CS259, Prof. Nowatzki (W. 20)
Matt Nicholas (305431314) & Bryan Bednarski (005428092)

Mini-Project 2: May 13, 2020

Github: https://github.com/bbednarski9/V100_Model

1. **Explanation:** Explain the chosen model, including rationale and any required tuning. Explain any challenges faced, and how you addressed them (or would address them). If your model contains arbitrary constants, please explain why they exist or what they might mean.

For this assignment, we elected to develop a mechanistic model of the Titan V GPU from NVIDIA. This model is extended from the Yalsa loop scheduler that was provided for this course. This goal is to compute the execution time of CUDNN convolution and fully connected kernels with various input dimensions. This model is designed to be simple, and to highlight the differences in execution time between the highly optimized CUDNN code, and the baseline computational and memory bounds that exist due to computational throughput and memory bandwidth of the system.

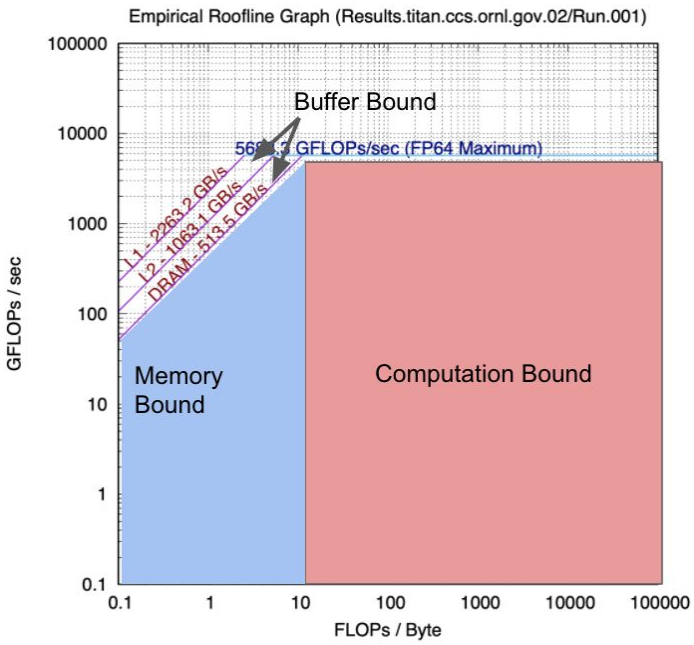

Figure 1: Roofline model for Titan V GPU generated by the Empirical Roofline Analysis Tool from Berkeley Labs.

Figure 1 above shows a modeled roofline model for the Titan V GPU that we use for this assignment, with 64-bit data representations. This figure provides a good visualization of the two possible bounds for the Titan V GPU: computational and memory. Kernels falling

beneath the red ‘DRAM’ line in this figure will be memory bound, as the bottleneck in execution will be due to a lack of data communication rates from DRAM/global memory to the functional units. Conversely, kernels falling beneath the blue ‘GFLOPS/sec’ line will be computationally bound as the data needed for computation is available to the functional units and the limiting factor is the speed of the processors in execution.

The above figure is a good visualization, but not completely accurate for the actual capability of the Titan V. ERT was used to generate the plot, and so the values are significantly below what the datasheet suggests the theoretical maximum would be. For this project, we will be considering 32-bit data representations for fully connected and convolutional kernel data. However, getting ERT to run for 32 bit was problematic, and so an actual value for TFLOPS was not gathered for 32 bit. As a result, the TFLOPS used for our device model was tuned, to arrive at an optimal parameter for predictive accuracy. We started by using values of 630 GB/sec and 18.7 TFLOPs for the global memory bandwidth and computational throughput, respectively. These values come from the datasheet, and we would expect real-world performance to be a bit slower.

The basic process for our model was to determine the computation time, and the memory time. The max of these two values were returned as our execution time.

1.1. Computational Bound Time:

In order to calculate the computation time, the number of floating point operations to be performed by each kernel is determined. This is determined by the input size of the matrices passed to the kernel. For GEMM, this is 2*m*n*k floating point operations. For convolution, this is 2*Nx*Ny*Nx*Ni*Kx*Ky (assuming stride of 1) floating point operations. Next, the number of computations is divided by the number of TFLOPS (18.7) to arrive at the time it would take to perform the number of computations required by the kernel. We started by using the theoretical maximum TFLOPS, which we would expect to be significantly faster than actual performance.

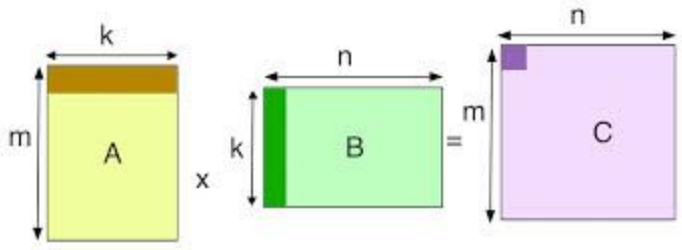

Figure 2: Fully connected layer, matrix dimension variables

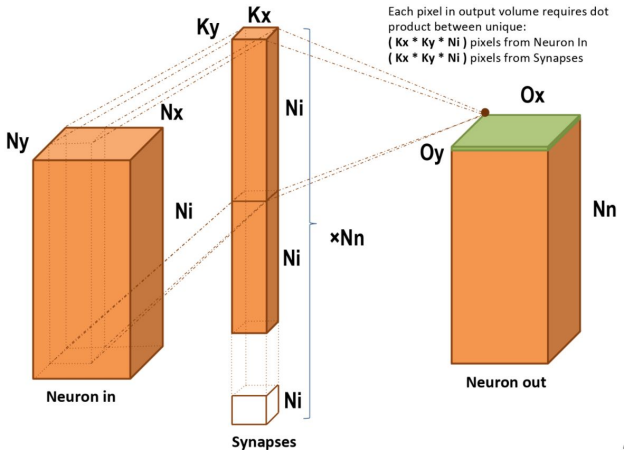

Figure 3: Convolution kernel layer dimensions

1.2. Memory Bound Time:

The function that was implemented to calculate memory bound time utilized the *yalsa* implementation of *bandwith_for_cache* to convert the required bandwidth from DRAM to the cache for various kernels and tiling ratios. The *bandwidth_for_cache* function takes these tiling loops into account by comparing which loop levels are able to fit into the L2 cache for a respective kernel, and determining the bandwidth required per cycle of computation. Originally, this function was implemented for the Diannao accelerator. However, by passing the *bandwidth_for_cache* function the appropriate data type size (4 bytes) and the number of computational iterations per second for this GPU (5120 cores with 1 computation each per cycle), we were able to compute an estimate for the bandwidth required per cycle. Scaling this by the operational frequency of the GPU, we derived a bandwidth in gigabytes per second required to prevent the system from being memory bound. We then create a ratio of this value to the memory bandwidth of the system and multiply that value by the expected computation time (if the ratio is greater than one) to estimate the memory bound time for a perfect memory system. If this ratio is less than 1, our system will still be computationally bound because the resultant memory bound time will be less than the original computational bound time. Note, this is not strictly a roofline analysis since the bandwidth determined for cache takes into account reuse. Thus, it is more of a lower bound.

1.3. Buffer Bound Time:

In modern GPUs, memory bandwidth bound times are a huge topic of consideration. The much lower latencies for memory transactions to L2, L1/shared memory hide this problem for higher level caches and buffers. The table below compares the sizes and latencies of the Titan V memory hierarchy. From these calculations we observe that due to the improved latency at each cache hierarchy level, our theoretical bandwidth at each successive level

from global memory is hidden, and continues to be with significant cache miss ratios[2]. This is also observable in the roofline model shown above. While the bandwidths are different for this simulation, we still see the same hierarchy of memory bandwidth bounds that is supported by our theoretical calculations shown below.

	DRAM	**L2 Cache**	**L1 Cache / Shared memory**
Size	12,037 MB	4.72 MB	64KB (for each of 80 SMs)
Latency	100nSec	53nSec	18nSec
Theoretical Bandwidth	653 GB/sec	1232 GB/sec	3627.7 GB/sec

1.4. Tunable Parameters

Titan V Model TunableParameters		
Name	**Value**	**Description**
memory_bw	630	DRAM theoretical bandwidth of 630 GB/sec
throughput	18700000000000	CUDNN-enabled kernel max theoretical throughput of 18.7 TFLOPS

We also allow for tiling within the convolution and GEMM kernels. Refer to the code for where and how these were implemented. Since we mainly used the theoretical max for throughput and memory_bw, we did not end up using tiling to its full extent, since our model was already too fast, and the tiling would reduce memory accesses. We believe that in using theoretical maximum values for computational throughput, and memory bandwidth we eliminated the need to incorporate active tiling in this model. However, with differently tuned memory bw and throughput, using tiling would likely result in more accurate estimates, since CUDNN almost certainly uses tiling.

1.5. Model Constants & Arbitrary Parameters

Titan V Model Constant Parameters		
Name	**Value**	**Description**
datatype_bytes	4	32-bit data representations require 4 bytes

iters_per_cycle	5120	1 computational iterations per cycle enabled by each of 5120 tensor cores across 80 streaming multiprocessors.
cache_bytes	4718592	L2 cache is 4.72MB
memory_bw	630	DRAM theoretical bandwidth of 630 GB/sec
freq_op	1.46	Titan V system clock is 1.46 Ghz
throughput	18700000000000	CUDNN-enabled kernel max theoretical throughput of 18.7 TFLOPS

2. **Validation:** For validation, explain the set of kernel parameters you use to validate your model. Then present an analysis that shows the error across some range of kernels. Explain what parameter settings the model performs well or poorly on.

Our model was evaluated on a number of different kernel inputs with varying batch size inputs. For Gemm we tested with pairings (M=25088 K=4096) (M=4096, K=1024) and then varied batch size from 1-512 for both. Additionally, we tested on a number of square inputs, with batch size of 1 (eg M=16384 K=16384, N = 1, M = 8192 K = 8192 N=1, etc). Our results were quite varied. When the batch size was 1 we seemed to get somewhat reasonable results. For the case where M = 25088 and K = 4096 we get execution time of 972 uS and the DeepBench actual is 1147 uS. Similarly, for M = 4096 and K = 1024 we get 40 uS and the actual is 74. Additionally, for other matrix inputs with batch size of 1, we generally seemed to be getting similar results, being faster than actual by 25-50%.

However, there was a much larger error when increasing the batch size. For instance, the DeepBench execution time stayed constant for M=25088 K=4096 for batch size 1 to 64 (execution time about 1150us for each). Similarly, for M=4096 K=1024 the execution time was about 85us from batch size 1 to 64. In contrast, our model's predicted execution time scaled linearly as the batch size (N) increased. This makes intuitive sense to us. Doubling the batch size doubles the number of computations that must be done, and also increases the amount of memory accesses that must take place. As a result, it seems reasonable for execution time to scale with batch size. This was not the case for DeepBench, and it is not until batch sizes greater than 64, for those that we tested (which also seems to be the case in the data that the processor gathered), that a noticeable increase in execution time is seen. This could be due to parallelization strategies in CUDNN, but is still confusing to us, since even with those batch levels there is enough parallelism for all the cores to be utilized. Thus we would expect execution time to go up when doubling batch size.

For convolution our results weren't as close percentage wise, but they followed the trend better for varying the input, batch, and filter size. When batch size was varied (keeping other parameters constant) we were pretty far off (about 70%), but the relative drop between our predicted execution time and actual is consistent. For instance, by varying from 64 to 32, the predicted execution time dropped to 12.5% of what it was previously. The actual execution time dropped to 18% of what it was previously. This amount varied, but the general trend in drop was matched by our predictive model. We believe that this can be somewhat satiated by tuning the model parameters (mainly TFLOPS) to reduce the computation speed, and be able to match the actual execution time better.

For the most part, our results are not great for convolution. We varied the input width/length, and the filter sizes, and we seem to be following the correct trend, but are significantly off in terms of error. We played with the tuning of the TFLOPS and bandwidth, but that mainly overfit it to specific inputs. Thus we decided to leave our results using the theoretical max TFLOPS and bandwidth in order to see the trend across all different variations in data.

We hoped to implement the L1 caches as well. For this we would have modeled it as a single cache, similarly to L2, but then multiplied it by the number of SMs to get the bandwidth requirement. We would take the max of L1 and L2 time to get memory bound time. However, we got caught up evaluating our initial results and tuning parameters and did not implement it. It's possible this could have helped with getting more accurate results.

3. **Architecture Insight:** Present an analysis of one or two parameters of your model that shed light on some aspect of the hardware itself. For example, does the peak computation throughput/memory bandwidth suggested by your model align with the datasheet. You might try varying the L2 cache size by 2x, or the compute throughput by 2x within the model to check the sensitivity. Which hardware parameters should future GPUs change if they want to perform better on convolution or matrix multiplication?

Parallel batch computation enabled by L2 Cache Size:

When running CUDNN kernel runtime simulations in DeepBench for fully connected layers, we observed that the forward (fwd) computation time for FC layers were not scaling linearly with the total number of computations like our naive model for the system was expecting them to. In our original design for this model, we elected to use a throughput constant value (Section 1.5) that mirrors the maximum throughput that has been observed through highly optimized CUDNN kernel code. Correspondingly, we chose a value for the number of computational iterations per cycle that assumed all cuda cores were active at all stages

regardless of kernel size. From the ‘Fully Connected Layer Results’ in the appendix, we see that this model attains semi-accurate results for batch sizes of 1 (for both sets of tests in ‘Batch Size Variance’) and scales accordingly, for different data sizes (separated by a large margin). As you can see from our DeepBench simulations for ‘Batch Size Variance’, the results across different batch sizes are generally the same for both the 25088x4096 and 4096x1024 kernels. We believe that this result is due to optimizations in CUDNN that allows for improved parallelization across kernels. This could be accounted for by the additional 640 Tensor Cores in the Titan V GPU that we did not model here. Additionally, it could be due to improved re-use of the weights matrix, which will be re-used frequently across all batches. Extensions of our model prompted by this discovery include modeling the Tensor Cores (if enabled) and restructuring our tiling loops to account for more re-use in weight matrices.

Cache Size

We doubled the cache size for a number of kernels and saw some interesting results. There did not seem to be a consistent speedup. Sometimes the timing was similar (often times when it was originally compute bound), and other times speedup was seen. It really depends on the input size. Having a larger cache doesn’t help when reuse isn’t exploited, and since we didn’t implement tiling a larger cache likely didn’t help as much as it could have.

GPUs are very good at doing convolution and gemm kernels. However, there is certainly room for improvement. Usually, this improvement comes with a tradeoff of being less generalizable. A lot of the accelerators we have studied are better at convolution and gemm, but can not be used for as many other applications as GPUs. GPUs will continue to get faster with Moore’s law naturally, and it will be interesting to see how much DNNs will play a role in the evolution of GPUs going forward.

References:

[1] T. Nowatzki, “Yalsa: Yet Another Loop Scheduler”, 2020, [Online] Available: https://github.com/PolyArch/yalsa

[2] M. Khairy, J. Akshay, T. Aamodt, T. Rogers, “Exploring Modern GPU Memory System Design Challenges through Accurate Modeling”, 2019. [Online]. Available: https://arxiv.org/abs/1810.07269

[3] SiSoftware, “NVIDIA Titan V: Volta GPGPU Performance in CUDA and OpenCL”, July 31, 2018. [Online]. Available: https://www.sisoftware.co.uk/2018/07/31/nvidia-titan-v-volta-gpgpu-performance/

Results: Download PDF from github

V100_modeling_results.pdf - Results for convolutional & fully connected layer DeepBench simulations and model evaluation is in github