Mini-Project 2: GPU Performance Modelling

Alexander Graening (804732740), William Chong (205114665)

Introduction

Our main goals for this project were to design a working GPU performance modeler for convolution and to gain further insight on the architectural mechanisms that accelerate ML inference on GPUs. Specifically, we sought to model performance on an NVIDIA Titan V GPU for a method using im2col and tiling to handle convolutions. We used this model to predict performance of the convolution layers in mini-project 1 and validated our model against the spreadsheet of cuDNN results.

Originally, we planned to implement the default project of modelling cuDNN performance. Due to not finding good documentation on how cuDNN models work, we adjusted this plan to focus on modelling performance of using a GEMM on the result of running im2col on the convolution layer, along with tiling based on the size of shared memory. We based this on the CUTLASS article linked in the Resources section [1]. One major limitation of our model is that we used reported specs and used them to calculate performance for our model. This required additional assumptions that degraded the quality of our model.

This report is divided into 6 sections in addition to the introduction. In the Design Process section, we describe choices we made and challenges we faced. In the Explanation section, we describe our final model along with assumptions the model is based on. In the Validation section, we compare our model's results to those for cuDNN using the spreadsheet provided for the first mini-project. In the Architecture Insights section, we discuss improvements to the GPU that would improve performance on the examples we tested as predicted by our model. We have a Discussion and Future Improvements section where we describe improvements and ways we would approach the same project differently in the future. Finally, we have a Resources section where we include informal citations in the form of links to some of the resources we used for the specs we used to calculate performance for our model. The Explanation, Validation, and Architecture Insights sections are meant to correspond to the descriptions in the project spec, while the other sections provide additional details.

Design Process

Initial Considerations

To start the project, we first implemented the 3-bound roofline approach suggested in the project specifications where we took the maximum of compute, memory, and buffer bound times. We then modified and added further modeling functions to this framework as we progressed. We included a variety of parameterizations for our functions such as word size and batch size. We also found some differences between using the reported performance numbers and performance numbers calculated from other sets of more primary specifications, so we parameterized this and left two selectable options.

The first option was to use the general performance values given in the Titan V documentation [2] and work backwards from those values to infer simulated performance given user inputs. The second was to find more detailed and specific throughput, latency, and bandwidth values for each hardware component (e.g. L2 to L1 bandwidth, FP16 Mult-add operation throughput). Both of these approaches were somewhat faulty, in different ways. The first, which we assume was based on empirically measured performance collected by NVIDIA engineers, abstracted away many of the finer-grained detail of the architecture's performance. This led to the numbers lending less utility towards detailed modeling techniques. We believe the second approach yielded us theoretical values. Moreover, they were difficult to find and incomplete/vague information. NVIDIA's official documentation was pretty sparse [2][3], so we ended up drawing from the hardware specs other modelling papers' had derived or found [4][5]. Fortunately for us, at the beginning we decided to initialize our model with a list of constants representing hardware performance/specs, allowing for us to fine-tune our model's primitives as we progressed. We also created classes for GEMM and Conv layers and wrote our functions to handle GEMM and Conv differently. This parameterization gives future users the flexibility to update the existing parameters, or design/import different GPU architecture parameters easily.

Base Modelling Functions

Building on the values we found, we built our base functions time_to_execute(), time_to_load_from_buffer(), and time_to_load_from_memory(). We parameterized each of these functions with the precision and size of the data used. Initially, these functions were overly simplistic--for example, it assumed memory bandwidth between any cache level on the card was the same ~650 GB/s value. But as our model progressed, we increased each function's accuracy and decreased their scope, in order to provide more granular information to higher functions. We also added more functions to better model the memory levels, sizes, bandwidths, and access latencies.

We made a few assumptions here: first, we assumed that the Titan V card interfaces with the CPU with a PCIe 3.0x16 bus interface. We did not have specific information on the configuration of the Tetracosa system's hardware, but since the NVIDIA Titan V has a PCIe 3.0x16 bus interface, we believe it is a reasonable logical step to assume that it is not using an older, slower PCI interface. Another assumption we made was that the system was equipped with higher-end DRAM (e.g 3600 MHz with CAS first word latency of 8.89 ns). These assumptions helped us to better model the CPU main memory to GPU transfer speeds, specifically DRAM access latency, main memory bandwidth, and PCIe latencies [6]. We also

modeled memory access latencies with a simple lookup function. This included L1/shared, L2, VRAM, and main memory latencies which we found and compiled from two research paper sources [4][5].

We also added kernel startup time, which we calculated using a kernel containing a single add operation, with no explicit memory loads or returns, and used nvprof to profile the time it took to startup and run this kernel. We added the add operation as a trivial computation, to prevent the compiler from detecting our "empty kernel" and writing it out. After running it several times, we found that the lowest profiled time was 1.216 us.

Implementing im2col and Tiling Operations

We added further complexity by implementing im2col for convolution layers and tiling for GEMM kernels that do not fit in shared memory. For im2col, we decomposed the convolution layer into an GEMM where the inputs were stored in an m x k matrix and the weights are stored in a k x n matrix. This resulted in large matrices that added some increased overhead for loading to memory. To deal with this size, we needed to add tiling. For tiling, we assumed that a set of rows from the m x k matrix and a set of columns from the k x n matrix must be loaded along with the output m x n values. When we move from one tile to the next, we only need to replace the row or columns that changed along with the outputs. Since we assumed that the tiling operation was done across the shared cache from L2, we computed optimal tiling shaping with respect to the shared memory size parameter. Using this, we found the total cost to load tiles from L2 into shared (scratchpad) memory.

Model Iterations

Our general model built on these constants and primitive cost functions in an iterative manner. Our first implementation was a basic roofline model with simple, idealized computation and memory models. With this, we were able to at least begin generating execution time predictions, and allowed us to improve each of the three components separately, without too much worry of inter-model component complexity. This roofline model took the maximum of compute and memory times. Here, compute time assumed perfect computation and complete independence from memory access costs. The memory model assumed perfect reuse, and thus only required DRAM access cost calculations.

From this, we added the aforementioned im2col and GEMM tiling optimizations to our base model. Along with this, we implemented and integrated our more accurate, complex memory model. In particular, the tiling operation incorporates the use of a more complex L2 to shared memory cost computation. We tuned this model by choosing to tile across the shared memory and further refined parameter choices and base constants.

Explanation of Final Design

Our final model takes the execution time as the maximum value of the compute bound time, the memory load time, and the time to load the L1 shared memory based on the tiling the GEMM resulting from running im2col on a Conv layer. Our model used average throughput numbers reported on NVIDIA documentation and a few third party analyses (see Resources) instead of chaining together times for different computations that take a fixed amount of time. For example, we would treat running a certain number of computations as requiring the average time to perform those computations instead of calculating the time it takes to complete each group of parallel computations. We checked our model against the numbers in the spreadsheet with cuDNN results provided for project 1 and used the same layer sizes as a starting point. We made a few assumptions with this model:

1. The entire layer fits in device memory. This works for the layers with batch size 1 on the spreadsheet. For larger batch sizes on the spreadsheet, this assumption holds to some extent since we can load new inputs as we finish the earlier inputs in the batch.
2. Loading device memory is not included in compute time. This assumes either there is sufficiently intelligent memory management so there is no need to stall during computation for memory loads or that the empirical performance numbers already account for initial loading of memory.
3. Convolution layers are computed using im2col to convert to a GEMM and tiled within the GEMM to fit within shared memory.
4. Our results assume that the "Tensor Advantage" number on the cuDNN result spreadsheet can be used as a conversion ratio between the time without tensors and the time with tensors.

Validation

We tested our model on the Conv1 and Conv2 layers from mini-project 1 for several batch sizes. Likely due to the weak correspondence between our model and cuDNN, our model had substantial errors for Conv2 and was incapable of generating results close to the expected values for Conv1.

w	h	c	n	k	Time (usec)	Tensor Advantage	No Tensor	Our Model (usec)	Error (%)
14	14	512	1	512	253	1.28	324	4010	1137
224	224	64	1	64	152	2.24	340	394	15.9
224	224	64	2	64	262	2.39	626	488	22.0
224	224	64	4	64	467	2.51	1172	675	42.4
224	224	64	8	64	916	2.11	1933	1050	45.7
224	224	64	16	64	1916	1.08	2069	1798	13.1
224	224	64	32	64	3590	1.15	4129	3296	20.2
224	224	64	64	64	7225	1.15	8309	6309	24.1

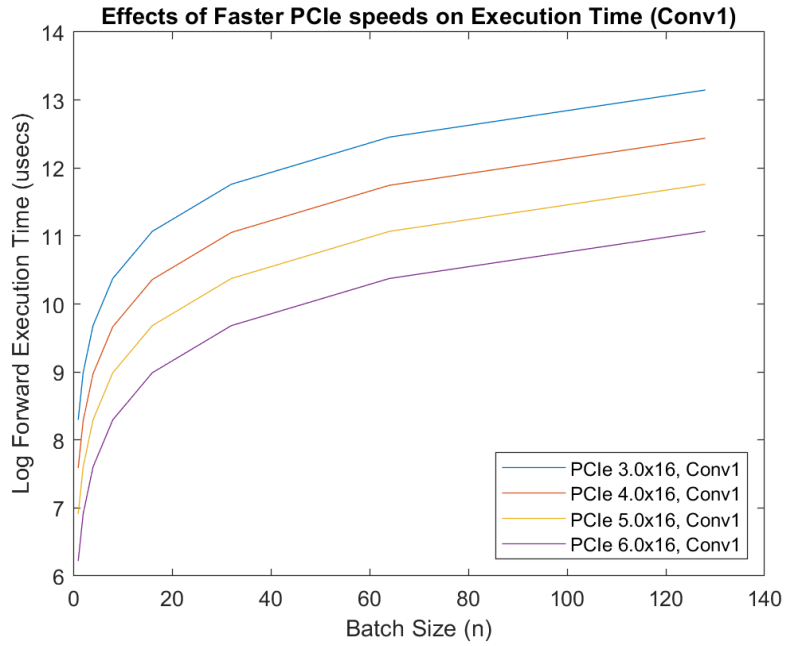
Figure 1: Result Comparison. This table shows our results for a 3x3 filter compared to the corresponding results on the cuDNN result spreadsheet. The column giving the time with no tensors is calculated by multiplying the time with tensors by the tensor advantage number. Note that all of these results were bounded by memory access time, with the exception of Conv2's 64 and 128 sized batches, which were bounded by buffer time.

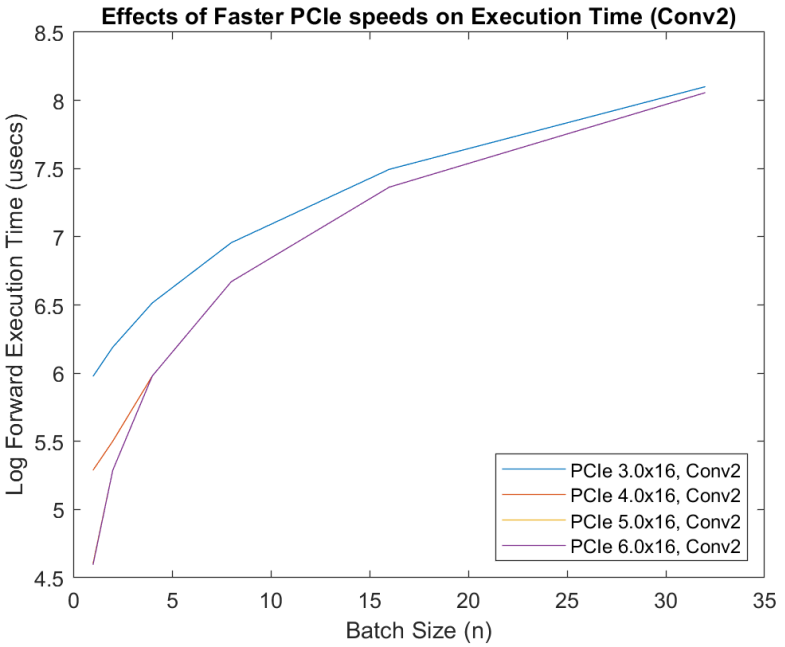
Architecture Insights

According to our model, the layers we examined were bound either by device memory bandwidth or by shared memory bandwidth. This implies that increasing bandwidth for loading to these levels of memory would improve the execution time. Our model appears to be reasonably good for small numbers of input channels and filter channels.

Since our model was mostly memory-bounded for im2col GEMM, we wanted to vary the PCIe version parameter and see how strongly it was tied to execution times. Essentially, we wanted to see how much wider PCIe bandwidth needed to be to ensure that the model was not memory-bounded, and how sensitive the model was. The execution time results are shown in **Figure 2** and **Figure 3**. Interestingly, the Conv1 layers (14, 14, 512, 512), even with wider PCIe bandwidths, remained memory-bounded for all batch sizes. However, execution times were quite sensitive to faster PCIe versions, and scaled directly with increasing bandwidth.

In contrast, the wider and shallower Conv2 layers (224, 224, 64, 64) quickly became buffer-bounded with increasing PCIe bandwidths. Note that in **Figure 3**, with an PCIe 4.0x16 interface, batch sizes of 4 and larger became bounded by shared-memory accesses. Faster PCIe versions (5.0 and 6.0) only slightly improved on this, where batch sizes 2 and onwards were buffer-bounded. This seems to demonstrate that for wider convolutional layers (e.g. Conv2), the memory access times exhibit diminishing returns with respect to increasing PCIe bandwidths. So, for a neural network with majority shallow, wide layers, a GPU architect could choose to instead optimize the shared-memory bandwidth and access times, as this would scale better over improvements to main memory bus interfaces. On the other hand, for applications with deep layers, a designer should consider upgrading the PCIe interface to a more modern version as these memory access costs should decrease linearly with PCIe bandwidth increases.





Figures 2 and 3: Comparing our model's Conv1 and Conv2 results with varying PCIe speeds. Conv1 layers have dimensions ((w, h, c), k) where w=14, h=14, and c=512 is the input shape, and k=512 number of 3x3 filters; Conv2 layers have dimensions, where input shape is w=224, h=224, c=64, and there are k=64 number of 3x3 filters. **Fig. 1** remains memory-bounded in execution time for all batch sizes and PCIe bandwidths. In contrast, **Fig. 2** shows that Conv2 quickly becomes largely buffer-bounded in execution time with PCIe 4.0x16 onwards. For PCIe 5.0 and 6.0, Conv2 is buffer-bounded for all batch sizes except for batch size 1.

Discussion and Future Improvements

We built our model around the specs we could find for the Titan V GPU online, but we may have been able to get better numbers experimentally using some sort of regression technique with carefully chosen test cases to determine better approximations for memory latencies at different levels and for computation times. With more time, we could have built a more complex model that would make fewer assumptions and possibly use smarter tiling.

Our model fell significantly short on the Conv1 model. The Conv1 model cuDNN results on the spreadsheet showed an interesting trend where execution time was not monotonically increasing for larger batch sizes. Our model does not come close to capturing this trend and was about an order of magnitude off for batch size 1.

Since our tiling optimization function is based solely on maximizing tile size within shared memory, we may be able to improve this by accounting for other features of memory such as L2, memory banks, and memory-access costs. These factors are likely taken into account with NVIDIA libraries.

The other project idea of using an ML model to predict performance seems interesting. A number of functions and GPU specifications are hidden to some degree and ML is good for approximating a black box. It may even be possible to incorporate this as a component of a larger model that accounts for design structures that we do know and uses well chosen features for training. This could be used to create a pretty powerful tool that could be used across multiple GPU models to help optimize kernel design with useful parameterization/design feedback.

Resources:

[1] CUTLASS: https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda/

[2] Titan V GPU Specs: https://www.nvidia.com/en-us/titan/titan-v/

[3] CUDA Toolkit Documentation (Section 5.4.1, Arithmetic Instructions): https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?fbclid=IwAR3_uznGWPcbbWDJMv2Gsa1VgqOCpfrBMoK7yJdhrIvvjz7DumnH4XDzlbY#arithmetic-instructions

[4] Exploring Modern GPU Memory System Design Challenges through Accurate Modeling: https://arxiv.org/pdf/1810.07269.pdf

[5] Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking: https://arxiv.org/pdf/1804.06826.pdf

[6] PCI Express High Speed Fabric https://www.dolphinics.com/download/WHITEPAPERS/Dolphinproductbrochure.pdf