CUDA Implementation and Results of Simple Deep Leanring Kernels

Josh Kimmel

UCLA — CS 259 — April 2021

1 Introduction

In this mini-project, we were tasked with implementing two general deep learning kernels on provided GPU hardware using the CUDA programming framework. These kernels are: a fully-connected classifier layer (i.e., matrix-vector multiply) and a 3D convolution kernel. It is assumed that the reader is familiar with these kernels both in their mathematical properties/structures as well as their applications to deep learning and artificial neural networks.

2 High-Level Parameters and Constraints

The subsections to follow describe the software and hardware constraints of the implementation. The code for the project can be found on GitHub at:
`https://github.com/joshkimmel16/cuda-kernels`

2.1 Hardware

A description of the hardware can be seen in the figure below. Some parameters of note are:

- 80 SM's, 64 SP's per SM.
- Maximum of 1024 threads (32 warps) per thread block. Maximum of 2048 threads per SM.
- 4.7 MBs of L2 cache.
- 49,152 bytes of shared memory per thread block.

```
Device 0: "TITAN V"
  CUDA Driver Version / Runtime Version          10.1 / 10.1
  CUDA Capability Major/Minor version number:    7.0
  Total amount of global memory:                 12037 MBytes (12621381632 bytes)
  (80) Multiprocessors, ( 64) CUDA Cores/MP:     5120 CUDA Cores
  GPU Max Clock rate:                            1455 MHz (1.46 GHz)
  Memory Clock rate:                             850 Mhz
  Memory Bus Width:                              3072-bit
  L2 Cache Size:                                 4718592 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 7 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 175 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
```

2.2 Software

As part of the project, 4 different kernels were implemented, each with specific parameters. Each is detailed below.

2.2.1 Classifier 1

The classifier is characterized by two parameters: the length of the input vector n_i and the length of the output vector n_n. For classifier 1 ("class1"), these parameters are:
$n_i = 25088$
$n_n = 4096$

2.2.2 Classifier 2

The classifier is characterized by two parameters: the length of the input vector n_i and the length of the output vector n_n. For classifier 2 ("class2"), these parameters are:
$n_i = 4096$
$n_n = 1024$

2.2.3 Convolution 1

The convolution is characterized by six parameters: the dimensions of the feature maps n_x, n_y, the dimensions of the filter k_x, k_y, the number of input feature maps n_i, and the number of output feature maps n_n. For convolution 1 ("conv1"), these parameters are:
$n_x = 224$
$n_y = 224$
$k_x = 3$

$k_y = 3$
$n_i = 64$
$n_n = 64$

2.2.4 Convolution 2

The convolution is characterized by six parameters: the dimensions of the feature maps n_x, n_y, the dimensions of the filter k_x, k_y, the number of input feature maps n_i, and the number of output feature maps n_n. For convolution 2 ("conv2"), these parameters are:
$n_x = 14$
$n_y = 14$
$k_x = 3$
$k_y = 3$
$n_i = 512$
$n_n = 512$

3 Implementation and Parallelization Strategies

In the sections to follow, the general strategies for implementing each kernel will be described, as well as the various ways in which parallelism was leveraged.

3.1 Class1

All data structures (inputs, outputs, weights) are represented as uni-dimensional arrays. This is straightforward for both the inputs and outputs. For the weights, which are logically a two-dimensional structure, it is assumed that the data is laid out in row-major order.

As part of the implementation, I have defined tiling parameters t_x and t_y. The purpose of these parameters is to chunk up the matrix-vector multiplication between the input vector and weights matrix into independent partial sums across distinct submatrices and subvectors of the weights and inputs respectively. For this implementation, I chose the tiling parameters as follows:
$t_x = 32$
$t_y = 32$

Each thread block is associated with the computation of a single tile. Each thread within the thread block handles a single pair of elements in that computation. Thus, the following performance characteristics hold:
$\#ofthreadblocks = (n_i/t_x) * (n_n/t_y) = (25088/32) * (4096/32) = 784 * 128 = 100352$
$\#ofthreads/threadblock = t_x * t_y = 32 * 32 = 1024$

The tiling parameters were chosen to maximize the number of threads per thread block (at 1024). Within each thread block, the relevant subset of the input vector can be (and is) reused across all threads for better memory performance.

A limitation on performance for this strategy is the very high number of thread blocks. Given that the number of threads per thread block is being maxed out, we have that at most two thread blocks can be assigned to a given SM. This means that only 160 thread blocks can be executing in parallel, which imposes a large degree of sequential processing. Another limitation is that the total required amount of memory to execute the computation far exceeds the size of the L2 cache on the GPU. Thus, without clever exploitation of locality, one might expect to see more cache misses and lower effective memory bandwidth.

3.2 Class2

All data structures (inputs, outputs, weights) are represented as uni-dimensional arrays. This is straightforward for both the inputs and outputs. For the weights, which are logically a two-dimensional structure, it is assumed that the data is laid out in row-major order.

Each thread block is associated with the computation of 1 input vector element across all outputs. Each thread within the thread block handles a single weight from the corresponding weights vector in that computation. Thus, the following performance characteristics hold:
$\#ofthreadblocks = n_i = 512$
$\#ofthreads/threadblock = n_n = 512$

Within each thread block, the relevant input vector element can be (and is) reused across all threads for better memory performance.

A limitation on performance for this strategy is again the number of thread blocks. We have that at most four thread blocks can be assigned to a given SM given the number of threads per block. This means that only 320 thread blocks can execute in parallel, which imposes some sequential processing.

3.3 Conv1

All data structures (inputs, outputs, weights) are represented as uni-dimensional arrays. For inputs and outputs, which are logically three-dimensional structures, it is assumed that each feature map occurs sequentially. Within an output feature map, output data is laid out in row-major order. Within an input feature map, the data is assumed to be chunked into submatrices based on a pre-defined "batch" parameter. Each submatrix is assumed to be laid out in row-major order. For the weights, which are logically a four-dimensional structure, it is assumed that the highest order dimension is the number of input feature maps, then the number of output features maps. Within a single weights matrix, the data is laid out in row-major order.

As mentioned earlier, I have defined a batch parameter. The purpose of this parameter is to chunk up input feature maps into sub-computations that can be handled by a single thread block. Partial sums are computed and accumulated on the output side. For this implementation, I chose the batch parameter as follows:
$batch = 64$

Each thread block is associated with the computation of 1 chunk of an input feature map with n_n weights matrices. Each thread within the thread block handles an application of each weights matrix (matrix-vector multiply) to a unique subset of the input feature map chunk. Each element in the input feature map chunk is processed as such, by applying the weights matrix centered at that element. Thus, the following performance characteristics hold:
$\#ofthreadblocks = n_i * batch = 64 * 64 = 4096$
$\#ofthreads/threadblock = n_x * n_y/batch = 224 * 224/64 = 784$

Within each thread block, the given input feature map chunk and all weights matrices are reused across all threads for better memory performance.

A limitation on performance for this strategy is again the number of thread blocks. We have that at most two thread blocks can be assigned to a given SM given the number of threads per block. This means that only 160 thread blocks can execute in parallel, which imposes some sequential processing.

3.4 Conv2

All data structures (inputs, outputs, weights) are represented as uni-dimensional arrays. For inputs and outputs, which are logically three-dimensional structures, it is assumed that each feature map occurs sequentially and, within a feature map, the data is laid out in row-major order. For the weights, which are logically a four-dimensional structure, it is assumed that the highest order dimension is the number of input feature maps, then the number of output features maps. Within a single weights matrix, the data is laid out in row-major order.

Each thread block is associated with the computation of 1 input feature map with 1 weights matrix. Each thread within the thread block handles an application of the weights matrix (matrix-vector multiply) to a unique subset of the input feature map. Each element in the input feature map is processed as such, by applying the weights matrix centered at that element. Thus, the following performance characteristics hold:
$\#ofthreadblocks = n_i * n_n = 512 * 512 = 262144$
$\#ofthreads/threadblock = n_x * n_y = 14 * 14 = 196$

Within each thread block, the given input feature map and weights matrix are reused across all threads for better memory performance.

A limitation on performance for this strategy is again the number of thread blocks. We have that at most ten thread blocks can be assigned to a given SM given the number of threads per block. This means that only 800 thread blocks can execute in parallel, which imposes some sequential processing.

4 Performance Analysis

4.1 Class1

The kernel execution time was captured using nvprof. It was reported to be: 9.76ms.

It seems that, from the perspective of nvprof, this kernel was distributed into a single "batch" and thus throughput was not reported (at least this is my hypothesis as to why I could not obtain an operational throughput metric). I decided, given the degree of parallelism that the GPU supports, my primary objective would be to optimize execution time as opposed to working under the constraint of a fixed batch size. Admittedly, I'm somewhat unsure of how my own tiling strategy discussed earlier maps onto the notion of batching as defined in the project spec.

Using the dram_read_throughput option of nvprof, we have that average read throughput from the device's main memory to the L2 cache is: 20.44 MB/s. Such a metric implies that this kernel is memory-bound. This is to be expected given the implementation described earlier, as the total memory required exceeds the size of the L2 cache, meaning that we expect to see capacity misses in the L2 cache. In addition, there is a non-uniform access pattern across weights within a thread-/thread block, which would exacerbate misses. Furthermore, there is rather poor reuse of data overall, meaning that scratchpad is unlikely to be a limiting factor. Compute is an interesting performance dimension to analyze. On one hand, given the simplicity of each thread's compute task, we are clearly not limited by the execution time of individual threads. However, the large number of thread blocks forces a considerable degree of sequential processing ($100352/160 = 627$ sequential thread block executions). This undoubtedly is a limiting factor on performance.

4.2 Class2

The kernel execution time was captured using nvprof. It was reported to be: 1.08ms.

It seems that, from the perspective of nvprof, this kernel was distributed into a single "batch" and thus throughput was not reported (at least this is my hypothesis as to why I could not obtain an operational throughput metric). I decided, given the degree of parallelism that the GPU supports, my primary objective would be to optimize execution time as opposed to working under the constraint of a fixed batch size. It seems as though my implementation strategy in this case has a batch size of 1, but I am still somewhat unsure of how it maps onto the notion of batching defined in the project spec.

Using the dram_read_throughput option of nvprof, we have that average read throughput from the device's main memory to the L2 cache is: 14.17 GB/s. Such a metric implies that this kernel is not memory-bound. This makes sense given that all data can fit in the L2 cache easily. Note that there is still rather poor reuse of data overall, meaning that scratchpad is also unlikely to be a limiting factor. Thus, compute is presumably the limiting performance factor. Though, again, individual thread computations are simple, there are still too many thread blocks to be handled by a single sequence ($512/360 = 2$ sequential thread block executions).

4.3 Conv1

The kernel execution time was captured using nvprof. It was reported to be: 2.09ms.

It seems that, from the perspective of nvprof, this kernel was distributed into a single "batch" and thus throughput was not reported (at least this is my hypothesis as to why I could not obtain an operational throughput metric). I decided, given the degree of parallelism that the GPU supports, my primary objective would be to optimize execution time as opposed to working under the constraint of a fixed batch size. Admittedly, I'm somewhat unsure of how my own batching strategy discussed earlier maps onto the notion of batching as defined in the project spec.

Using the dram_read_throughput option of nvprof, we have that average read throughput from the device's main memory to the L2 cache is: 5.82 GB/s. Such a metric implies that this kernel is somewhat memory-bound (for lack of a better way to put it). This makes sense as, despite the fact that there is decently good data reuse via scratchpad memory, the batching strategy results in certain irregular access patterns that could be problematic for caching. In addition, scratchpad was somewhat of a limiting factor because the batching strategy was only adopted to deal with the fact that entire input feature maps do not fit in the scratchpad. Finally, compute is also presumably a limiting performance factor, due to the sheer number of thread blocks ($4096/160 = 26$ sequential thread block executions). Threads within the thread block are noticeably more complex than the classifiers, but this does not seem to make much of a difference in performance. This phenomenon is an interesting avenue of exploration for performance improvements.

4.4 Conv2

The kernel execution time was captured using nvprof. It was reported to be: 685.15us.

It seems that, from the perspective of nvprof, this kernel was distributed into a single "batch" and thus throughput was not reported (at least this is my hypothesis as to why I could not obtain an operational throughput metric). I decided, given the degree of parallelism that the GPU supports, my primary objective would be to optimize execution time as opposed to working under the constraint of a fixed batch size. It seems as though my implementation strategy in this case has a batch size of 1,

but I am still somewhat unsure of how it maps onto the notion of batching defined in the project spec.

Using the dram_read_throughput option of nvprof, we have that average read throughput from the device's main memory to the L2 cache is: 14.62 GB/s. Such a metric implies that this kernel is not memory-bound. This makes sense as there is good data reuse via scratchpad memory and entirely regular (contiguous) data access patterns. Given that entire input feature maps and weights matrices could fit into scratchpad memory, this was clearly not a limiting factor. Thus, compute is presumably the limiting performance factor, again due to the sheer number of thread blocks ($262144/800 = 328$ sequential thread block executions). Threads within the thread block are noticeably more complex than the classifiers, but this does not seem to make much of a difference in performance. This phenomenon is an interesting avenue of exploration for performance improvements.

5 Comparison to CUDNN

There are a couple of important notes to make here before diving into the comparison. First, I have implemented these kernels using 4-byte integers, whereas the reported results from CUDNN (`https://docs.google.com/spreadsheets/d/1LRDl_3oUGBdZlpaJv6JSguBBw9Mj6ut5QuQojfrapbs/edit#gid=0`) only capture results using floats. My choice to use integers was mainly for ease of debug and didn't seem to matter for the analysis. Second, it is somewhat unclear to me which "batch" results to use for comparison, but I will elaborate on why I chose the particular results that I did in each subsection.

5.1 Class1

For this comparison, I chose to use the results with a batch size of 32. This is based on the tiling parameters that I chose to implement, whereby each input sub-array was of length 32.
The glaring metric to note is the difference in execution time between my implementation and that of CUDNN. The speedup can be characterized as follows:
$\frac{9760us}{774us} = 12.6$
This is a considerably high speed-up. My hypothesis as to the root cause is the low effective bandwidth that my implementation achieves. A possible explanation (and avenue for improvement) is that the access patterns in my implementation are such that the number of L2 cache misses is considerably higher. As such, I should re-evaluate the data layout and processing order to better understand how I can optimize memory throughput.

5.2 Class2

For this comparison, I chose to use the results with a batch size of 1. This is based on the fact that each thread handles the processing of a single element in the input vector.
The glaring metric to note is the difference in execution time between my implementation and that of CUDNN. The speedup can be characterized as follows:
$\frac{1080us}{40us} = 27$
This is another considerably high speed-up. My hypothesis as to the root cause is the compute efficiency - namely with respect to outputs. Data reuse is only available at the input and output vector dimensions. Though I am reusing input vector elements, there is no reuse across output vector elements. Presumably, these store operations thus have higher latencies associated with them (though no bandwidth penalty per se), leading to a higher execution time. I must further explore ways in which to optimize the output capture.

5.3 Conv1

For this comparison, I chose to use the results with a batch size of 64. This is based on my "batching" mechanism, which implicitly breaks each input feature map into 64 submatrices, meaning that each submatrix contains $\frac{1}{64}$ of the data.
The glaring metric to note is the difference in execution time between my implementation and that of CUDNN. The speedup can be characterized as follows:
$\frac{2090us}{8341us} = 0.25$
This is a very surprising result (if I'm understanding the reported results correctly) as it implies that my implementation is about 6x faster than CUDNN. I find this hard to believe as CUDNN has been optimized far more than my simple implementation. This leads to me to three possible conclusions: (1) the sheer volume of FP16 computations (CUDNN) versus int4 computations (me) and the resulting latency differences account for the discrepancy (2) my assessment of which batch results to compare to is incorrect or (3) my implementation has a bug which renders it computationally simpler than it should be. A next step would be to explore each of these avenues to understand the results discrepancy.

5.4 Conv2

For this comparison, I chose to use the results with a batch size of 1. This is based on the fact that a single thread handles a single element of a given input feature map.
The glaring metric to note is the difference in execution time between my implementation and that of CUDNN. The speedup can be characterized as follows:
$\frac{685us}{323us} = 2.1$
This result stands to reasonable belief. My implementation is taking advantage of some of the data reuse available (i.e., on the input and weights side) but none on the output side. My hypothesis is that the CUDNN speedup results from exploiting data reuse of output feature maps. This is an avenue for further exploration and optimization.

6 Discussion of Optimization and Conclusion

In terms of the optimizations that I used, the one that was trivially the most useful was of course the ability to exploit considerable parallelism in the form of threads and thread blocks on the GPU hardware. My initial thinking was to try and chunk computations into thread blocks that maximized the number of threads within (at 1024). I also tried to ensure that individual threads within a thread block were performing as simple (i.e., as few instructions) of a computation as possible. In hindsight, it is possible that both of those strategies could have backfired. For my kernels that are compute-bound (most of them), my suspicion as to the performance degradation is the fact that there are a large number of thread blocks. With large thread counts per thread block, this severely limits the number of thread blocks that can be executing in parallel on the fixed number of SM's, thereby imposing a noticeable degree of sequential execution. By having more complex threads, I could have considerably lowered the number of thread blocks. By using less threads per thread block, more thread blocks can be assigned to the same SM. It remains to be seen if there is a more optimal distribution of compute, but the CUDNN results certainly suggest that there is.

The other optimization that I tried to take advantage of was scratchpad memory and data reuse. In the classifier kernels, the effectiveness of this strategy was limited, given that there is inherently less data reuse available as I have structured the computation (only the inputs and outputs, not the weights). For the convolution kernels, I seemed to get decently good performance using this strategy, given that a weights matrix and at least partial feature maps could be reused across very many instructions. One limiting factor in general in my use of this strategy was that I did not implement

any data reuse of outputs. I couldn't quite conceptualize how this would look and whether or not it would be useful in the initial iterations of my implementation and did not get around to refining my implementation by adding this later on. However, I think that all kernels would have benefited notably and in certain cases I had scratchpad capacity to spare.

Next, I would like to discuss tensor cores and, more importantly, the fact that I did not use them. One reason for this is that I was not familiar enough with the syntax to do so effectively. But the major reason was that I didn't mathematically map my computation strategy onto matrix algebra. I think that doing so would have improved certain kernels - especially conv1 and class1 where I had to chunk out the computation into smaller data structures anyways.

Finally, I would like to address the notion of sparsity. I did not have time to implement any clever exploitation of sparse computation, though I imagine this can benefit the overall compute greatly. On the flip side, my understanding is that exploitation of sparsity typically arises from pre-existing knowledge of the underlying structure of the inputs/weights, which clearly cannot be the case for these kernels as all inputs and weights are generated randomly. Furthermore, I imagine mapping such sparsity exploitation onto the parallel programming paradigm would be difficult, but it is an optimization tactic that I'm interested in thinking about more as a next step.