Modeling CUDNN Classifier Layer on a TitanV GPU

Daniel Ahn Adrien Hadj-Chaib Josh Kimmel Hannah Nguyen

UCLA — CS 259 — May 2021

1 Explanation

In this project, we modeled the CUDNN classifier on a TitanV GPU. We referenced the NVIDIA spec document for publicly available parameters about the TitanV GPU, and made best-effort reasoning for parameters that we had to estimate, such as the L2 cache bandwidth. Our model can vary the dimensions of the input and output vectors to the classifier, the width and height of the computation, and the level of analysis that the model will do. We designed, implemented, and tested incrementally, changing assumptions about the memory hierarchy in order to look at different aspects of the architecture and how they affect the overall execution time. We discuss how assuming infinite and pre-populated registers, to infinite and pre-populated scratchpad, to L2 cache, then finally to GPU memory gets our model closer and closer to the CUDNN profiled performance. The code for the model is available at: `https://github.com/joshkimmel16/cudnn-titan-model`.

1.1 Machine-Specific Constants

We made use of the following constants as part of the model:

- `global_mem_cap`: The capacity (in MB) of GPU memory. This value is provided by the NVIDIA spec document.
- `global_mem_bw`: The bandwidth (in MB/s) of GPU memory to L2 cache. This value is provided by the NVIDIA spec document.
- `num_cores`: The number of CUDA cores. This value is provided by the NVIDIA spec document.
- `gpu_clock`: The frequency (in MHz) of the GPU clock. This value is provided by the NVIDIA spec document.
- `warp_size`: The number of threads in a single warp. This value is provided by the NVIDIA spec document.
- `num_sms`: The number of SM's. This value is provided by the NVIDIA spec document.
- `max_threads_block`: The maximum number of threads that can run in a single thread block. This value is provided by the NVIDIA spec document.
- `max_threads_sm`: The maximum number of threads that can run in a single SM. This value is provided by the NVIDIA spec document.
- `l2_cap`: The capacity (in MB) of the L2 cache. This value is provided by the NVIDIA spec document.
- `l2_bw`: The bandwidth (in MB/s) of the L2 cache to local L1 caches/scratchpads. This value is assumed to be 1000 MB/s.

- `constant_cap`: The capacity (in MB) of constant memory (we assume this means local L1 cache, but are unsure). This value is provided by the NVIDIA spec document.
- `shared_cap`: The capacity (in MB) of shared memory (we assume this means local scratchpad, but are unsure). This value is provided by the NVIDIA spec document.
- `val_size`: The size (in bytes) of individual elements in the computation. For this analysis, we assume to be working only with FP64 (8 bytes).
- `cpi`: The cycles per instruction of basic operations required in the computation (adds, multiplies, loads, stores). This value is uniformly assumed to be 1.
- `max_lat_hide`: The maximum percentage of memory latency that can be hidden by running threads that are not currently dependent on memory. This value is assumed to be 0.7.
- `sync_penalty`: The number of cycles required to synchronize writes across multiple SMs and/or thread blocks to a single memory location. This value is assumed to be 500 cycles.

1.1.1 Discussion of Assumptions

The above list describes several parameters that we chose to make assumptions about.

The L2 bandwidth assumption was based on research of other systems using a similar cache hierarchy. Given the ability to hide memory latency, a reasonable amount of error is tolerable here.

A more robust modeling tool would allow for users to define the value size (as opposed to fixing it as FP16). We recognize this limitation but chose to constrain our model as such.

CPI is an assumption that has a major impact on modeling results. A CPI of 1 is ideal, which is why we chose to use it to start. Presumably, FP instructions can in general be pipelined such that this might be achievable - especially considering the computation does not have any divides. However, it might be worth re-assessing how reasonable this assumption is.

The maximum latency hide percentage is another major assumption - especially for memory-bound workloads. However, the actual value used in the model depends on the concurrency within each thread block/SM. As such, this assumption in itself is unlikely to have a huge impact on model accuracy.

Finally, the synchronization penalty is another big assumption. We further assume that the thread blocker scheduler attempts to optimize such that synchronization is as limited as possible. Within such a framework, we ensure not to overestimate the impact of synchronization. That being said, for kernels with a large number of tiles, this assumption can have a noticeable effect on the predicted execution time and thus must continue to be refined.

1.2 User-Defined Constants

We made use of the following user-defined inputs as part of the model:

- `n_i`: The dimension of the input vector.
- `n_n`: The dimension of the output vector.
- `t_i`: The width of each tile in the computation.
- `t_n`: The height of each tile in the computation.

- `type`: The type of analysis to perform. Options are: REGISTER, SCRATCHPAD, L2, MEMORY.

1.2.1 Discussion of Analysis Type

We decided to offer various levels of analysis to users. REGISTER makes the assumption that each SM has an infinite number of registers that are pre-populated with the necessary data. SCRATCHPAD makes the assumption that each thread block/SM has infinite local L1 cache/scratchpad capacity that is pre-populated with the necessary data. L2 makes the assumption that the L2 cache has infinite capacity and is populated with all data for the computation. MEMORY simply assumes that main GPU memory has enough capacity to hold all data for the computation and that this data is already present.

We offer multiple analysis types because it was a useful exercise to implement each sequentially (relaxing assumptions along the way). Furthermore, users can glean certain insights about the workload by running the analysis at various levels. For example, large hops in execution time from one level to another indicate that the assumptions relaxed in that hop are likely limiting performance factors.

1.3 High-Level Flow

1.3.1 Stage 1: Parameter Computation

The model starts by computing various parameters that dictate CUDA arguments such as number of thread blocks, threads per thread block and more. Each computation is described below.

1.3.1.1 Number of Tiles

This is the total number of tiles in the computation and is determined as follows: $\frac{n_i}{t_i} * \frac{n_n}{t_n}$

1.3.1.2 Threads Per Tile and Elements Per Thread

These parameters work in lockstep. One is the number of threads in each tile. The other is the number of elements that each thread works on within a tile. The major underlying assumption is that there is a **1:1 mapping of tiles to thread blocks**. The algorithm to compute these values works as follows:

First, compute the maximum number of threads within the tile by multiplying the tile dimensions and assume the elements per thread is 1. Then, loop while the number of threads per tile is greater than the maximum number of threads for a thread block. At each iteration, divide the original thread count by the iterator (starting at 2) and update the elements per thread value to the value of the iterator. If, after the adjustment, we are now below the max threads per thread block, return.

1.3.1.3 Tiles Per SM

Each SM can possibly run multiple thread blocks/tiles. As such, we compute the number of tiles that will run on a single SM as: $\frac{max\_threads\_sm}{threads\_per\_tile}$

1.3.1.4 Number of Rounds and Tiles Per Round

There is a finite number of SM's and an arbitrary number of tiles/thread blocks. Thus, it is possible that not all tiles can be executed in parallel. We must compute the number of rounds of parallel tile computation as part of the overall latency. Furthermore, it is useful to know how many tiles are being computed in parallel in each round.

The number of rounds is determined by: $\frac{num\_tiles}{tiles\_sm*num\_sms} + 1$

The tiles per round computation is straightforward from there. Note that we only add 1 if the numerator above is not evenly divisible by the denominator so that we are rounding up.

1.3.1.5 Tile Overlap

A key performance consideration is the need to synchronize writes to the output vector across tiles which have their own local memories that are being updated in parallel. We assume that the scheduler will do its best to map tiles (thread blocks) to SM's so as to minimize the need to synchronize. This can be accomplished by picking tiles that work on disjoint subsets of the output to run in a given round. However, it is not always possible to eliminate the need to synchronize altogether. As such, we compute the minimum amount of overlap of tiles to output vector subsets in each round as: $\frac{tiles\_round}{\frac{n_n}{t_n}}$

1.3.1.6 Alternative Mapping Strategies

As stated above, we made the assumption to have a 1:1 mapping of tiles to thread blocks. However, this assumption is only reasonable within some range of tile dimensions. Too small would generate considerably many thread blocks which is likely not optimal from a performance standpoint (as there is overhead associated with each thread block). Too big would result in more complex programs where threads must work on many elements, which forces more sequential execution. In recognition of these limitations, we would like to devote some discussion to alternate mapping strategies.

One alternative is to fix the total number of elements processed in the computation per thread block - independent of tiles. This allows for flexibility in the cases where fixing the mapping of tiles to thread blocks does not (i.e., the smaller and larger tile sizes). This is especially nice given that the dimensions of the tiles are user-defined parameters. However, this mapping strategy is more complex because tiles potentially need to be chunked together or broken out. If users can be trusted to run the kernel with reasonably performant tile dimensions, such flexibility might not actually be worth the implementation cost.

Therefore, if we have a tile size that is smaller than $32 * 32$, less than 1024 threads will be used per block, leaving the thread block underused. One solution is to have a block process multiple smaller tiles, but this come at the cost of having the block's register and scratchpad being shared for multiple tiles. This resource contention can slowdown data accesses, and therefore the overall execution in a manner similar to cache pollution. On the other hand, if we have a tile size greater than $32*32$, our thread block will require more than 1024 threads, and will fail to start due to being above the thread limit, or have to process the tile in multiple rounds. One solution is to spread the work between multiple blocks but this come at the disadvantage of losing the register and scratchpad sharing which is per block.

Another parameter at our disposal is the `elements_per_thread` which is the number of tile elements that each thread processes, and therefore the number of times that each thread runs the kernels. Hence, for a fixed tile size, `elements_per_thread` is the number of tiles that are processed and is therefore proportional to the amount of word done by the system. For a fair comparison, one needs to adjust both the tile size and `elements_per_thread` so that the total amount of work done is the same.

1.3.2 Stage 2: Execution Modeling

In this stage, we use the parameters computed in stage 1 to model the actual execution on GPU hardware. At the highest level, we loop over the number of sequential rounds. In each round (loop iteration), we compute the number of cycles of execution of the SM with the maximum number of tiles (thread blocks) mapped to it. As all tiles in the given round are run in parallel, this maximum suffices for the round's execution. This number of cycles is added to a global counter. At the completion of the loop, the global number of execution cycles is converted to an execution time using the GPU clock (which is the basis for the cycle count). The subsections to follow detail specifically how the number of cycles to compute a single tile is determined.

The methodology for determining tile computation latency depends on the underlying assumptions of the analysis - namely, the analysis type. As such, we have unique methods that map onto the unique types. These methods are called: tileop1, tileop2, tileop3, tileop4.

1.3.2.1 tileop1: REGISTER Analysis

Under the assumptions of the REGISTER type, all data for tile computation exists in the registers of the SM (of which there are infinitely many). As such, we must only compute the latency associated with dot products and accumulations (i.e., no loads and stores).

To do this, we treat the computation as follows: (1) The input and associated weights vectors are broken out into ∼SIMD vectors based on the warp size. (2) Perform a SIMD vector-vector multiply for each input chunk with all associated weight chunks. (3) Perform an additive reduction for each result from (1) and (2). (4) Add each reduced result from (3) to its appropriate output vector chunk.

Given the assumed CPI of 1 for all instructions, we have that 7 cycles are required per input-weight warp. The derivation: 1 cycle (vector-vector multiply), 5 cycles (reduction), 1 cycle (add to output).

1.3.2.2 tileop2: SCRATCHPAD Analysis

Under the assumptions of the SCRATCHPAD type, all data for tile computation exists in local L1 cache/scratchpad memory of the SM (which has infinite capacity). As such, relative to tileop1, we must include the latency associated with loads and stores to/from the L1/scratchpad.

To do this, we must: (1) load the input vector subset for the given tile, which is broken out into multiple distinct loads based on warp size. (2) load the weights matrix subset for the given tile, which is similar to the input load but performed for each row. (3) store the results to the output vector subset, which is also broken out into multiple distinct stores based on warp size.

1.3.2.3 tileop3: L2 Analysis

Under the assumptions of the L2 type, all data for tile computation exists in the global L2 cache initially (which is assumed to have infinite capacity). As such, relative to tileop2, we must introduce the extra latency associated with loads and stores that miss in the local L1 caches/scratchpads. Furthermore, tiles in different SM's could be writing the same outputs in the global L2 cache, meaning that write synchronization must occur for correctness.

To account for the extra latency associated with moving data to/from the L2 cache, we first compute the total amount of data that must be moved per tile based on the number of loads and stores (see tileop2). We take this amount in tandem with L2 bandwidth and the amount of concurrency (in the form of other SM's also trying to access the L2) to compute the overall extra latency. However, one of the major performance benefits of this architecture is the ability to hide the extra latency associated with memory accesses by running threads that are ready and strategically scheduling accesses to minimize downtime. As such, we take the overall latency penalty and reduce it by a "hide" factor. The "hide" factor is computed by taking `max_lat_hide` and reducing it by the parallelism factor of the workload. The parallelism factor is computed as the ratio of threads per SM to `max_threads_sm`. The closer this is to 1, the closer we are to achieving the maximum latency hide.

To account for the synchronization penalties, we use the tile overlap parameter defined earlier. If there is overlap, synchronization must occur. The more overlap there is, the more thread blocks must be synchronized and thus the greater the penalty. We use `sync_penalty` as the basis and scale linearly based on the amount of overlap in the given round to compute the overall penalty.

1.3.2.4 tileop4: MEMORY Analysis

The MEMORY analysis type imposes no unrealistic constraints/assumptions - data is in the GPU's main memory at the start of the computation. As such, relative to tileop3, we must introduce the extra latency associated with loads and stores that miss in the L2 cache. This is in addition to potential misses at the L1/scratchpad level that hit in L2 as well as the synchronization problem.

To simulate an L2 cache miss, we develop a model of a cache defined by the parameter p which is the portion of data requested that is currently in the cache. This parameter is estimated with the following equation

$$p = \frac{\texttt{l2\_cap}}{\texttt{num\_access} \times \texttt{warp\_size} \times \texttt{val\_size}} \tag{1}$$

where `num_access` is the number of loads and stores normalized by the `cpi`. By multiplying by `warp_size` and `val_size` we can compute the number of bytes requested.

We can see that if the `l2_cap` is greater than or equal to number of bytes requested, $p \geq 1$ so we assume that all the data exists within the L2 cache and there is no need for memory accesses. In the case the $p < 1$ then $p\%$ of the data can be accessed with a cost of `l2_latency` and $(1-p)\%$ with a cost of $\texttt{l2\_latency} + \texttt{mem\_latency}$ since the data must be fetched from memory and put into the cache. Therefore we define our access latency to be

$$\texttt{access\_latency} = p \times \texttt{l2\_latency} + (1-p) \times (\texttt{l2\_latency} + \texttt{mem\_latency}) \tag{2}$$

1.4 Challenges

In implementing our model, we faced a number of interesting challenges - many of which will require further work/research.

1.4.1 Memory Latency Modeling

In modeling memory latency, we had some trouble coming up with an accurate estimate of the memory access times. Several factors for an accurate estimate were unknownable so we resorted to using accepted values reported in literature. The values used can be seen in Cook (2012). There were other factors such as bus contention that we were unable to model. Ignoring bus contention allowed us to make simplifying assumptions about estimating the effects of thread execution on concurrency and its ability to hide latency. An possible next step would be to do further research into accepted estimates of the cost of contention and come up with a pipelined based latency hide. We also acknowledge that our probabilistic modeling of hits and misses does not account for the fact that all accesses are initially misses (i.e., there is a warm-up period). In future work, we plan to add a temporal aspect to modeling of hits and misses.

1.4.2 Cache Control Modeling

The policies for cache control - specifically placement - were difficult to determine at various levels of the hierarchy and can have an impact on our analysis. We generally took the simplest approach - namely, all caches are fully-associative - because this was the easiest to model. However, in reality, for large caches like L2, there is more likely some lower set associativity. This would result in more misses due to conflicts, which would increase the number of accesses and corresponding access latency. In future work, we will confirm the specific and relevant cache parameters to bake into the model.

1.4.3 Synchronization Modeling

The modeling of synchronization latency was challenging for a couple of reasons. First, it is hard to say how "good" the GPU's mapper of thread blocks to SM's is in regard to minimizing synchronization overhead. We make a pretty optimistic assumption about how thread blocks are mapped, but the reality could be different for a variety of reasons. (1) the GPU hardware does not assume responsibility for mapping at all and does so arbitrarily. In this case - if the software must synchronize - it must do so under the worst case assumption of overlap. (2) the GPU hardware tries to optimize, but doesn't have perfect information and/or has other priorities. It seems like at least one of these explanations is likely (at least to some degree), which has important implications. Second, determining the penalty for synchronization when it must occurs was difficult. Documentation on the subject is limited and this penalty is a function of the degree of overlap, but it is unclear whether that function is linear or not (we treat it as linear). Our hope is that the various assumptions we've made "cancel out" in the sense that the end result is close to reality despite the possibility of any one assumption being far off.

1.4.4 Computational Modeling

Though the computation itself is relatively straightforward (dot product then accumulate), a key assumption that underlies our model is the CPI associated with the instructions we hypothesize to execute the computation. It remains to be seen whether or not a CPI of 1 for FP adds, multiplies, loads, and stores is reasonable in the TitanV architecture. We made this assumption as it is a "better" case scenario and can help mitigate the effect of underestimating the instructions involved, but this could easily have resulted in an underestimation of latency along this dimension of the analysis - which is hugely important. We did make sure to assign this as a parameter that can be changed as we do more research to validate the CPI assumption. Furthermore, it might be worth breaking out from a uniform CPI to a distribution across instruction types.

1.5 Future work

Much of our future work initiatives have already been outlined throughout the report thus far. However, below is a more concise summary, with a few additions:

- More robust modeling of the relationship between local L1 cache and scratchpad. These are treated as one entity but presumably have different properties and advantages.
- Modeling value types other than `fp16`.
- Modeling CPI at a more granular level. This involves both determining true CPI's for various instruction types and enabling non-uniformity across them.
- Modeling memory latency more accurately at various levels of the memory hierarchy. Specifically, honing in on correctly quantifying the amount of latency that can be hidden by parallelism.

- Modeling synchronization across thread blocks more accurately. This involves several pieces: (1) accurately computing how much synchronization is actually necessary. (2) accurately determining the penalty associated with synchronization, potentially distinguishing between thread blocks in the same versus different SM's.
- Modeling the overhead of cache management - more specifically placement and replacement - more accurately. In addition, modeling the temporal behavior of cache hits and misses.
- Adding support for different kernels into the analysis tool. A nice next step would be to enable the modeling and analysis of convolutions.

2 Validation

For fixed dimensions of the input and output vectors to the classifier, we microbenchmarked our model's performance for different tile sizes. It validated that our model's performance depends significantly on the assumption that CUDNN chooses an optimal or close to optimal tiling dimensions to achieve good performance. As the tile area increases, the execution time decreases. We picked 32x32 as our default tiling dimensions.

tile height	tile width	model time (usec)	tile height	tile width	model time (usec)
8	8	1397	16	16	178
8	12	616	16	24	76
8	16	352	16	32	41
8	24	149	16	64	13
8	32	79	24	24	48
8	64	22	24	32	23
12	12	412	24	64	18
12	16	222	32	32	22
12	24	105	32	64	17
12	32	51	64	64	21
12	64	11			

We varied the sizes of the input and output vectors to the classifier, and compared the execution times computed by our model to the measured execution times of CUDNN's gemm problems on the Titan V. The input vectors ranged from 4096 to 131072 elements; the output vectors ranged from 512 to 4096 elements. Regardless of the input vector size, the performance of the model improves as the output vector size grows closer to 4096. The dimensions (Ni=8192, Nn=4096), (Ni=16384, Nn=4096), (Ni=25088, Nn=4096), (Ni=131072, Nn=4096) achieve the closest execution time estimates to the measured execution times: 0.03, 0.10, 0.11, and 0.10 respectively. We believe that this is a result of the combination of our assumptions (including the 32x32 default tiling dimensions and minimal tiling overlap) optimizing for the output vector size 4096.

m	k	measured time (usec)	model time (usec)	model error
4096	512	21	86	0.76
4096	1024	36	84	0.57
4096	2048	69	96	0.28
8192	512	31	173	0.82
8192	1024	57	168	0.66
8192	2048	111	191	0.42
8192	4096	233	240	0.03
16384	512	58	346	0.83
16384	1024	109	403	0.73
16384	2048	213	380	0.44
16384	4096	432	480	0.10
25088	512	89	526	0.83
25088	1024	163	615	0.73
25088	2048	322	583	0.45
25088	4096	735	735	0.00
65536	512	224	1508	0.85
65536	1024	424	1607	0.74
65536	2048	839	1523	0.45
65536	4096	1711	1919	0.11
131072	512	457	3016	0.85
131072	1024	855	3214	0.73
131072	2048	1718	3045	0.44
131072	4096	3459	3837	0.10

3 Architecture Insight

3.1 Effects of Tiling on Performance

As can be seen in our validation results, differing tile dimensions on fixed input and output vector sizes can have a major impact on performance. Given that, with such fixed dimensions, the nominal cycles devoted to computation and to memory accesses is expected to be relatively constant, there must be architectural overhead that comes about specifically because of the tile structure. We have gleaned a couple of insights from this observation - one that is relatively obvious and one that is more nuanced.

The most apparent architectural insight is, under the assumption of 1 tile to 1 thread block, smaller tile dimensions imply a larger number of tiles. In theory, this shouldn't be problematic because the same number of underlying threads are required - this is how our model is structured. However, there must be some limit on the number of distinct thread blocks that can run on an SM that is independent of the maximum number of threads for that same SM. If this were the case, actual execution times would be inflated relative to our model's predictions because not as much parallelism can be exploited. This is indeed what we see in our validation results.

The more subtle insight stems from synchronization across thread blocks/tiles. Our model computes the minimal tile overlap such that the need to synchronize is as small as possible (i.e., across a minimal number of thread blocks). Furthermore, our model assumes that there is a uniform synchronization penalty regardless of the locations of the thread blocks to synchronize (different SM's, same SM). Our understanding based on the validation results is, relative to reality, these assumptions likely cancel each other. We are probably too optimistic about how thread blocks are mapped to SM's in the context of any one kernel. On the flip side, we are probably too pessimistic about the synchronization penalty for thread blocks that must be synchronized but exist in the same SM. Regardless, our modeling efforts have brought light to this notion which can have a meaningful impact on execution time.

3.2 Cost of Reduction Operations

A key step in the actual computation of the classifier kernel is the additive reduction step in the dot product of the input vector and weights vector subsets. Our model uses $5 * \texttt{CPI}$ cycles per warp for the reduction operation (which might consist of multiple warps). Thus, it is by far the biggest contributor to computational latency. This points to limitations in the microarchitecture of the SM. If there was hardware to perform an additive reduction of a vector, as one might find in a true SIMD machine, this step in the computation would presumably be much less expensive. However, given that the SM is only ∼SIMD, our understanding is that such operations aren't explicitly supported in hardware - resulting in the overhead we observe.

References

[Cook 2012] Cook, Shane: *CUDA Programming, A Developer's Guide to Parallel Computing with GPUs*. 2012. – URL `http://www.hds.bme.hu/~fhegedus/C++/Shane%20Cook%20-%20CUDA%20Programming%20-olvasOM.pdf`