Speeding up Capsules

Mithilesh Vaidya*
Georgia Institute of Technology
CS 6245 Course Project

Abstract

Capsules are a promising replacement for convolutions in deep learning. They are inspired by computer graphics: instead of simply learning visual features using convolutional filters, we learn the feature and capture it's orientation with respect to it's parent. However, it's implementations failed to scale up to large datasets. Part of the reason for this failure was the lack of a tuned kernel for CUDA GPUs, which are the standard compute devices for training neural networks. In this work, we implement the core Capsule operator from scratch in CUDA and borrow various code transformations from standard Compilers theory. We observe a net speedup of about 5x over a naive implementation and explain this performance improvement by examining the output of the profiler. Our observations highlight the failure of nvcc to carry out well-known code transformations in order to maximise cache and register usage. To conclude, we discuss a number of future extensions which can potentially make capsules a viable alternative to convolutions.

1 Introduction

Capsules [SFH17] were pioneered by Geoffrey Hinton in 2017 as a replacement for convolutions in deep learning. The main idea behind capsules is to capture the orientation of a feature with respect to it's parent. This is in contrast to convolutions, which simply learns different features for different orientations. Capsules allows the model to generalise better to unseen data as it does not need to learn the same feature in multiple orientations. This is especially useful for tasks such as object detection, where the same feature can appear in multiple orientations, especially those unseen in the training data. We can think of it as inverse graphics, in which we are learning network parameters which can invert the hierarchical matrix multiplcations done during rendering in computer graphics.

However, the idea could not be scaled up to large datasets. One of the major factors for this failure was the lack of a tuned kernel for accelerators (such as NVIDIA GPUs) which happen to be the standard devices for efficient training of deep neural networks. This issue was the focus of an aptly titled paper called *Machine Learning Systems are Stuck in a Rut* [BI19], which criticized the current systems paradigm of hyper-optimizing large known kernels at the expense of flexibility. More specifically, programmers are focused on tuning a small number of known kernels (such as convolution), which reduces expressiveness and modularity. Hence, the current systems paradigm is not well suited for the rapid prototyping and experimentation required for exploring new ideas in machine learning.

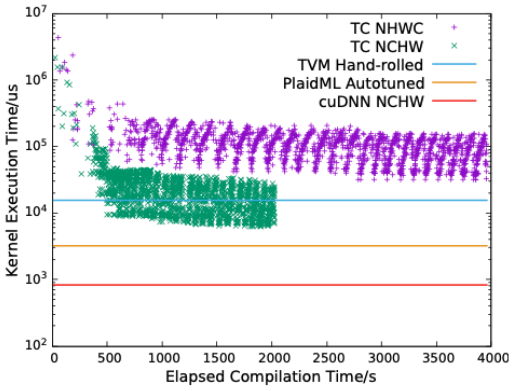
1.1 Difference between Convolutions and Capsules

As stated in [BI19], standard convolution can be formally written as:

$$O^{n,c_o}_{x,y} = \sum_{k_x} \sum_{k_y} \sum_{c_i} I^{n,c_i}_{x+k_x,y+k_y} \times K^{c_i,c_o}_{k_x,k_y} \quad \forall \quad n, c_o, x, y \tag{1}$$

Here, O, I, K are 4-dimensional arrays which represent the Output, Input and Kernel respectively. n is the batch index, c_o is the output channel index, c_i is the input channel index and x, y are the spatial indices. The summation is over the kernel sizes in both spatial dimensions and the input channel dimension. Note that the innermost operation is a scalar multiplcation.

*For any clarifications/code, you can reach me at mithilesh.vaidya@gatech.edu



(a) Compilation and execution time for **standard convolution** using various frameworks.

Compiler	Device	Compilation	Execution
gcc	x86 (1 core)	500ms	64.3ms
gcc -fopenmp	x86 (6 cores)	500ms	11.7ms
PlaidML	GTX1080	560ms	604ms
Tensor Comp.	GTX1080	3.2s	225ms
Tensor Comp.	GTX1080	64s	18.3ms
Tensor Comp.	GTX1080	1002s	1.8ms
CUDA	GTX1080	48h	1.9ms

(b) Compilation and execution time for **capsules** using various frameworks. Note the absurdly high compilation time for CUDA highlighted in red.

Figure 1: Performance of convolution and capsules using various frameworks. Both figures reproduced from [BI19].

The analogous operator for capsules is:

$$O^{n,c_o}_{x,y} = \sum_{k_x}\sum_{k_y}\sum_{c_i} I^{n,c_i}_{x+k_x,y+k_y} \cdot K^{c_i,c_o}_{k_x,k_y} \quad \forall \quad n, c_o, x, y \tag{2}$$

In the case of capsules, O, I, K are 6-dimensional arrays which represent the Output, Input and Kernel pose matrices respectively. The only difference between the two operators is the innermost operation: **scalar multiplication** in the case of convolutions and a **4x4 matrix product** in the case of capsules. This seemingly small change has a huge impact on the performance of the operator, as seen in the next section.

1.2 Performance difference

Before we discuss capsules, it is worth examining the performance of the standard convolution operator using various frameworks (all results from [BI19]). In each case, the core convolution operator was written using the lowest-level primitives offered by the framework. As we can see in figure 1(a), the performance of the operator varies widely across frameworks. The best performance is achieved by the cuDNN library, which is a highly optimized library for deep learning operations on NVIDIA GPUs. The general trend is that despite extensive compilation time, none of the frameworks discover a kernel which can match the execution time of cuDNN. Refer to section 2.3 in [BI19] for a more detailed discussion of the results.

We now discuss the corresponding results for capsules shown in figure 1(b). Firstly, note the large gap in the execution time of a parallelised x86 implementation and CUDA (from 11.7ms to 1.9ms). This highlights the importance of using accelerators for training deep neural networks. However, note the high compilation time for CUDA (about 2 days). The source of such high compilation time is not very clear.[1] We hypothesize that memory layout optimizations (ordering of the dimensions, splitting the data across blocks and threads, etc.) is the key culprit. This high compilation time is a major bottleneck for rapid prototyping and experimentation. It is important to note that the above compilation times are for a **single** kernel. In practice, a deep learning model will consist of multiple operators (such as convolution, fully connected layers, activations, etc.). Each will have their own kernels interacting in a very complex fashion, which would increase the compilation time even further.

[1] We contacted the authors but did not hear back from them. Moreover, their code is not available online.

2 Problem Statement

In this work, our main focus is on speeding up the core matrix multipliation operation in equation 2. A few things to keep in mind before we proceed:

- The focus is **not** on generality but instead on specifically speeding up the capsule operator using the techniques studied in class (and beyond).
- We focus on speeding up the nested loop; whether capsules eventually lead to an improvement over convolutions on standard datasets is not the focus of this work.
- It is important to note that convolution cannot be **easily** repurposed to perform the capsule operation. Refer to section 2.4 in [BI19] for more details.
- While [BI19] stresses on **poor compilation** time as the main bottleneck in trying out new ideas, we focus on speeding up the kernel itself. Finding the right parameters for the kernel given a GPU device is a problem in itself and is not the current focus of this work.

3 Approach

Let us examine the capsule operator more carefully:

$$O^{n,c_o}_{x,y} = \sum_{k_x} \sum_{k_y} \sum_{c_i} I^{n,c_i}_{x+k_x,y+k_y} \cdot K^{c_i,c_o}_{k_x,k_y} \quad \forall \quad n, c_o, x, y \tag{3}$$

We can make the following observations:

- There is no dependence across the dimensions n, x, y, c_o and hence we can parallelise the computation across all these dimensions.
- The 3 loops k_x, k_y, c_i can be interchanged since addition is associative and commutative.

Hence, there is no loop-carried dependence and we can parallelise the computation across all 7 loops (One each for $n, x, y, c_o, k_x, k_y, c_i$). In other words, the problem boils down to memory layout optimizations. We can explore transformations such as **loop tiling** for maximising cache usage and **unroll-and-jam** for maximising register usage.

We follow the same procedure as convolution for distributing the computation across the GPU:

- Each block of threads is assigned one row of the image and each thread within a block is responsible for computing a single output pixel across that row.
- Each thread is responsible for computing the innermost operation, which is a 4x4 matrix product inside a 3-nested loop.
- The filter coefficients are stored in the shared memory of the GPU, which is accessible to all threads within a block. This prevents reads from global memory.
- The input image is stored in the global memory of the GPU since it is too large to fit in the shared memory.

3.1 Assumptions

For the purpose of this work, we make the following assumptions:

- $n = 1$ i.e. we only consider a single image at a time. This is a reasonable assumption since the batch size is usually small and the computation is dominated by the core capsule operation. Moreover, we can scale it to multiple images by simply running more threads.
- $c_o = 1$ i.e. we only consider a single output channel at a time. This is a reasonable assumption because filters are stored in shared memory. It is better to compute the output for one corresponding output filter before swapping in the next set of filter coefficients.

3.2 Layout

We use the following layouts for the kernel and the input, output images:

- Kernel: $k_x, k_y, c_i, 4, 4$
- Input: $N_x, N_y, c_i, 4, 4$
- Output: $N_x, N_y, 4, 4$

Having the 4x4 matrix laid out contiguously in memory allows us to maximise cache hits.

In the future, it would be interesting to compare the layout and the assumptions stated above with the hand-optimized convolution implementation.

4 Code transformations

Our baseline implementation is a simple matrix multiplication loop inside the kernel. Note that the order of the loops is optimized so as to maximise cache hits. The code is shown below:

```
1     (loop over kx, ky, channel)
2         for (int i = 0; i < 4; i++){
3             for (int k = 0; k < 4; k++){
4                 for (int j = 0; j < 4; j++){
5                     out[idx*16 + i*4 + j] += img[img_base + 4*i + k]*
6                     kernel[kernel_base + j + 4*k];
7                 }
8             }
9         }
```

Listing 1: B: Baseline implementation

Note the following:

- Loops i, k, j are present **inside** the k_x, k_y, c_i loop (line 1) in equation 2. We are only examining the core matrix multiplication.
- *idx, image_base, kernel_base* (lines 5, 6) are computed using the block index, thread index and the values of current iteration of k_x, k_y, c_i loop. They are used to index into the input image and the filter coefficients.

4.1 Scalar accumulation

Our first improvement involves a simple temporary variable for accumulating the dot product entries for each element of the output 4x4 matrix. This allows us to reduce the number of stores into the output array which is stored in global memory.
The code is shown below:

```
1     (loop over kx, ky, channel)
2         for (int i = 0; i < 4; i++){
3             for (int j = 0; j < 4; j++){
4                 float cur_val = 0;
5                 for (int k = 0; k < 4; k++){
6                     cur_val += img[img_base + 4*i + k]*kernel[kernel_base + j + 4*k];
7                 }
8                 out[idx*16 + i*4 + j] = cur_val;
9             }
10        }
```

Listing 2: I1: Scalar accumulation

Note the introduction of a temporary variable *cur_val* (line 4) which is used to accumulate the dot product entries. The temporary variable is then stored in the output array (line 8).

4.2 Unrolling the loops

By unrolling all 3 loops, we can expect some performance improvement since we can reuse some of the values loaded from memory by storing them in registers. This allows us to reduce the number of loads from the input image and the filter coefficients.
For brevity purposes, only the first two statements of the code are shown below. Please refer to appendix A for the full code.

```
1     out[idx*16 + 0] = img[ib + 0]*kernel[kb + 0]+
2                       img[ib + 1]*kernel[kb + 4]+
3                       img[ib + 2]*kernel[kb + 8]+
4                       img[ib + 3]*kernel[kb + 12];
5
6     out[idx*16 + 4] = img[ib + 4]*kernel[kb + 0]+
7                       img[ib + 5]*kernel[kb + 4]+
8                       img[ib + 6]*kernel[kb + 8]+
9                       img[ib + 7]*kernel[kb + 12];
10    ...
```

Listing 3: I2: Loop unrolling (partial code)

In other words, we explicitly write out each entry of the 4x4 output matrix as a sum of 4 dot products.

4.3 Scalar replacement

Ideally, a smart compiler would detect the repeated loads to the same memory location (of *img, kernel*) and replace them with a scalar so as to avoid redundant loads from global/shared memory and instead store these in registers. However, we can do this manually by introducing a temporary variable for each entry of the input image and the kernel coefficients.
A snippet of the code is given below. The entire code can be found in appendix B.

```
1     float t0 = img[ib + 0];
2     float t1 = img[ib + 1];
3     ...
4
5     float k0 = kernel[kb + 0];
6     float k1 = kernel[kb + 1];
7     ...
8
9     out[idx*16 + 0] = t0*k0+t1*k4+t2*k8+t3*k12;
10    out[idx*16 + 4] = t4*k0+t5*k4+t6*k8+t7*k12;
```

Listing 4: I3: Scalar replacement (partial code)

In lines 1-6, temporary variables are introduced in order to store the *img, kernel* read from memory. In lines 9-10, these values are used to compute the output and store it in the output array.

5 Experiments

In this section, we mention the experimental setup and the results obtained for each of the code transformations mentioned above. We also discuss the profiler output and the running times for each of the code transformations.

5.1 Problem setup

The hyperparameters used for the experiments are given below:

Hyperparameter	Notation	Value
Image size	$N_x = N_y$	128
Filter size	$K_x = K_y$	5
Number of input channels	c_i	3

Table 1: Hyperparameters used for the experiments

We use the Tesla V100 GPU (PACE cluster) for all our experiments. It has a maximum of 96 kilobytes of shared memory. This restricts the size of k_x, k_y, c_i as the kernel coefficients are stored in shared memory.

5.2 Results

The running times, along with global loads and stores for each of the code transformations (extracted using the NVIDIA profiler *nvprof*) are given below. The running times are averaged over 100 runs.

Implementation	Running time (ms)	Global stores	Global loads
B (baseline)	24.693	393625600	488843700
I1 (Scalar accumulation)	8.3399 (↓0.33x)	98406400 (↓0.25x)	488470968 (≈1x)
I2 (Loop unrolling)	6.9763 (↓0.28x)	98406400 (↓0.25x)	392512360 (↓0.80x)
I3 (Scalar replacement)	4.9513 (↓0.20x)	98406400 (↓0.25x)	97851359 (↓0.20x)

Table 2: Comparison of execution times and the number of global load/store transactions for each of the code transformations. Reduction factors (depicted using ↓) are with respect to the baseline implementation B.

We can reason for the performance improvements as follows:

1. **I1:** The speedup is due to the reduced number of stores to output array. A reduction of 0.25x in the number of stores aligns perfectly with our expectations as the 64 stores per matrix multiplication in the baseline are replaced with 16 stores in I1. The number of global loads remains (approximately) the same since we still need to load the input image and the kernel coefficients from global and shared memory respectively.

2. **I2:** The additional speedup (compared to I1) is due to the reduced number of global loads. The slight reduction in number of global loads can be attributed to some compiler optimizations for storing the loaded variables in temporary registers. The number of global stores remains the same as I1 (expected).[2]

3. **I3:** Lastly, on introducing scalar replacement, we observe a further reduction in the number of global loads. This is because the compiler is able to store the loaded variables in registers and reuse them for the multiply and accumulate computations. The number of global stores remains the same as I1 and I2 (expected). Also note that the speedup in running time is the exact same as the reduction in number of global loads (despite I3 having only 25% of the stores). This indicates that the running time is dominated by the number of global loads beyond a certain point.

We did not observe any major difference in the compilation times of the three implementations.

6 Conclusion

In this work, we motivated the use of capsules and highlighted the lack of tuned CUDA kernels for the proposed operator (drawing from extensive discussions in [BI19]). We then used standard techniques in compilers to optimize the baseline implementation of the capsule operator. We observed that the running time is dominated by the number of global loads and introducing scalars to hold these variables led to a **5x** improvement in performance.
The work can be extended in several different direction:

- Improve cache performance using tiling.
- Study advanced techniques such as cooperatively loading and exchaning data values between threads in a block.
- Comparison with code generated by other frameworks such as Tensor Comprehension and PlaidML.

[2]We could not measure the register spill rate using *nvprof* and hence could not verify this claim.

- Increase scope of this work by allowing $n, c_o \geq 1$.
- Optimize end-to-end by drawing inspiration from the optimized convolution operator and develop a PyTorch extension for the capsule operator.

7 Acknowledgements

Firstly, I would like to thank Prof. Sarkar for his guidance. The compiler optimization techniques taught in class led to a noticeable improvement in performance. The feedback on the project presentation was very helpful. I would also like to thank my classmates for the insightful discussions. Finally, I would like to thank the PACE cluster at GeorgiaTech for providing the necessary compute resources.

References

[BI19] Paul Barham and Michael Isard. Machine learning systems are stuck in a rut. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 177–183, 2019.

[SFH17] Sara Sabour, Nicholas Frosst, and Geoffrey E Hinton. Dynamic routing between capsules. *Advances in neural information processing systems*, 30, 2017.

A Code for I2: Unrolling the loop

We simply unroll the loops and write out each entry of the 4x4 output matrix explicitly as the sum of 4 products. *ib, kb* refer to the *kernel_base, img_base* described previously.

```
    out[idx*16 + 0] = img[ib + 0]*kernel[kb + 0]+img[ib + 1]*kernel[kb + 4]+img[ib +
    2]*kernel[kb + 8]+img[ib + 3]*kernel[kb + 12];
    out[idx*16 + 4] = img[ib + 4]*kernel[kb + 0]+img[ib + 5]*kernel[kb + 4]+img[ib +
    6]*kernel[kb + 8]+img[ib + 7]*kernel[kb + 12];
    out[idx*16 + 8] = img[ib + 8]*kernel[kb + 0]+img[ib + 9]*kernel[kb + 4]+img[ib +
    10]*kernel[kb + 8]+img[ib + 11]*kernel[kb + 12];
    out[idx*16 + 12] = img[ib + 12]*kernel[kb + 0]+img[ib + 13]*kernel[kb + 4]+img[ib
    + 14]*kernel[kb + 8]+img[ib + 15]*kernel[kb + 12];
    out[idx*16 + 1] = img[ib + 0]*kernel[kb + 1]+img[ib + 1]*kernel[kb + 5]+img[ib +
    2]*kernel[kb + 9]+img[ib + 3]*kernel[kb + 13];
    out[idx*16 + 5] = img[ib + 4]*kernel[kb + 1]+img[ib + 5]*kernel[kb + 5]+img[ib +
    6]*kernel[kb + 9]+img[ib + 7]*kernel[kb + 13];
    out[idx*16 + 9] = img[ib + 8]*kernel[kb + 1]+img[ib + 9]*kernel[kb + 5]+img[ib +
    10]*kernel[kb + 9]+img[ib + 11]*kernel[kb + 13];
    out[idx*16 + 13] = img[ib + 12]*kernel[kb + 1]+img[ib + 13]*kernel[kb + 5]+img[ib
    + 14]*kernel[kb + 9]+img[ib + 15]*kernel[kb + 13];
    out[idx*16 + 2] = img[ib + 0]*kernel[kb + 2]+img[ib + 1]*kernel[kb + 6]+img[ib +
    2]*kernel[kb + 10]+img[ib + 3]*kernel[kb + 14];
    out[idx*16 + 6] = img[ib + 4]*kernel[kb + 2]+img[ib + 5]*kernel[kb + 6]+img[ib +
    6]*kernel[kb + 10]+img[ib + 7]*kernel[kb + 14];
    out[idx*16 + 10] = img[ib + 8]*kernel[kb + 2]+img[ib + 9]*kernel[kb + 6]+img[ib +
    10]*kernel[kb + 10]+img[ib + 11]*kernel[kb + 14];
    out[idx*16 + 14] = img[ib + 12]*kernel[kb + 2]+img[ib + 13]*kernel[kb + 6]+img[ib
    + 14]*kernel[kb + 10]+img[ib + 15]*kernel[kb + 14];
    out[idx*16 + 3] = img[ib + 0]*kernel[kb + 3]+img[ib + 1]*kernel[kb + 7]+img[ib +
    2]*kernel[kb + 11]+img[ib + 3]*kernel[kb + 15];
    out[idx*16 + 7] = img[ib + 4]*kernel[kb + 3]+img[ib + 5]*kernel[kb + 7]+img[ib +
    6]*kernel[kb + 11]+img[ib + 7]*kernel[kb + 15];
    out[idx*16 + 11] = img[ib + 8]*kernel[kb + 3]+img[ib + 9]*kernel[kb + 7]+img[ib +
    10]*kernel[kb + 11]+img[ib + 11]*kernel[kb + 15];
          out[idx*16 + 15] = img[ib + 12]*kernel[kb + 3]+img[ib + 13]*kernel[kb + 7]+
    img[ib + 14]*kernel[kb + 11]+img[ib + 15]*kernel[kb + 15];
```

Listing 5: I2: Loop unrolling (entire code)

B Code for I3: Scalar replacement

We declare temporary variables to store the values of *img[ib + i]* and *kernel[ib + i]* and use them in the computation of the output matrix. *ib, kb* refer to the *kernel_base, img_base* described previously.

```
    float t0 = img[ib + 0];
    float t1 = img[ib + 1];
    float t2 = img[ib + 2];
    float t3 = img[ib + 3];
    float t4 = img[ib + 4];
    float t5 = img[ib + 5];
    float t6 = img[ib + 6];
    float t7 = img[ib + 7];
    float t8 = img[ib + 8];
    float t9 = img[ib + 9];
    float t10 = img[ib + 10];
    float t11 = img[ib + 11];
    float t12 = img[ib + 12];
    float t13 = img[ib + 13];
    float t14 = img[ib + 14];
    float t15 = img[ib + 15];

    float k0 = kernel[kb + 0];
    float k1 = kernel[kb + 1];
    float k2 = kernel[kb + 2];
    float k3 = kernel[kb + 3];
    float k4 = kernel[kb + 4];
    float k5 = kernel[kb + 5];
    float k6 = kernel[kb + 6];
```

```
    float k7 = kernel[kb + 7];
    float k8 = kernel[kb + 8];
    float k9 = kernel[kb + 9];
    float k10 = kernel[kb + 10];
    float k11 = kernel[kb + 11];
    float k12 = kernel[kb + 12];
    float k13 = kernel[kb + 13];
    float k14 = kernel[kb + 14];
    float k15 = kernel[kb + 15];

    out[idx*16 + 0] = t0*k0+t1*k4+t2*k8+t3*k12;
    out[idx*16 + 4] = t4*k0+t5*k4+t6*k8+t7*k12;
    out[idx*16 + 8] = t8*k0+t9*k4+t10*k8+t11*k12;
    out[idx*16 + 12] = t12*k0+t13*k4+t14*k8+t15*k12;
    out[idx*16 + 1] = t0*k1+t1*k5+t2*k9+t3*k13;
    out[idx*16 + 5] = t4*k1+t5*k5+t6*k9+t7*k13;
    out[idx*16 + 9] = t8*k1+t9*k5+t10*k9+t11*k13;
    out[idx*16 + 13] = t12*k1+t13*k5+t14*k9+t15*k13;
    out[idx*16 + 2] = t0*k2+t1*k6+t2*k10+t3*k14;
    out[idx*16 + 6] = t4*k2+t5*k6+t6*k10+t7*k14;
    out[idx*16 + 10] = t8*k2+t9*k6+t10*k10+t11*k14;
    out[idx*16 + 14] = t12*k2+t13*k6+t14*k10+t15*k14;
    out[idx*16 + 3] = t0*k3+t1*k7+t2*k11+t3*k15;
    out[idx*16 + 7] = t4*k3+t5*k7+t6*k11+t7*k15;
    out[idx*16 + 11] = t8*k3+t9*k7+t10*k11+t11*k15;
    out[idx*16 + 15] = t12*k3+t13*k7+t14*k11+t15*k15;
```

Listing 6: I3: Scalar replacement (entire code)