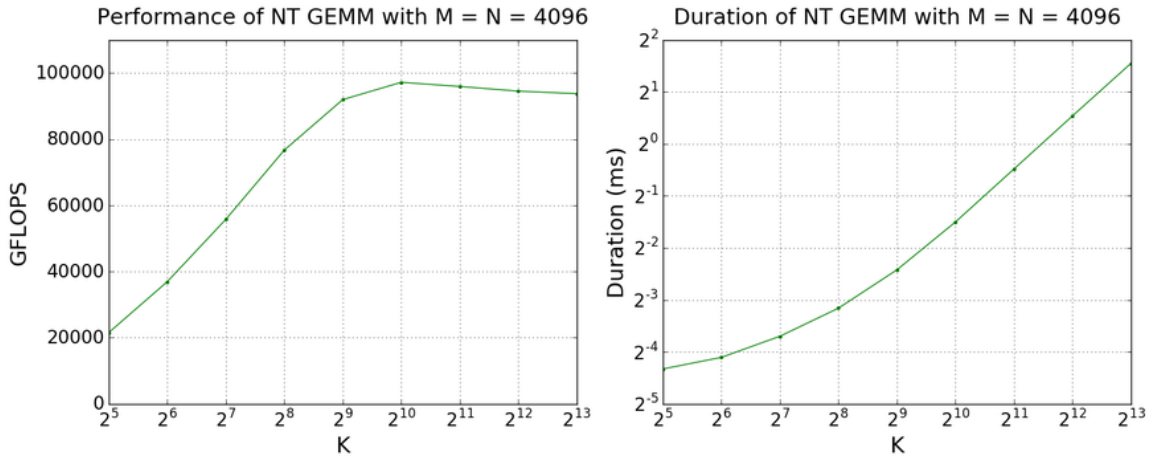

Figure 7 Performance improves as the K dimension increases, even when M=N is relatively large, as setup and tear-down overheads for the computation are amortized better when the dot product is longer. Tesla V100-SXM3-32GB GPU, cuBLAS v10.1.

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

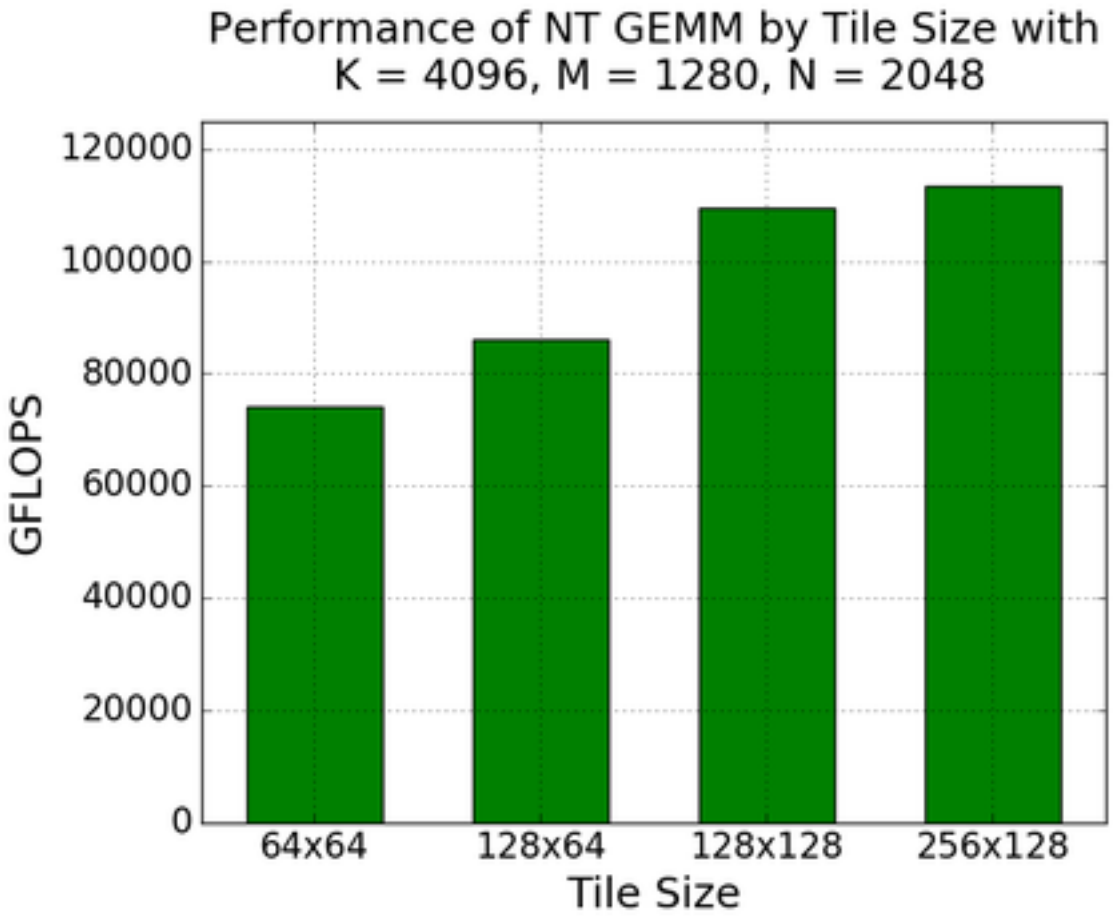

Figure 8 Larger tiles run more efficiently. The 256x128-based GEMM runs exactly one tile per SM, the other GEMMs generate more tiles based on their respective tile sizes. Tesla V100-SXM3-32GB GPU, cuBLAS v10.1.

The chart shows the performance of a MxNxK = 1280x2040x4096 GEMM with different tile sizes. It demonstrates that the increased tile parallelism with smaller tiles (64x64 enables 8x more parallelism than 256x128) comes at 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.

5.2. Dimension Quantization Effects

As described in the GPU Execution Model section above, 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.

5.2.1. 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 9, which assumes 128x128 tiles and two matrix dimensions.

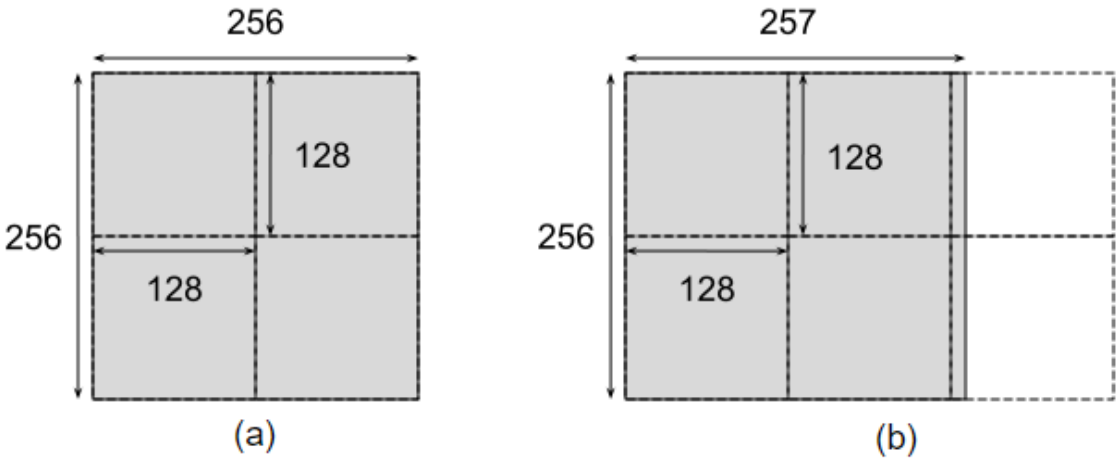

Figure 9 Example of tiling with 128x128 thread block tiles. (a) Best case - matrix dimensions are divisible by tile dimensions (b) Worse case - tile quantization results in six thread blocks being launched, two of which waste most of their work.

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 9 (b) executes 1.5x as many arithmetic operations as Figure 9 (a) despite needing only 0.39% more operations algorithmically. As this shows, 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 = 20480, 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 increase with N, as reflected by the GFLOPS in Figure 10 below. Notice that throughput reduces by nearly half 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.

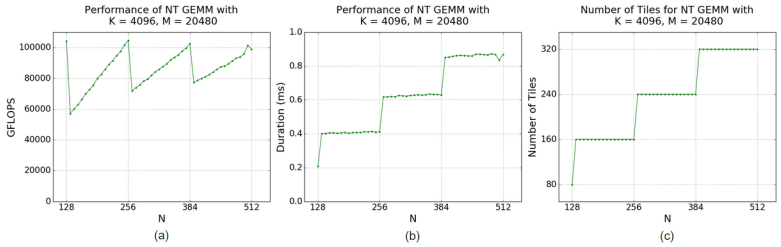

Figure 10 Tile quantization effect on (a) achieved flops throughput and (b) elapsed time, alongside (c) the number of tiles created. Measured with a function that forces the use of 256x128 tiles over the MxN output matrix. In practice, cuBLAS would select narrower tiles (for example, 64-wide) to reduce the quantization effect. Tesla V100-SXM3-32GB GPU, cuBLAS v10.1.

5.2.2. 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 = 1280. A Volta V100 GPU has 80 SMs; in the particular case of 256x128 thread block tiles, it can execute one thread block per SM, leading to a wave size of 80 tiles that can execute simultaneously. Thus, GPU utilization will be highest when the number of tiles is an integer multiple of 80 or just below.

The M dimension will always be divided into 1280/256 = 5 tiles per column. When N = 2048, the N dimension is divided into 2048/128 = 16 tiles per row, and a total of 5*16 = 80 tiles are created, comprising one full wave. When 2048 < N <= 2176, an additional tile per row is created for a total of 5*17 = 85 tiles, leading to one full wave and a 'tail' wave of only 5 tiles. The tail wave takes nearly the same time to execute as the full 80-tile wave in this example, but uses only 5/80 = 6.25% of V100's SMs during that time. Consequently, GFLOPS roughly halve and duration roughly doubles from N = 2048 to N = 2056 (Figure 11). Similar jumps can be seen after N = 4096, N = 6144, and N = 8192, which also each map to an integer number of full waves.

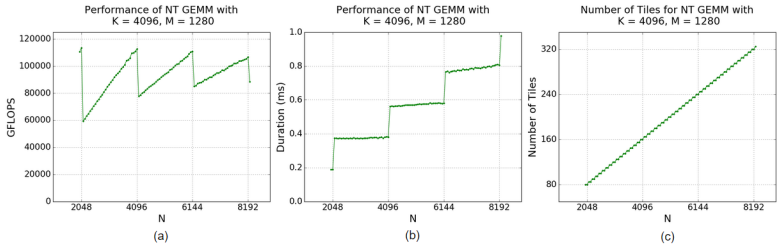

Figure 11 The effects of wave quantization in terms of (a) achieved flops throughput and (b) elapsed time, as well as (c) the number of tiles created. Measured with a function that uses 256x128 tiles over the MxN output matrix. Note that the quantization effect occurs when the number of tiles passes a multiple of 80. Tesla V100-SXM3-32GB GPU, cuBLAS v10.1.

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 10 (c) and Figure 11 (c) in both the tile and wave quantization illustrations show this difference.

Chapter 6. FULLY-CONNECTED LAYER

Fully-connected layers are commonly used in neural networks, and connect every input neuron to every output neuron.

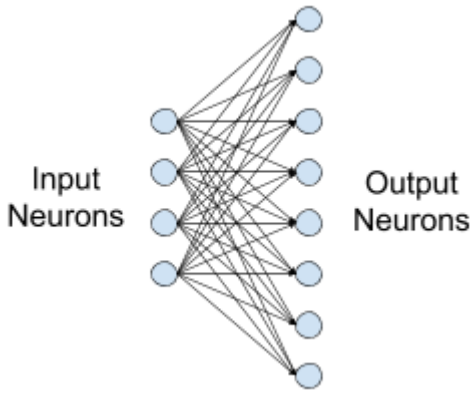

Figure 12 Example of a small fully-connected layer with four input and eight output neurons.

Three parameters define a fully-connected layer: batch size, number of inputs, and number of outputs. Forward propagation, activation gradient computation, and weight gradient computation are directly expressed as matrix-matrix multiplications. How the three parameters map to GEMM dimensions varies among frameworks, but the underlying principles are the same. For the purposes of the discussion, we adopt the convention used by PyTorch and Caffe where A contains the weights and B the activations. In TensorFlow, matrices take the opposite roles, but the performance principles are the same.

Table 2 Mapping of inputs, outputs, and batch size to GEMM parameters M, N, K.

Computation Phase	M	N	K
Forward Propagation	Number of outputs	Batch size	Number of inputs
Activation Gradient	Number of inputs	Batch size	Number of outputs

Computation Phase	M	N	K
Weight Gradient	Number of inputs	Number of outputs	Batch size

The compositions of the matrices in the GEMM are shown in Figure 13.

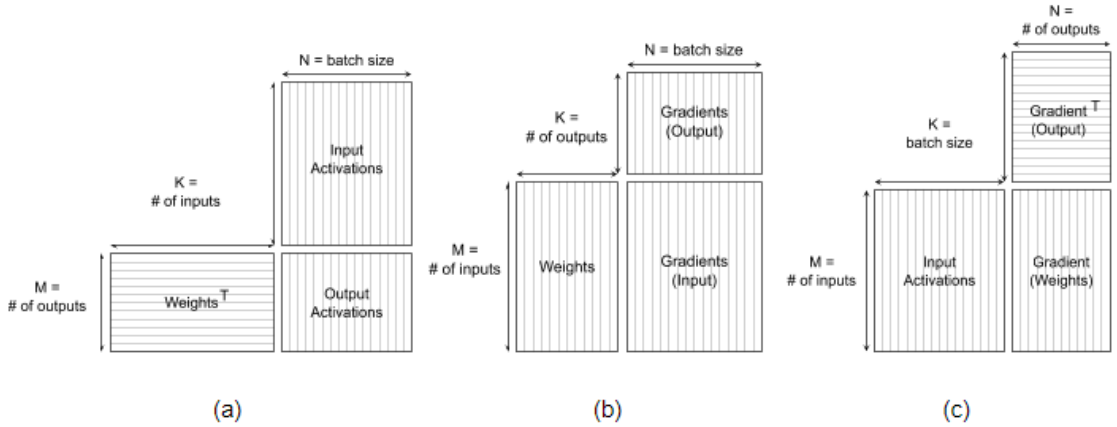

Figure 13 Dimensions of equivalent GEMMs for (a) forward propagation, (b) activation gradient, and (c) weight gradient computations of a fully-connected layer.

6.1. Performance

6.1.1. Input Features And Output Neuron Counts

As fully-connected layers directly correspond to GEMMs, their performance trends are identical to the descriptions in the previous section. Larger parameters tend to allow better parallelization and thus better performance.

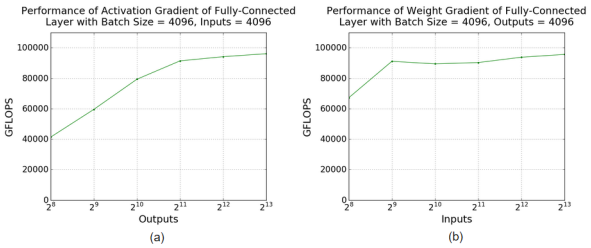

Figure 14 Larger fully-connected layers are equivalent to larger GEMMs, which perform better. Tesla V100-SXM3-32GB GPU, cuBLAS v10.1.

6.1.2. Batch Size

Batch size directly contributes to the tiling strategy for two out of three training phases - forward pass and activation gradient computation.

For these phases, the output matrix dimension includes batch size, so larger batch sizes result in more tiles. Thus, when the model size is too small to fully utilize a GPU, one option is to train with larger batch sizes which will help extract more performance. For weight gradient computation, the output matrix has the same dimensions as the weights, thus batch size does not affect the tile count directly. Instead, batch size here maps to the K dimension of the GEMM, so for weight gradients larger batch size enables more efficient computation per tile. Figure 15 shows the performance impact of varying batch size on forward, activation gradient, and weight gradient computations for a fully-connected layer with 4096 inputs and 1024 outputs. The larger batch sizes exceed 90 TFLOPS delivered performance.

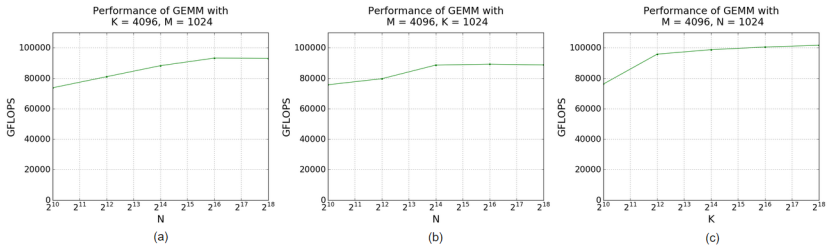

Figure 15 Performance data for (a) forward propagation, (b) activation gradient computation, and (c) weight gradient computation for a fully-connected layer with 4096 inputs, 1024 outputs, and varying batch size. Tesla V100-SXM3-32GB GPU, cuBLAS v10.1.

Of particular interest are GEMMs where one dimension is very small. For example, on Tesla V100 and for a fully-connected layer with 4096 inputs and 4096 outputs, forward propagation, activation gradient computation, and weight gradient computation are estimated to be memory-bound for batch sizes below 128 (see Figure 16).

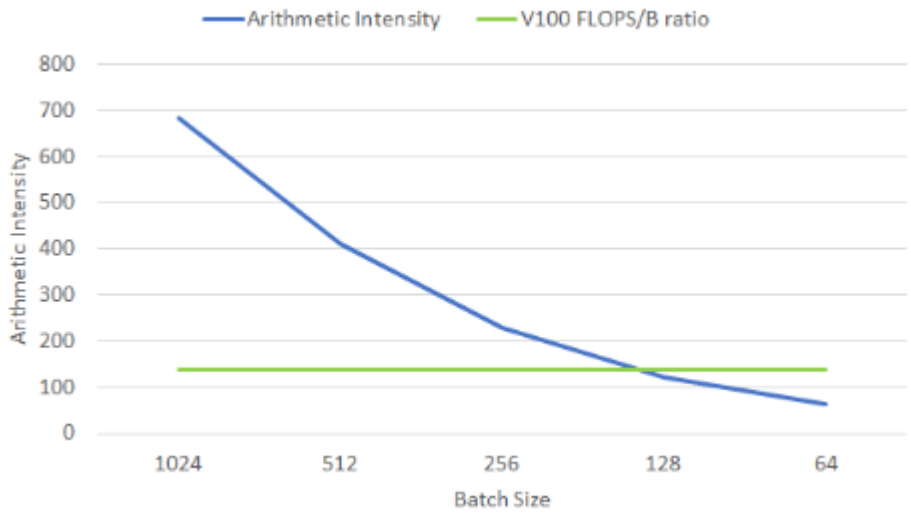

Figure 16 Arithmetic intensity for a fully-connected layer with 4096 inputs and 4096 outputs. Batch sizes below 128 become bandwidth limited on Volta V100 accelerators.

Larger numbers of inputs and outputs improve performance somewhat, but the computation will always be bandwidth-limited for very small batch sizes, for example 8 and below. For a discussion of math- and bandwidth-limited computations, see Math And Memory Bounds.

Chapter 7. CONVOLUTIONS

7.1. Introduction

A convolution is defined by several parameters describing the sizes of the input and filter tensors as well as the behavior of the convolution, such as the padding type being used.

Figure 17 illustrates the minimum parameter set required to define a convolution:

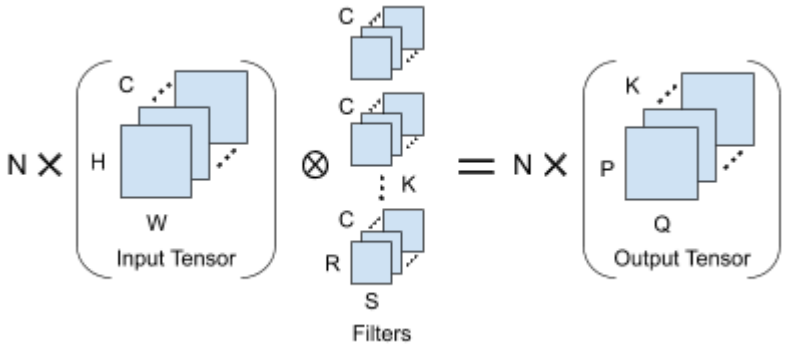

Figure 17 Convolution of an NCHW input tensor with a KCRS weight tensor, producing a NKPQ output.

In the remainder of the text, we'll use the single-letter abbreviations for each parameter.

N and K here are not related to the N and K parameters in a GEMM; GEMM dimensions will be represented here as "M" x "N" x "K" to keep them distinct from the convolution's parameters.

Table 3 Parameters defining a convolution

Parameter	Tensor	Meaning
`N`	N/A	Batch size

Parameter	Tensor	Meaning
`C`	Input	Number of channels
`H`		Height
`W`		Width
`K`	Output	Number of channels
`P`		Height (often derived from other parameters)
`Q`		Width (often derived from other parameters)
`R`	Filter	Height
`S`		Width
`U`		Vertical stride
`V`		Horizontal stride
`PadH`		Input padding in the vertical dimension
`PadW`		Input padding in the horizontal dimension
`DilH`		Dilation in the vertical dimension
`DilW`		Dilation in the horizontal dimension

7.2. Convolution Algorithms

NVIDIA cuDNN library implements convolutions using two primary methods: implicit-GEMM-based and transform-based.

The implicit GEMM approach is a variant of direct convolution, and operates directly on the input weight and activation tensors. Alternatively, convolutions can be computed by transforming data and weights into another space, performing simpler operations (for example, pointwise multiplies), and then transforming back. The cuDNN library provides some convolution implementations using FFT and Winograd transforms.

7.2.1. Choosing A Convolution Algorithm With cuDNN

When running a convolution with cuDNN, for example with **`cudnnConvolutionForward`**, you may specify which general algorithm is used.

The cuDNN API provides functions for estimating the relative performance of different algorithms. One set of functions, prefixed with **`cudnnGet`**, uses a set of heuristics to predict the relative performance of available algorithms. These functions evaluate quickly; however, although we're constantly improving our heuristics, the predictions may not always be accurate. Suboptimal algorithm choice may occasionally occur, and is more common for unusual types of convolutions and corner cases.

An alternative set of functions, prefixed with **`cudnnFind`**, tests and reports the performance of all available algorithms to determine the most efficient option for the given convolution operation. The benefit to using these functions is that the algorithm selected is the best choice. However, since actual performance tests are being run, these functions can be time and resource-intensive.

After an algorithm is chosen, our heuristics specify additional low-level details; for example, tile size, discussed at length in the following section. These parameters are not exposed in the API.

7.3. Tensor Core Usage And Performance Recommendations

The primary method to execute convolutions (without transforms) used by NVIDIA Tensor Core GPUs is called implicit GEMM. It performs exactly the same number of math operations as a direct convolution, and hence is computationally equivalent.

Implicit GEMM operates natively on the convolution input tensors, converting the computation into a matrix multiply on the fly. It is important to note that corresponding matrices are never created in memory. Thus, to calculate arithmetic intensity, one can use the original tensor sizes.

To illustrate the concept of convolution as a matrix multiply let's first consider a single application of a convolution filter to input data. Say we are applying a 3x3 convolution to a 128-channel input tensor. To compute a single output value, we effectively compute a dot-product of two 1,152-element vectors. One is the weights for a filter (3x3x128 = 1,152). The other is composed of the data (activation) values that are multiplied with the weights to produce the output. Since not all 1,152 data values are contiguous in memory, the original tensor layout is read and on the fly is converted to the appropriate vector form. To compute all the outputs we perform multiple dot-products, which can be seen as a matrix multiply, but since the matrices are implicitly formed, rather than created in memory, this method is called *implicit* GEMM. To understand the performance of convolutions, however, it can be useful to understand the shapes and sizes of these "virtual" matrices.

7.3.1. Tensor Layouts In Memory: NCHW vs NHWC

Convolutions typically operate on four-dimensional tensors: a batch composed of N "images" of C channels of H x W feature maps.

Deep learning frameworks commonly use NCHW and NHWC layouts in memory (the acronym lists the dimensions from the slowest to the fastest varying in memory). Layout choice has an effect on performance, as convolutions implemented for Tensor Cores require NHWC layout and are fastest when input tensors are laid out in NHWC.

NCHW layouts can still be operated on by Tensor Cores, but include some overhead due to automatic transpose operations, as shown in Figure 18. To maximize performance, we recommend using NHWC tensor layout.

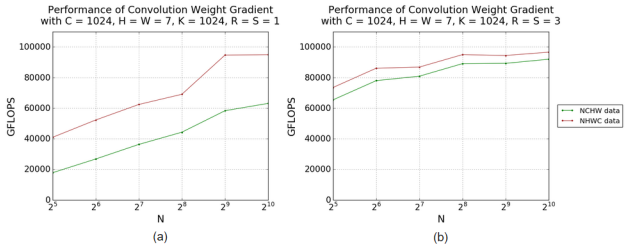

Figure 18 Kernels that do not require a transpose (NHWC) tend to perform somewhat better than kernels that require one or more (NCHW). Tesla V100-SXM3-32GB GPU, cuDNN v7.5.

In practice, NHWC layouts are natively available and well supported in MxNet, and can be used via XLA in TensorFlow. Native PyTorch implementations are in development. Performance examples in this section can be assumed to use input and output data in the NHWC layout unless otherwise stated.

7.3.2. Implicit GEMM Dimensions

Let's now consider the dimensions of the matrices we encounter when performing forward convolution, calculating activation gradients, and calculating weight gradients.

Table 4 Translation of convolution parameters to corresponding GEMM parameters.

Computation Phase	"M"	"N"	"K"
Forward Propagation	N*P*Q	K	C*R*S
Activation Gradient	N*H*W	C	K*R*S
Weight Gradient	C*R*S	K	N*P*Q

The composition of the "virtual" matrices is shown in Figure 19. For each pass, there is one virtual matrix that, if explicitly constructed, would contain more values than its corresponding tensor: during forward convolution, the A matrix (N*P*Q x C*R*S) is composed of input activations (a tensor with dimensions N x H x W x C). Each individual input activation appears in R*S places in the matrix, repeated with necessary offsets to cause multiplication of that input value with the overlaid values of the matching R x S filter channel. Similar conceptual expansions occur for output activation gradients when computing input activation gradients, and for input activations during weight gradient calculation.

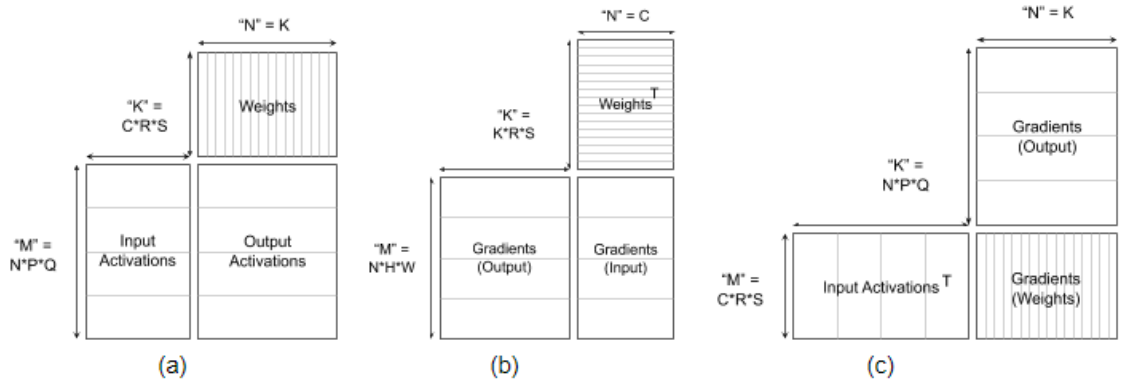

Figure 19 Dimensions of equivalent GEMMs for (a) forward convolution, (b) activation gradient calculation, and (c) weight gradient calculation.

It is important to reiterate that matrices of these sizes are not stored in memory; they are an abstraction to help explain the computation. The 'repeated' values are not literally copied, and wasteful reads from memory are avoided. This is directly visible in the calculation of arithmetic intensity: A (forward) implicit GEMM performs CRS multiplies and adds for each element of the NKPQ output tensor, so NKPQCRS multiply-accumulates in total, reads two input tensors of size NCHW and KCRS, and produces one output tensor of size NKPQ. Therefore, arithmetic intensity in FP16 with 2 bytes/element is:

$$Arithmetic\ Intensity = \frac{\#\ ops}{\#\ bytes} = \frac{2 \cdot (N \cdot K \cdot P \cdot Q) \cdot (C \cdot R \cdot S)}{2 \cdot (N \cdot C \cdot H \cdot W + K \cdot C \cdot R \cdot S + N \cdot K \cdot P \cdot Q)} = \frac{N \cdot K \cdot P \cdot Q \cdot C \cdot R \cdot S}{N \cdot C \cdot H \cdot W + K \cdot C \cdot R \cdot S + N \cdot K \cdot P \cdot Q}$$

For example, computing a 3x3 convolution on a 256x56x56x64 input tensor, producing a 256x56x56x128 output, all in half-precision, has an arithmetic intensity of 383.8 FLOPS/byte.

7.3.3. Quantization Effects

As discussed previously in Dimension Quantization Effects, tile and wave quantization effects can be significant, especially for small problem sizes. Just like for GEMMs, in implicit GEMM, the representation of the output matrix is divided into tiles of a chosen size, and that set of tiles is distributed across available multiprocessors.

Our testing GPU has 80 SMs. Each SM can handle a number of thread blocks in parallel that is dependent on the kernel being used; for best parallelization, an implicit GEMM should contain an integer multiple of 80 tiles.

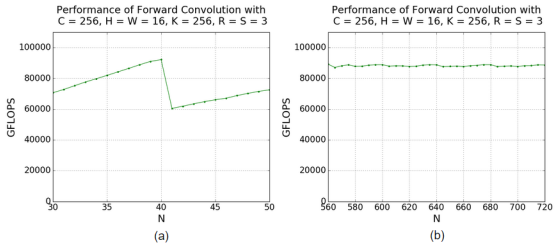

Figure 20 Graphs showing performance of convolution with filter size 3x3, input size 16x16, 256 channels of input, and 256 channels of output. Best performance is observed when N is divisible by 40 (when a multiple of 160 tiles, two in parallel on each SM, are created). Tesla V100-SXM3-32GB GPU, cuDNN v7.5.

As Figure 20 shows, convolutions that result in small equivalent GEMMs can exhibit significant quantization effects. When N = 40, 160 tiles are created; when N = 41, 164 tiles are created. The former results in high Tensor Core utilization, while the latter will require an additional wave to process the remainder of 4 tiles, severely impacting performance (Figure 20 (a)). Once the convolutions are reasonably large, the effect is less pronounced (Figure 20 (b)).

It is worth noting that weight gradient quantization does not behave as implied by the GEMM dimensions in Figure 19; for quantization purposes, the height of the matrix to be tiled should be seen as C (rather than C*R*S). This is discussed in more detail at the end of Filter Size.

7.3.4. How Convolution Parameters Affect Performance

In this section, we discuss the trends affecting performance. To keep things simple, padding is set such that H = P and W = Q, and both the stride and dilation are equal to one unless indicated otherwise.

7.3.4.1. Batch Size, Height And Width

When representing a forward convolution as a GEMM, the product of batch size, output height, and output width (N*P*Q) is the “M” dimension of the unrolled input tensor (A matrix) as well as the output (C matrix).

The individual values of these parameters are not especially important to GEMM performance; only the final dimension, the product N*P*Q, is significant. Conveniently, in most applications, batch size may be changed more easily than the parameters contributing to the height and width of the output tensor.

Generally, efficiency improves as N*P*Q increases, with diminishing returns. From Figure 21, we can see that points with equivalent N*P*Q have roughly equivalent performance, as the corresponding GEMMs have the same dimensions.

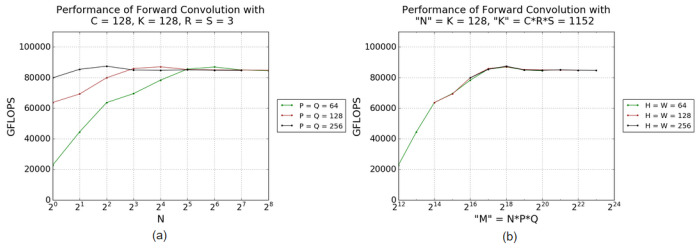

Figure 21 Different perspectives on the same data. Performance improves with increased N*P*Q. Tesla V100-SXM3-32GB GPU, cuDNN v7.5.

When calculating the activation gradient, N*H*W is the "M" dimension of the equivalent GEMM (compared to N*P*Q in forward convolution). With a filter stride of 1, the performance of forward convolution and activation gradient calculation will be roughly the same.

In contrast, for weight gradient calculation, N*P*Q becomes the accumulation ("K" in a GEMM) dimension. The performance impact of this dimension is not as straightforward, as it does not affect the tiling of the output matrix in any way. However, larger values of N*P*Q generally lead to more time spent multiplying and accumulating elements, rather than in setup and teardown overhead for the GEMM computation, improving the fraction of peak performance achieved for the whole operation. It is worth noting that cuDNN generally supports tiling in the N*P*Q dimension as well for weight gradients, as many common layers result in tiny (for example, 64x64 in the first block of a standard ResNet) output matrices that, by themselves, don't offer enough tile parallelism to keep the GPU occupied.

7.3.4.2. Filter Size

The equivalent GEMM for forward convolution has a "K" dimension of C*R*S. As mentioned previously, the "K" dimension does have an impact on performance, and this effect is most pronounced for small GEMMs.

When using a 1x1 filter, layers with more input channels tend to perform better in forward convolution (Figure 22), as it is ultimately the C*R*S product that matters.

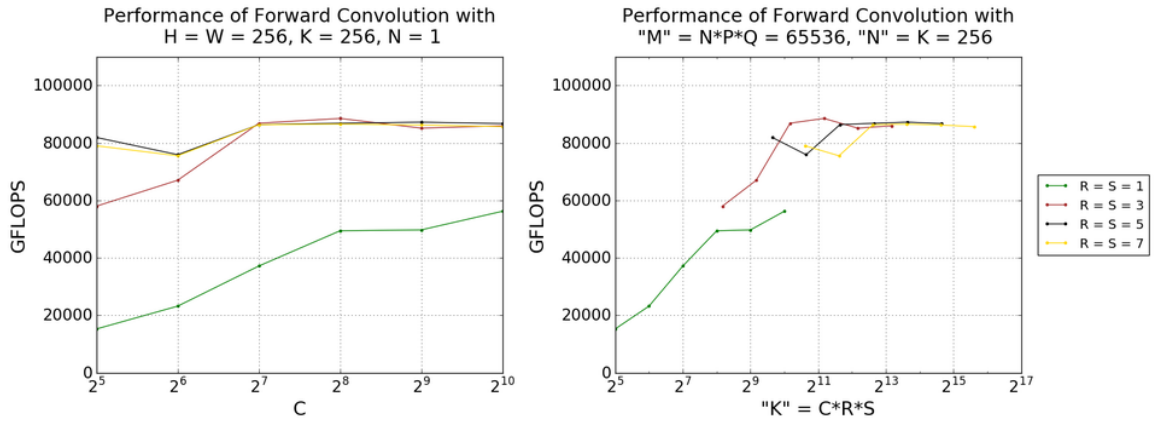

Figure 22 Convolutions with larger filters tend to perform better (C*R*S matters). Tesla V100-SXM3-32GB GPU, cuDNN v7.5.

When calculating activation gradients of the convolution, K affects this dimension instead: “K” = K*R*S. There is a clear similarity between Figure 22 and Figure 23; in general, trends related to C for forward convolution are related to K for activation gradient calculation and vice versa.

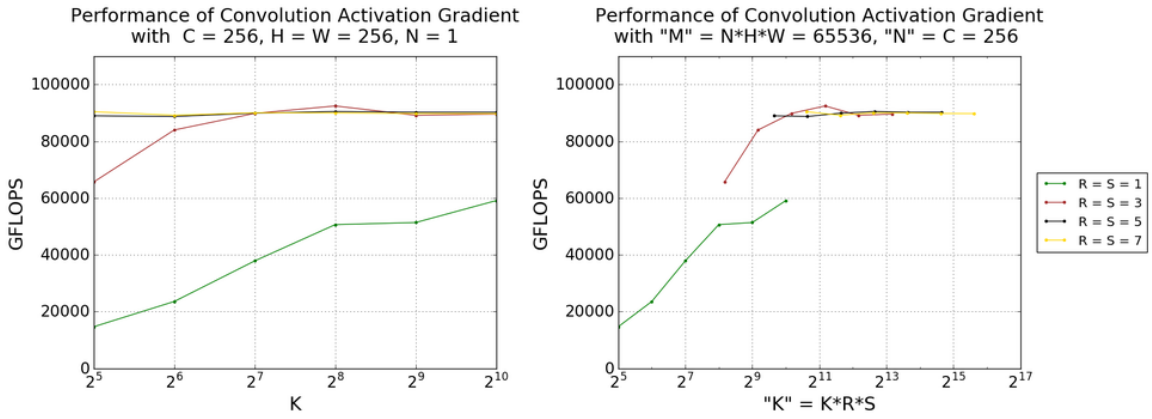

Figure 23 Near-identical relations as in Figure 22, but involving K. For activation gradients, K*R*S matters. Tesla V100-SXM3-32GB GPU, cuDNN v7.5.

The weight gradient calculation has “M” = C*R*S, therefore the impact of filter size on performance is similar to that discussed for batch size, height, and width previously; larger values tend to perform better. When considering tile quantization, however, the weight gradient algorithm differs from the forward and data gradient ones; only the C dimension, not the full C*R*S dimension, quantizes against the tile size (meaning, filter size parameters can be ignored). For example, when using 64x64 tiles, if C = 32, half of each tile (vertically) is wasted regardless of R and S; only the value of C matters.

7.3.4.3. Channels In And Out

Using Tensor Cores on Volta requires that C and K be multiples of 8 in FP16 or 16 in INT8. A caveat applies here: currently, for NCHW-packed FP16 data, channels will be automatically padded to multiples of 8 such that tensor cores will be enabled.

However, using NCHW data with tensor core enabled kernels involves some additional transpose costs, which are discussed in Tensor Layouts In Memory: NCHW vs NHWC. On the other hand, automatic padding doesn’t kick in for NHWC-packed data, so a less-efficient fallback kernel, which does not make use of tensor cores, is chosen. Convolutions with NHWC data do perform better than those with NCHW data given that C and K are divisible by 8. In other words, if a layer is already being used with NCHW data, automatic padding will occur; however, if NHWC data is being used, choosing or padding C and K to be a multiple of 8 improves performance.

Unfortunately, especially for layers near the beginning and end of a network, C and K can be small and non-negotiable. For the first layer in a network, it is common to have a very small value of C (1 or 3 for grayscale and RGB or YCrCb images, respectively). Special-case convolution implementations are available to meet this need, specifically for C = 4 and a stride of 2 (Figure 24), a common case in many CNNs.

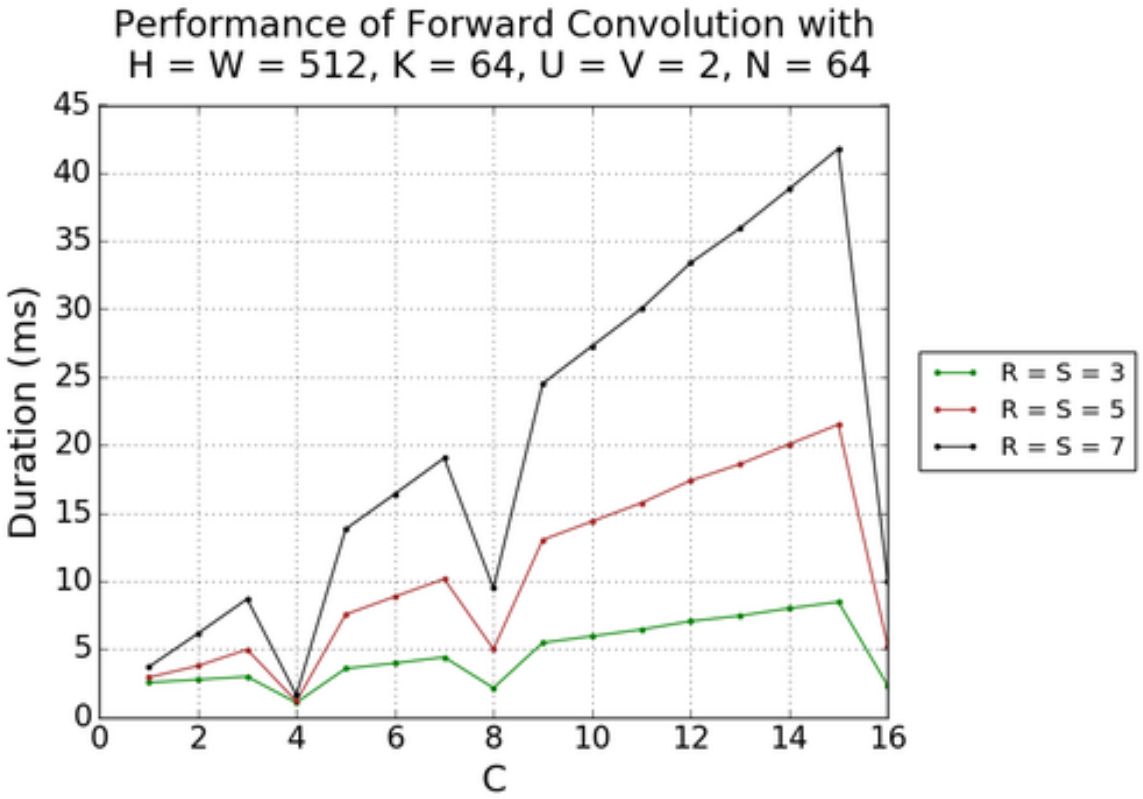

Figure 24 Specialized kernels for C = 4 speed up common first layers in convolutional neural nets (NHWC data used). Padding to C = 4 or the next multiple of 8 results in a performance improvement. Tesla V100-SXM3-32GB GPU, cuDNN v7.5.

Forward convolution performance relative to C was previously discussed in Filter Size ("K" = C*R*S), as was activation gradient calculation relative to K ("K" = K*R*S). The effect of C on weight update performance (with "M" = C*R*S) is mentioned in Batch Size, Height And Width. In short, larger values usually give higher efficiency, with diminishing returns.

The number of channels of input and output can have a more immediate impact on performance, however; the GEMM dimension "N" is equal to C or K for all of forward convolution, activation gradient calculation, and weight gradient calculation. Thus, your choosing of these parameters can have a direct impact on performance.

Similar behavior can be seen in forward convolution and weight gradient computation as varied across K (Figure 25). In both cases, "N" = K, resulting in a strong trend for small values of K and diminishing returns once K is larger than most tile sizes.

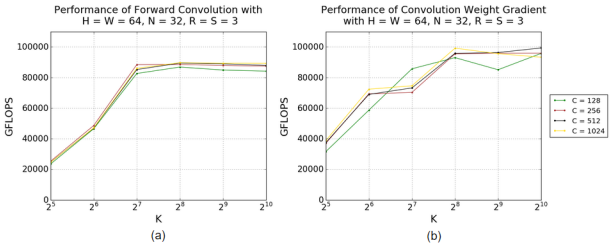

Figure 25 Forward convolution and weight gradient computation performance is much better for larger K, up to a point. Tesla V100-SXM3-32GB GPU, cuDNN v7.5.

The same effect is present for channels of input in activation gradient computation ("N" = C), as seen in Figure 26. As previously mentioned, the effect of C on activation gradient computation tends to match the effect of K on forward convolution.

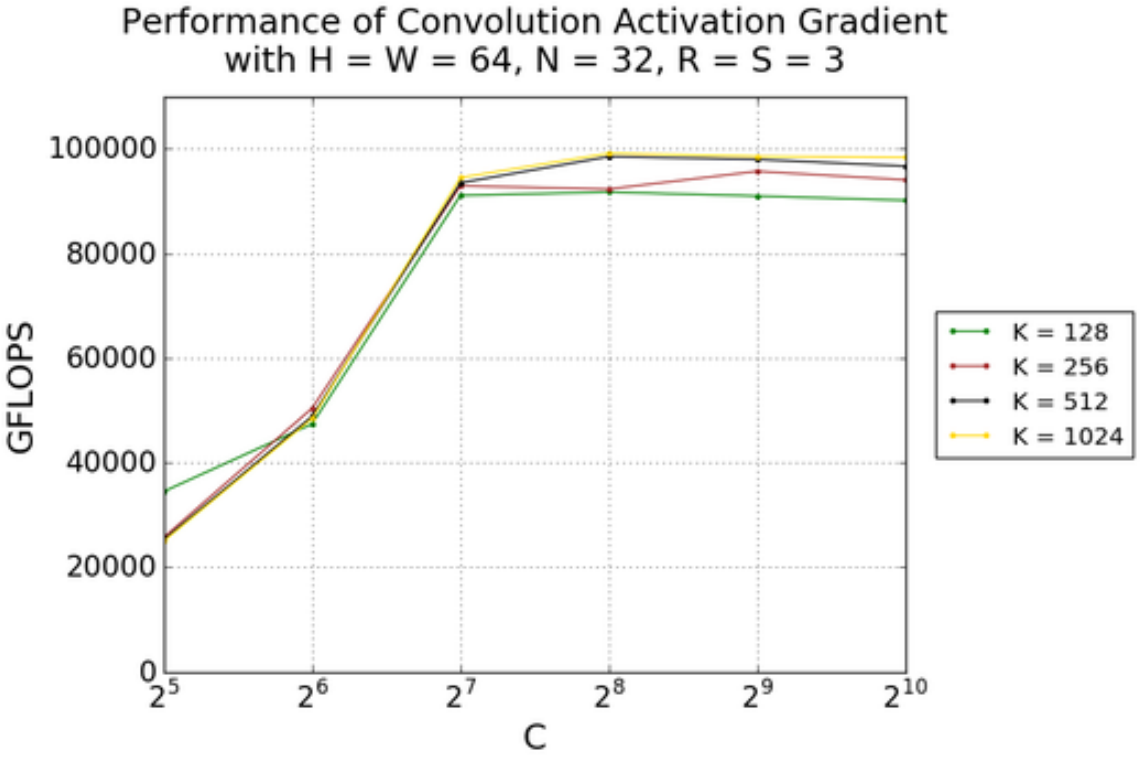

Figure 26 Activation gradient calculation performance improves as C increases, with diminishing returns. Tesla V100-SXM3-32GB GPU, cuDNN v7.5.

7.3.5. Strides

Filter strides (U and V) impact performance mostly via their effect on input and output tensor dimensions. Using a horizontal and vertical stride of 1, H and W are roughly equal to P and Q respectively, depending on the filter size and padding. However, when

larger strides are used, there is a several fold difference in the size of input and output feature maps. In turn, this impacts GEMM dimensions and performance.

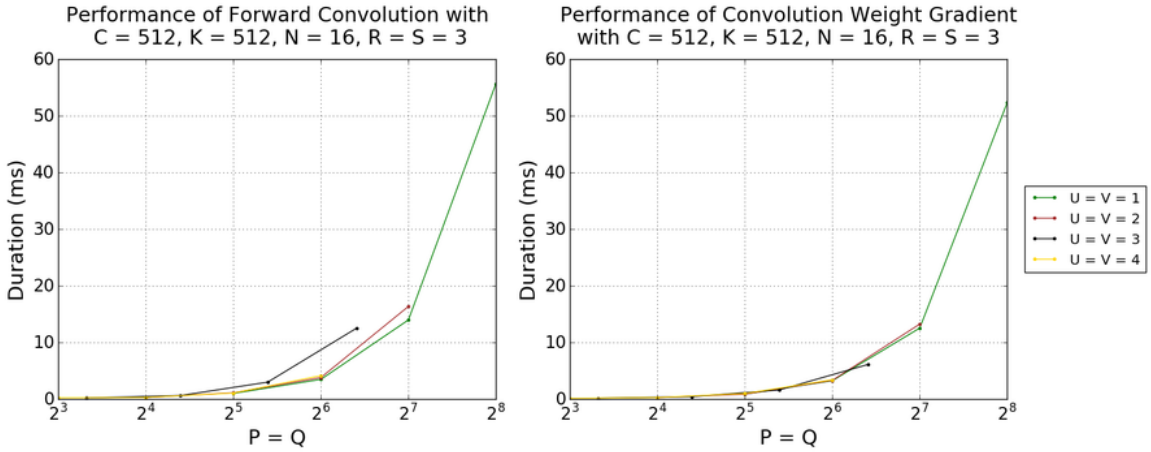

Figure 27 Performance of forward convolution and weight gradient calculation is relatively unaffected by variations in stride or input height and width as long as output height and width are constant. Tesla V100-SXM3-32GB GPU, cuDNN v7.5.

Forward convolution and weight gradient calculation perform similarly for equal P and Q, although H, W, U, and V vary (Figure 27).

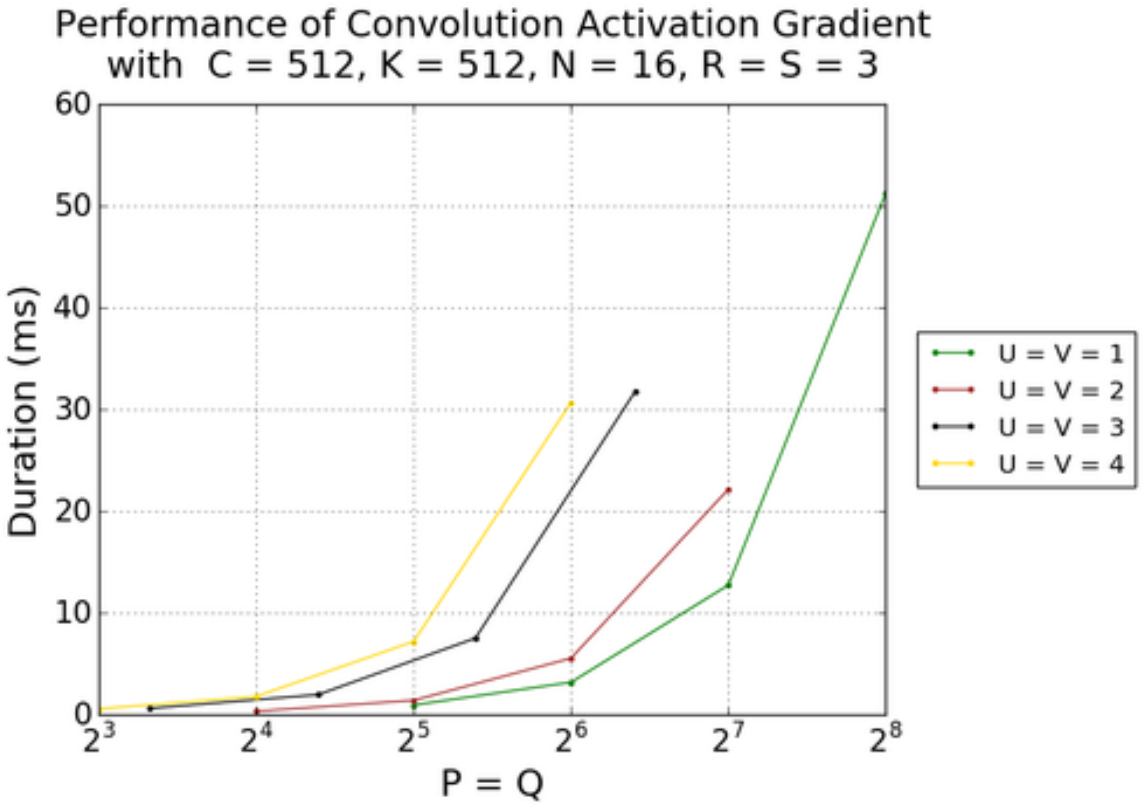

Figure 28 Activation gradient computation performance is not exclusively governed by output dimensions; input height and width have an impact. Tesla V100-SXM3-32GB GPU, cuDNN v7.5.

In contrast, activation gradient calculation is affected more heavily by input feature map size (Figure 28).

7.3.6. High-Performance Example

An example of a convolution with high performance is shown in Figure 29. This scenario is based on a convolutional layer with input feature maps of size 32x32, filters of size 5x5, 1024 input channels, and 1024 output channels; each parameter is reasonably large. NHWC data is used, to avoid overhead from additional transposes. Input and output channel numbers are divisible by 8, so Tensor Cores will be enabled.

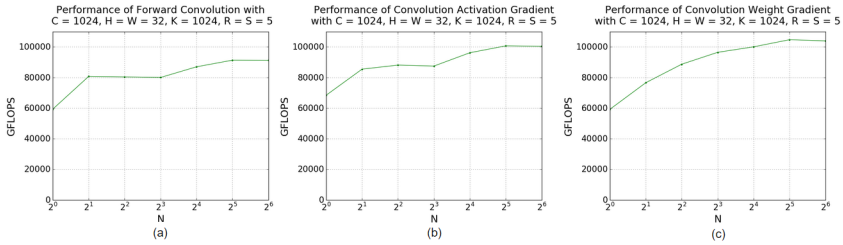

Figure 29 Good performance can be seen for the convolutions with larger batch sizes here, above 100 TFLOPS for activation and weight gradient computation. Tesla V100-SXM3-32GB GPU, cuDNN v7.5.

With batch sizes of 32 or 64, performance for this case exceeds 100 TFLOPS for activation and weight gradient calculation. Forward convolution performs at just over 90 TFLOPS.

7.4. Convolution Variants

7.4.1. Dilated

Dilated convolutions are a variant of a regular convolutional layer that effectively expands the filter being applied by inserting zeros between filter elements.

The dilation factor is one greater than the number of zeros added between each pair of elements. As a result, the overall 2D area overlapping with each channel of the filter increases.

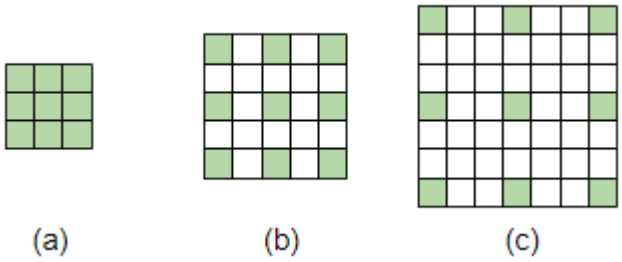

Figure 30 Dilation of a 3x3 filter; dilation factors are (a) 1, (b) 2, and (c) 3.

$$r_{effective} = dilation_h \cdot (r - 1) + 1$$

$$s_{effective} = dilation_w \cdot (s - 1) + 1$$

Choice of dilation factor affects how a convolution is represented as a virtual GEMM, but does not actually change the dimensions of that GEMM; therefore, performance for forward convolution and activation gradient computation is similar regardless of dilation factor (Figure 31).

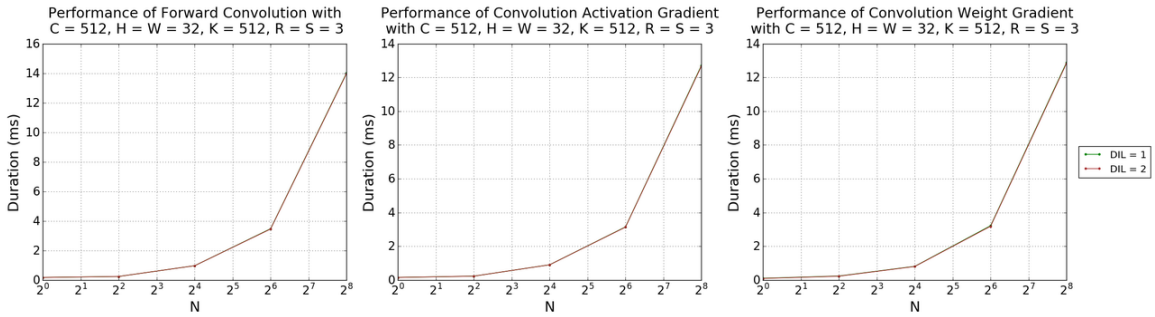

Figure 31 Performance of forward convolution, activation gradient computation, and weight gradient computation is virtually identical for a non-dilated convolution and one with a dilation factor of 2. Tesla V100-SXM3-32GB GPU, cuDNN v7.5.

7.4.2. How This Guide Fits In

NVIDIA's GPU deep learning platform comes with a rich set of other resources you can use to learn more about NVIDIA's Tensor Core GPU architectures as well as the fundamentals of mixed-precision training and how to enable it in your favorite framework.

The Tesla V100 GPU architecture whitepaper provides an introduction to Volta, the first NVIDIA GPU architecture to introduce Tensor Cores to accelerate Deep Learning operations. The equivalent whitepaper for the Turing architecture expands on this by introducing Turing Tensor Cores, which add additional low-precision modes.

The Training With Mixed Precision Guide describes the basics of training neural networks with reduced precision such as algorithmic considerations following from the numerical formats used. It also details how to enable mixed precision training in your framework of choice, including TensorFlow, PyTorch, and MxNet. The easiest and safest way to turn on mixed precision training and use Tensor Cores is through Automatic Mixed Precision, which is supported in PyTorch, TensorFlow, and MxNet.

Chapter 8. MEMORY-LIMITED LAYERS

8.1. Normalization

Normalization layers are a popular tool to improve regularization in training.

There are many variants of normalization operations, differing in the "region" of the input tensor that is being operated on (for example, batch normalization operating on all pixels within a color channel, and layer normalization operating on all pixels within a mini-batch sample). All these operations have very similar performance behavior; they are all limited by memory bandwidth. As an example, let's consider batch normalization.

8.1.1. Batch Normalization

Batch normalization (BN) layers take a 4D (NCHW or other layout) tensor as input, and normalize, scale, and shift all the pixels within each channel C. In most convolutional neural networks, BN layers follow after a convolutional layer.

Batch normalization does not have enough operations per value in the input tensor to be math limited on any modern GPU; the time taken to perform the batch normalization is therefore primarily determined by the size of the input tensor and the available memory bandwidth.

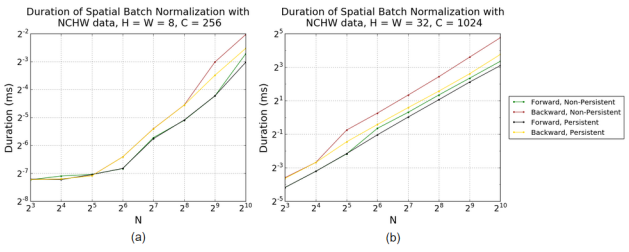

Figure 32 Duration of spatial and persistent spatial batch normalization in two different regions with NCHW data. Note that both axes are logarithmically scaled. Tesla V100-SXM3-32GB GPU, cuDNN v7.5.

When input tensors are very small, duration does not change with input size (Figure 32 (a), with batch size below 64). This is due to tensors being small enough that memory bandwidth isn't fully utilized. However, for larger inputs, duration increases close to linearly with size (Figure 32 (b)); it will take twice as long to move twice as many input and output values.

Two different algorithm options are benchmarked here. Non-persistent batch normalization is a multi-pass algorithm, where input data will be read one or more times to compute statistics such as mean and variance, then read again to be normalized. When inputs are small enough, a better single-pass algorithm (persistent batch normalization) can be used by cuDNN - here, inputs are read once into on-chip GPU memory, and then both statistics computation and normalization is performed from there, without any additional data reads. Fewer data reads result in reduced traffic to off-chip memory, which - for constant bandwidth - means the duration is reduced. In other words, spatial persistent batch normalization is faster than its non-persistent variant.

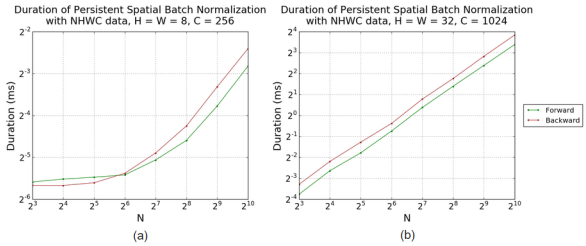

Figure 33 Duration of persistent spatial batch normalization in two different regions, this time with NHWC data. Note that both axes are logarithmically scaled. Tesla V100-SXM3-32GB GPU, cuDNN v7.5.

Similar trends can be seen when NHWC data is used.

8.2. Activations

Activation functions typically follow a fully-connected, convolutional, or recurrent layer in a network. These functions are applied to each activation value independently, hence we refer to them as "element-wise" operations. The shape of the activation tensor remains unchanged.

For details on a possible implementation of activation functions, see the documentation for cudnnActivationForward and cudnnActivationBackward. Deep learning frameworks such as TensorFlow and PyTorch often use their own (non-cuDNN) implementations and libraries to execute activation functions, for example via the Eigen library in TensorFlow 1.13. Regardless of implementation, the general performance behavior of activation functions (and other element-wise operations) on the GPU is always the same, as we'll describe below.

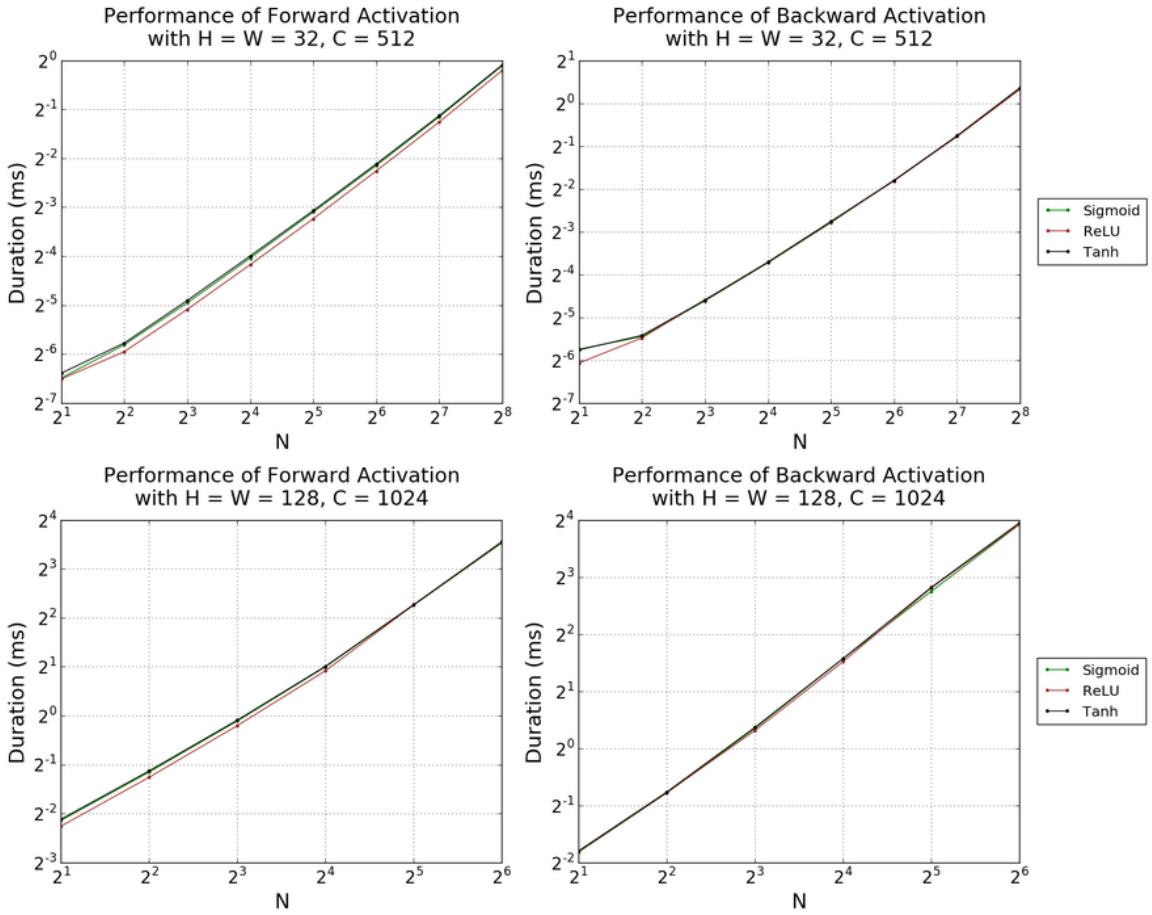

Figure 34 Duration of forward and backward propagation of activation functions is proportional to input size (N*H*W*C here). Note that both axes are scaled logarithmically. Tesla V100-SXM3-32GB GPU, cuDNN v7.5.

These functions involve few enough calculations that their forward and backward propagation speed is dictated by memory bandwidth. Sigmoid, ReLU, and tanh functions all rely only on the individual activation value, and so have very similar memory access requirements and thus performance. Figure 34 shows that duration is consistently proportional to the number of activations (here, N*H*W*C). Consequently, reducing the number of activations is the only way to speed up activation functions. As an exception to this rule, very small activation tensors may not be transferring enough data to and from memory to saturate bandwidth; this behavior is visible for the smallest batch sizes in the backward activation chart for the H=W=32 case shown in Figure 34.

8.3. Pooling

Pooling layers are commonly used in neural networks to introduce robustness to small spatial variations in the input, and to reduce the spatial dimensions (height and width) of the activations flowing through the neural network. They are defined by the dimensions of the input (N x H x W x C), the type of pooling (for example, maximum or average across the activations inside the pooling window), the size and shape of the

pooling window (**win_h** and **win_w**), and stride between applications of the pooling window (U and V).

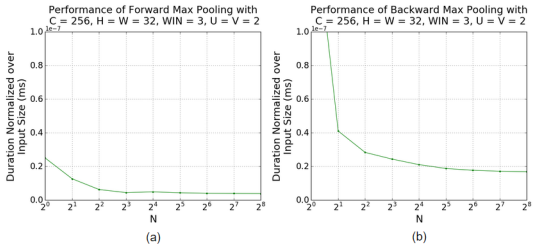

Figure 35 Duration becomes proportional to input size for larger dimensions. Performance differs for forward and backward propagation of pooling operations. Note that duration is normalized over N*H*W*C here. Tesla V100-SXM3-32GB GPU, cuDNN v7.5.

Most pooling operations practically used in deep neural networks are memory bound, as they do not perform enough computation per element to amortize the cost of reading the input and writing the output: Even in an ideal (forward pass) implementation, each element is re-used only **win_h * win_w** times. Hence, most practical implementations focus on maximizing the achieved memory bandwidth utilization, and both implementations shown in Figure 35 reach bandwidth saturation for large-enough inputs. Once saturation is achieved, execution time is directly proportional to the size of the input tensor.

Chapter 9. CASE STUDIES

9.1. Transformer

9.1.1. Basics

Transformers are a popular neural network architecture used for sequence-to-sequence mapping tasks, for example for natural language translation. They use an encoder-decoder architecture making heavy use of attention, both to "self-attend" over input sequences, as well as to give the decoder access to the encoder's context. Figure 36 shows the complete neural network architecture (Attention Is All You Need 2017 paper, page 3).

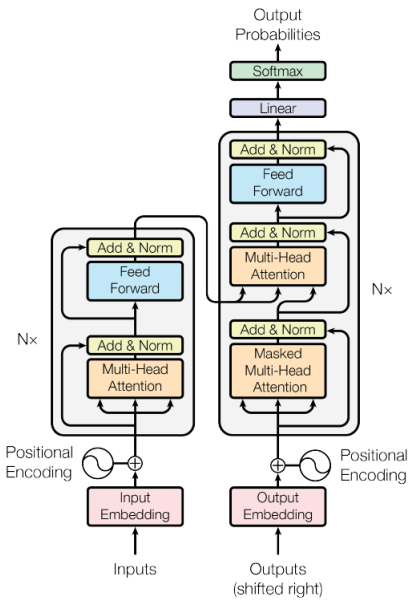

Figure 36 Transformer neural network architecture with N macro-layers in the encoder and decoder, respectively. Macro-layers consist of attention layer(s) and a feed-forward network.

From a performance standpoint, Transformers fundamentally process all the tokens in an input sequence in parallel, unlike - for example - RNN architectures with their sequential dependency. That makes Transformers very amenable to highly parallel architectures such as GPUs, and leads to large GEMMs that, with a few simple guidelines, can take great advantage of Tensor Core acceleration.

9.1.2. Applying Tensor Core Guidelines

9.1.2.1. Step 1: Padding The Vocabulary Size

Consider the final linear layer in the Transformer network, whose number of outputs is equal to the vocabulary size, as it is feeding the final SoftMax layer in the network to produce a probability distribution across tokens in the vocabulary.

This linear layer, as discussed in the Fully-Connected Layer section, has M equal to the vocabulary size, N equal to the batch size, and K equal to the input feature size (all in the forward pass). Because the vocabulary is usually large, this is a heavyweight computation, and it is important to ensure Tensor Cores are being used effectively.

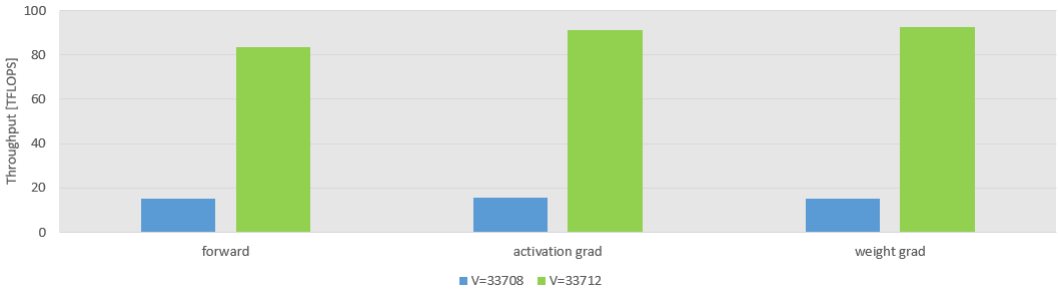

Figure 37 Performance benefits substantially from activating tensor cores by choosing vocabulary size to be a multiple of 8. The projection layer uses 1024 inputs and a batch size of 5120. Tesla V100-DGXS-16GB GPU, cuBLAS 10.1.

Figure 37 shows what happens when the vocabulary size is chosen without regard to the multiple-of-8 rule (V=33708): Tensor Cores cannot be applied, and performance reduces drastically to the levels sustained by the CUDA cores. Simply adding four padding tokens (to reach V=33712) to ensure a multiple-of-8 size can dramatically accelerate the overall computation.

9.1.2.2. Step 2: Choosing Multiple-Of-8 Batch Sizes

Besides the projection layer near the end of the network, fully-connected layers are a major Transformer building block in all other parts of the network as well, including the big self-attention and feed-forward blocks. As described before, batch size directly maps to one of the GEMM dimensions in such layers - N in the forward and activation gradient passes, K in the weight gradient pass - and therefore, the guideline to enable Tensor Cores by padding to a multiple of 8 applies to batch size as well.

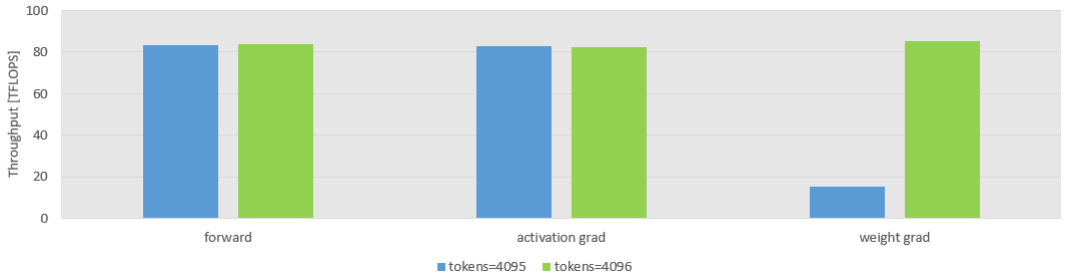

Figure 38 Fully-connected layer performance benefits from padding batch size to be a multiple of 8. The first FC layer (4096 outputs, 1024 inputs) from the Transformer feed-forward network is shown. This example illustrates that the multiple-of-8 padding rules sometimes don't have to be applied to all three dimensions, as forward and activation gradient computations perform the same with and without padding. The weight gradient pass shows the expected performance difference between CUDA cores and Tensor Cores. Tesla V100-DGXS-16GB GPU, cuBLAS 10.1.

The effect from padding batch size on one of the fully-connected layers in the network is shown in Figure 38. Here, we've picked the first layer in the feed-forward block, which is an FC layer with 1024 inputs and 4096 outputs. As the chart shows, this is an example where the multiple-of-8 rule does not necessarily need to be applied to all three GEMM dimensions; both forward and activation gradient passes perform the same with and without padding. The weight gradient pass, on the other hand, shows the same dramatic performance difference we saw on the projection GEMM earlier: With a batch size of 4095 tokens, CUDA cores are used as a fallback, whereas a batch size of 4096 tokens enables Tensor Core acceleration. As a rule of thumb, it is still recommended to ensure all three GEMM dimensions are multiples of 8 when training in FP16.

9.1.2.3. Step 3: Avoiding Wave Quantization Through Batch Size Choice

Because batch size directly controls the shape of the MxN output matrix and Tensor Core GEMMs are parallelized by tiling the output matrix, choosing batch size appropriately can be used to reduce tile and wave quantization effects.

For Transformer, let us consider the first layer in the feed-forward block again (4096 outputs, 1024 inputs). In this layer, the output matrix is of shape 4096 x batch size. Assuming a tile size of 256x128 as an example, the M=4096 dimension results in 4096/256=16 thread block tiles stacked vertically. On a Tesla V100 GPU with 80 SMs, wave quantization is minimal if the total number of thread blocks is a multiple of 80 (or just below). Therefore, choosing batch size to result in n*80/16=n*5 thread block tiles in the N dimension achieves optimal wave quantization. With 256x128 thread blocks, this is achieved by choosing batch sizes of N=1*5*128=640, N=2*5*128=1280, and so on. Figure 39 illustrates the effect this has using two common batch sizes, 2048 and 4096.

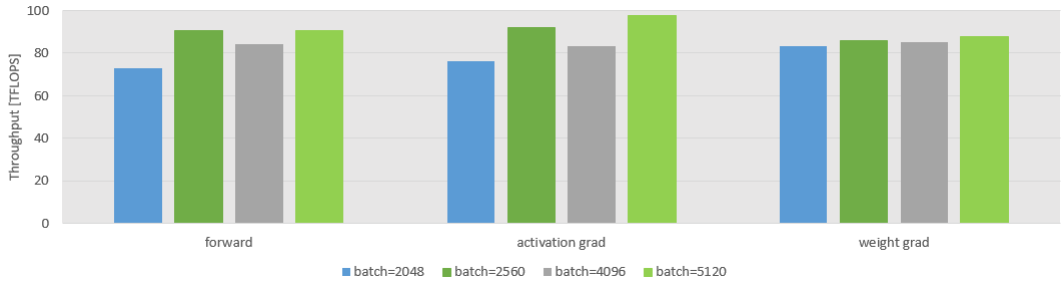

Figure 39 Fully-connected layer performance benefits from eliminating wave quantization by choosing batch size appropriately. The first FC layer from the feed-forward block is shown as an example. Batch sizes 2560 and 5120 result in a multiple of 80 thread blocks running on an 80-SM Tesla V100 GPU. In the weight gradient, batch size maps to the K parameter of the GEMM and hence does not control the shape of the output matrix or have any immediate effect on wave quantization. Tesla V100-DGXS-16GB GPU, cuBLAS 10.1.

The chart shows that choosing a quantization-free batch size (2560 instead of 2048, 5120 instead of 4096) can noticeably improve performance. In particular, it is noteworthy that batch size 2560 (resulting in 4 waves of 80 thread block tiles each, assuming 256x128

tile size) *achieves higher throughput than the larger batch size of 4096* (512 thread block tiles, 6.4 waves with 256x128 tile size). The activation gradient with batch size 5120 achieves about 100 TFLOPS delivered performance. For the weight gradient computation, batch size maps to the K parameter of the GEMM, and hence does not directly influence the size and shape of the output matrix or the number of thread block tiles that are created.

Notice

THE INFORMATION IN THIS GUIDE AND ALL OTHER INFORMATION CONTAINED IN NVIDIA DOCUMENTATION REFERENCED IN THIS GUIDE IS PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE INFORMATION FOR THE PRODUCT, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the product described in this guide shall be limited in accordance with the NVIDIA terms and conditions of sale for the product.

THE NVIDIA PRODUCT DESCRIBED IN THIS GUIDE IS NOT FAULT TOLERANT AND IS NOT DESIGNED, MANUFACTURED OR INTENDED FOR USE IN CONNECTION WITH THE DESIGN, CONSTRUCTION, MAINTENANCE, AND/OR OPERATION OF ANY SYSTEM WHERE THE USE OR A FAILURE OF SUCH SYSTEM COULD RESULT IN A SITUATION THAT THREATENS THE SAFETY OF HUMAN LIFE OR SEVERE PHYSICAL HARM OR PROPERTY DAMAGE (INCLUDING, FOR EXAMPLE, USE IN CONNECTION WITH ANY NUCLEAR, AVIONICS, LIFE SUPPORT OR OTHER LIFE CRITICAL APPLICATION). NVIDIA EXPRESSLY DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY OF FITNESS FOR SUCH HIGH RISK USES. NVIDIA SHALL NOT BE LIABLE TO CUSTOMER OR ANY THIRD PARTY, IN WHOLE OR IN PART, FOR ANY CLAIMS OR DAMAGES ARISING FROM SUCH HIGH RISK USES.

NVIDIA makes no representation or warranty that the product described in this guide will be suitable for any specified use without further testing or modification. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer's sole responsibility to ensure the product is suitable and fit for the application planned by customer and to do the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer's product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this guide. NVIDIA does not accept any liability related to any default, damage, costs or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this guide, or (ii) customer product designs.

Other than the right for customer to use the information in this guide with the product, no other license, either expressed or implied, is hereby granted by NVIDIA under this guide. Reproduction of information in this guide is permissible only if reproduction is approved by NVIDIA in writing, is reproduced without alteration, and is accompanied by all associated conditions, limitations, and notices.

Trademarks

NVIDIA, the NVIDIA logo, and cuBLAS, CUDA, cuDNN, cuFFT, cuSPARSE, DALI, DIGITS, DGX, DGX-1, Jetson, Kepler, NVIDIA Maxwell, NCCL, NVLink, Pascal, Tegra, TensorRT, and Tesla are trademarks and/or registered trademarks of NVIDIA Corporation in the Unites States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright