Convolutional Neural Networks for Object Classification in CUDA

Alex Krizhevsky (kriz@cs.toronto.edu)

April 16, 2009

1 Introduction

Here I will present my implementation of a simple convolutional neural network in CUDA. The network takes as input a 32×32 colour image and produces as its output one of ten possible class labels. The convolution operations, which account for 90% of the time required to train this network, are 125-150x faster on the GPU than on an Intel Core 2 Duo 2.4GHz.

2 Related work

I have not found any other implementations of 2D convolution. The CUDA SDK has an implementation for the separable case, but that is quite different. I am interested in non-separable filters. I have heard from Yann LeCun of NYU (the primary pusher of convolutional neural nets) that students at UC Irvine have claimed a 100x speedup of convolutional neural net implementations on the GPU.

3 Neurons

A neuron is simply a function. It takes an input x, computes some function $f(x)$ and outputs it. In my implementation, I use neurons that compute the logistic function

$$f(x) = \frac{1}{1 + e^{-x}},$$

which is plotted in Figure 1. This is by far the most commonly used function. It has the nice property that its output is effectively linear in the input if the size of the input is small. This means that neural networks with small weights essentially compute a linear function, and gradually increasing the weights allows one to control the degree of nonlinearity. The degree of nonlinearity controls the "capacity" of the neural network. For example, a network with purely linear neurons can only compute linear functions.

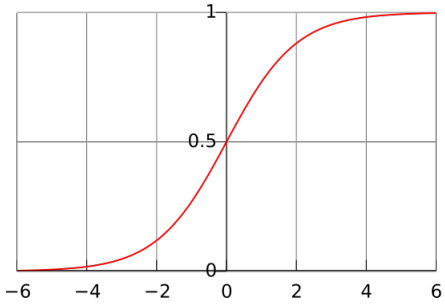

Figure 1: The logistic function $f(x) = \frac{1}{1+e^{-x}}$.[1]

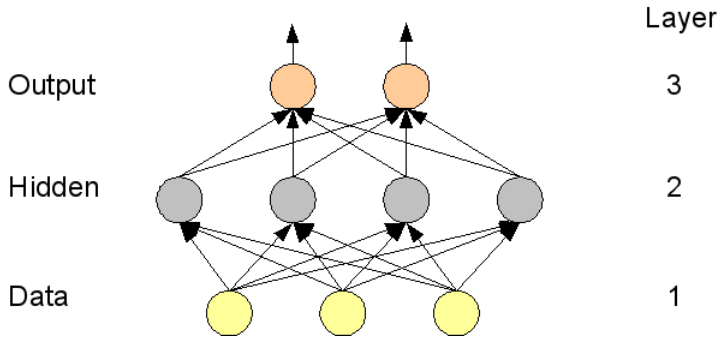

Figure 2: A feed-forward neural network with one hidden layer.

4 Feed-forward neural networks

Feed-forward neural networks link together layers of neurons. Every neuron in layer l is connected to every neuron in layer $l+1$, but there are no intra-layer connections. This architecture is depicted in Figure 2. On each connection there is a real-valued weight. Neuron k in layer l receives as input the value

$$x_k^l = b_k^l + \sum_{i=1}^{N_{l-1}} w_{ik}^{l-1} y_i^{l-1}$$

where

- b_k^l is the bias into neuron k in layer l,
- N_{l-1} is the number of neurons in layer $l-1$,
- w_{ik}^{l-1} is the weight between unit i in layer $l-1$ and unit k in layer l, and
- y_i^{l-1} is the output of unit i in layer $l-1$.

The neuron then computes its output

$$y_k^l = f(x_k^l)$$

where f is any differentiable function of the neuron's total input (as mentioned, I use the logistic function). The "neurons" in the data layer just output the data.

Finally, we come up with a function

$$E(y_1^L, \ldots, y_{N_L}^L)$$

of the output that we would like the neural net to maximize (this can be seen as just another layer on top of the output layer), where L is the number of layers in the neural network. E should be differentiable so $\frac{\partial E}{\partial y_k^L}$ is readily computable. Training the network consists of clamping the data neurons at the data and updating the parameters (the weights and biases) in the direction of the gradient. The derivatives can be computed as follows:

$$\begin{aligned} \frac{\partial E}{\partial w_{ik}^{l-1}} &= \frac{\partial E}{\partial x_k^l} y_i^{l-1} \\ \frac{\partial E}{\partial b_k^l} &= \frac{\partial E}{\partial x_k^l} \end{aligned}$$

where

$$\begin{aligned} \frac{\partial E}{\partial x_k^l} &= \frac{\partial E}{\partial y_k^l} \frac{\partial y_k^l}{\partial x_k^l} \\ \frac{\partial E}{\partial y_k^l} &= \begin{cases} \frac{\partial E}{\partial y_k^L} & \text{if } l = L \\ \sum_{i=1}^{N_{l+1}} \frac{\partial E}{\partial x_i^{l+1}} w_{ki}^l & \text{otherwise} \end{cases} \end{aligned} \tag{1}$$

and $\frac{\partial E}{\partial y_k^L}$ is assumed to be readily computable. From this, derivatives with respect to all the weights and biases can be computed, working down from the top layer. This is known as the backpropagation algorithm. Typically, the gradient is averaged over a whole batch of images, and then the parameters are updated with this average gradient. This is known as batch learning.

If the network is to be used for classification, as in my case, the number of outputs is typically set at the number of possible classes. The output of each output unit then corresponds in some way to the network's belief in some class label (I will get more specific about this in the next section when I describe my network).

5 Convolutional neural networks

In ordinary feed-forward neural networks, every neuron in layer l is connected to every neuron in layer $l-1$. This is fine if there is no local structure in the activations of the neurons of layer $l-1$. But images have just this kind of structure. Nearby pixels are highly correlated and faraway pixels are weakly correlated. Therefore we would like to encourage the neural net to focus on extracting local features of the image. Convolutional nets achieve this by connecting each unit in the hidden layer to only a small local patch of units in the image. In my case, this patch is 8×8. A convolutional net with one hidden unit will apply this 8×8 filter at all 33×33 possible locations in the 32×32 image (with a 4px-wide padding of zeros on each edge). The outputs of the hidden layer will then consist of the 33×33 outputs of this one filter, applied all over the image. This is where the convolution comes in.

But having just one hidden unit is too limiting, even though its filter is applied all over the image and thus the unit has over a thousand outputs. We would like to have lots of hidden units that learn different filters, each applied all over the image. This scenario is depicted in Figure 3. In the figure there are F hidden units in total, and each 32×32 plate represents the 32×32 outputs produced by one hidden unit by convolving its 8×8 filter with the image and applying the logistic function. There are 32×32 outputs instead of 33×33 because GPUs are more fond of numbers like 32 than 33, and the neural net will not miss those few extra missing outputs at the edges of the image. In a standard neural net, to get 32×32 outputs from the hidden layer, we would need it to have 32×32 units in that hidden layer. Here this is achieved with just one unit. So we can think of convolutional neural nets as ordinary neural nets, but with the constraint that certain groups of units must share weights (and also that the units be only locally-connected to the image).

Clearly, the outputs of a hidden unit when applied to nearby regions of the image will be similar. Therefore the values in nearby regions of the same 32×32 plate of hidden unit outputs (Figure 3) in the hidden layer are similar. So convolutional nets often include a subsampling layer right above the convolutional layer, to locally average nearby hidden unit responses. In my network, the averaging units average 4×4 non-overlapping patches of hidden unit outputs. The averaging layer reduces the size of the plates of hidden unit responses from 32×32 to 8×8. It can also be argued that losing some precision as to the exact location of the features in the image is actually advantageous, because it achieves a greater degree of invariance. It often turns out in vision that the precise location of a feature is not as important as its approximate location relative to other features.

The averaging units produce a combined total of $8 \times 8 \times F$ outputs. From this point on, the network is just an ordinary fully-connected feed-forward network. The network has 10 outputs because my dataset consists of 60,000 images in 10 classes. The $8 \times 8 \times F$ averaging units are connected to all 10 of the output units. Output unit k computes the function

$$y_k = \frac{e^{x_k}}{\sum_{j=1}^{10} e^{x_j}}$$

where x_k is the total input going into output unit k. Since $\sum_{k=1}^{10} y_k = 1$, y_k is interpreted as the probability that the network assigns to class label k. To train the network, I maximize the average log probability of getting the correct label. To evaluate the network's performance, I show it an image and compare the class to which it assigns the highest probability with the true label of the image. In my results I simply report the percentage of labels that it got correct.

Convolutional neural nets are actively developed by Yann LeCun's group at New York University, and they have been shown to achieve state-of-the-art performance in handwritten digit recognition as well as other classification tasks. The types of networks that achieve such performance are typically more complicated than the one that I have implemented. Specifically, they have more than one layer of convolution. But the principle is the same.

5.1 Training convolutional neural networks

Mathematically, the algorithm for training convolutional neural networks is the same as it is for training ordinary feed-forward networks. This is because, as mentioned above, a convolutional network can be viewed as a feed-forward network with weight-sharing constraints. In the next section I will describe in more detail the GPU kernels that need to be implemented to compute the various derivatives.

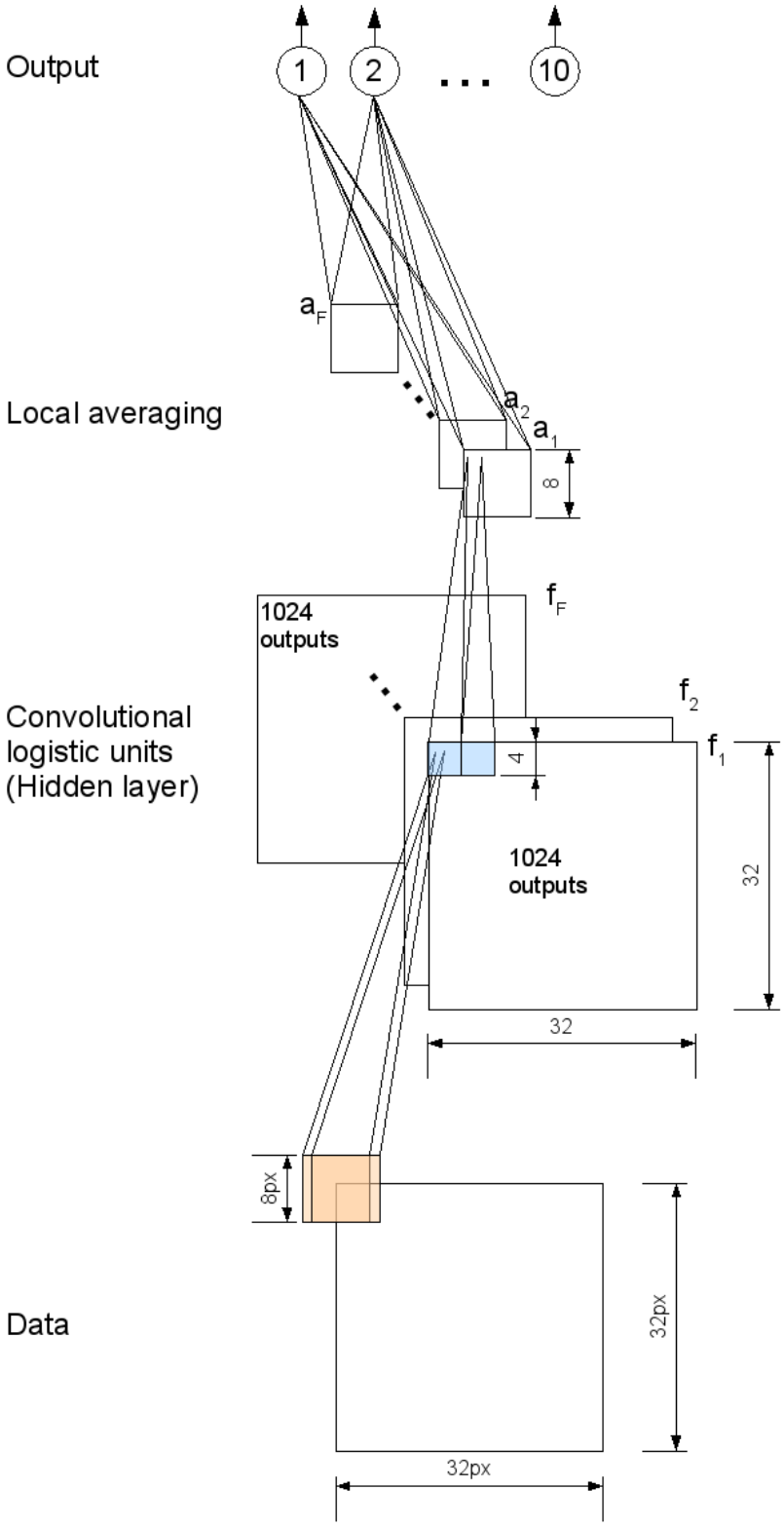

Figure 3: The architecture of the convolutional neural network that I implemented. The convolutional units compute the logistic function: $f(x) = \frac{1}{1+e^{-x}}$. The output layer is a standard logistic regression classifier: $f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{10} e^{x_j}}$.

Not all the connections are shown, to avoid cluttering the diagram. For simplicity, this diagram does not make it explicit that the images and filters are colour. In reality, each image should be thought of as having dimensions $32 \times 32 \times 3$ and each filter $8 \times 8 \times 3$.

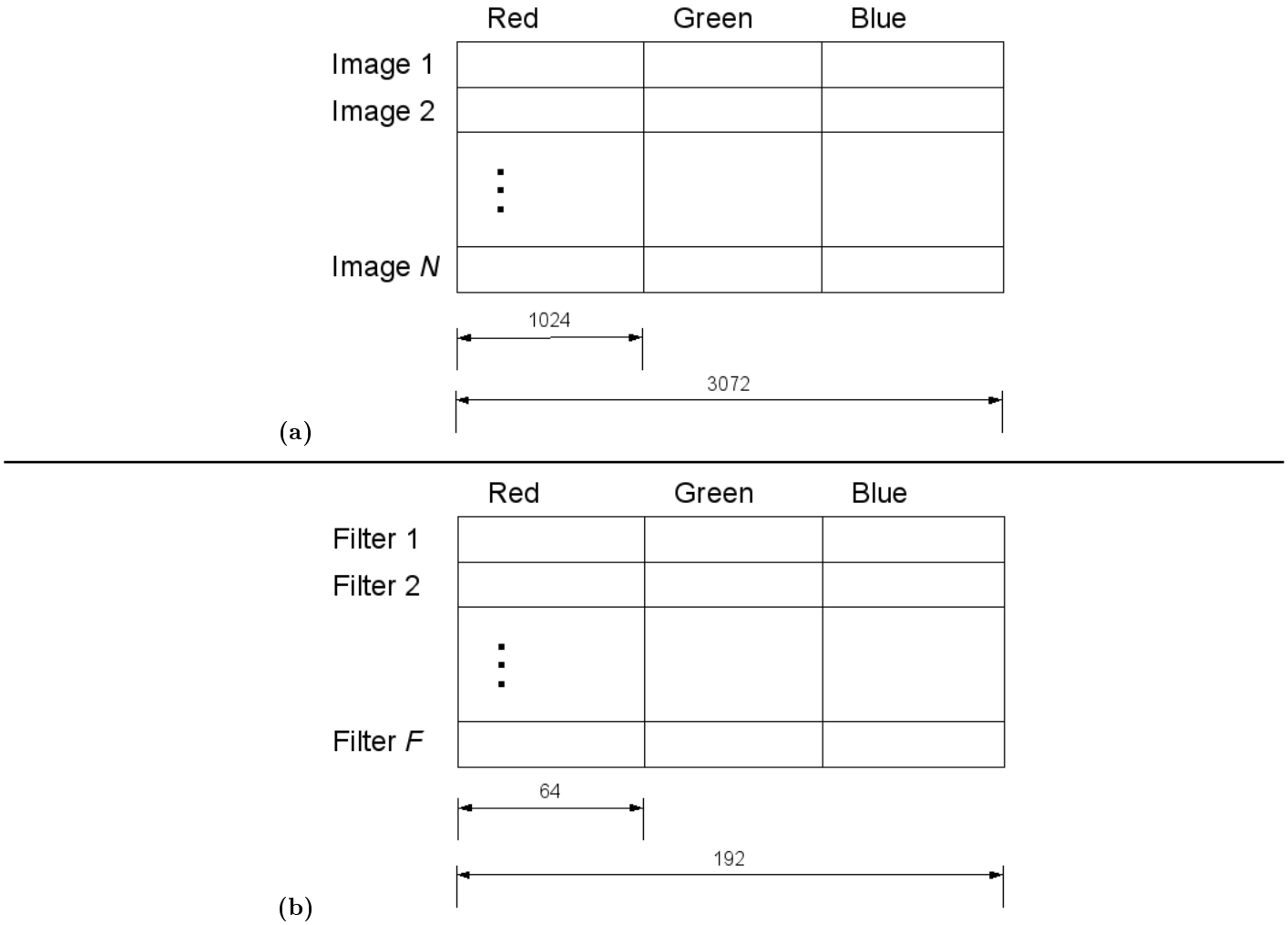

Figure 4: The layout of **(a)** the images and **(b)** the filters in memory. Note that both images and filters are stored as *floats* – each pixel consumes 4 bytes of memory. This is because in many applications, the data will have been pre-processed in some way, or normalized to be in the range 0-1, etc. The filters too must be floats because the filters represent the weights on the connections of the neural net. One cannot take the derivative with respect to a discrete quantity.

6 Implementation in CUDA

6.1 Image format

The first step in training the convolutional net is to convolve the F filters of the net with the N images that make up a batch of training cases. I generally used 64 filters and batches of 128 images, although these values are runtime variables. Figure 4 makes clear the exact format of the data and filters.

6.2 Convolution of 8x8 filters with 32x32 images

The algorithm for doing this convolution is explained in Figure 5. It uses a block size of $8 \times 32 = 256$. Each block convolves one image with one filter, so the grid size is $F \times N$. The algorithm uses 14 registers and 3264 bytes of shared memory, so it achieves 100% occupancy. It is 125x faster than the equivalent CPU algorithm on a Core 2 Duo 2.4GHz when $F = 64, N = 128$. This convolution takes 13.18ms on the GPU. This is the best algorithm that I have been able to find. I have tried variants in which each thread convolves more than one filter with the image, but they perform slightly worse (they also don't achieve 100% occupancy). I have also tried a variant that uses 512 threads per block and loads the entire image into shared memory at once. It also performed slightly worse because loading was slightly more complicated.

I have made some effort to make sure that there are no warp-splitting if statements. In the few conditionals that exist, all threads in the same warp take the same branch (except the initial filter load). I have checked that removing all loads

and their corresponding if statements improves the algorithm's performance by only 2%, so memory bandwidth does not appear to be a bottleneck. All of the loops have bounds that are known at compile time, so they are unrolled by the compiler. All global memory accesses are coalesced and there are no bank conflicts in accessing shared memory. I have also tried to minimize the amount of pointer arithmetic. I got a slight speed improvement by incrementing pointers at each loop iteration rather than recomputing the index of the array that I want to access.

6.3 Subsampling (averaging) layer

Figure 6 makes clear the goal of this step. We need to go from a 32×32 array of floats to an 8×8 array by averaging every non-overlapping 4×4 region. Figure 7 describes the algorithm. The algorithm uses blocks of 8×32 threads so the grid size is again $F \times N$. The algorithm has 2-way bank conflicts in one of its steps, but avoids bank conflicts in all other steps. All global memory accesses are coalesced. The index computations are particularly simple because they only involve division by powers of 2. Avoiding all bank conflicts requires slightly more complicated index computations which negate the benefits. All of the reduction in this algorithm is performed by threads in the same warp so little synchronization is necessary. In any case, this step accounts for less than 1% of the time required to train the convolutional net. The kernel uses 1152 bytes of shared memory and 10 registers, and so it achieves full occupancy. This kernel is 31x faster than its CPU counterpart when $F = 64, N = 128$.

6.4 Output and derivative computation

Once we have the averaged outputs of the previous step, computing the outputs of the neural net is a trivial task since the remaining layers are fully connected. I will not go into detail for lack of space; I will just say that it involves some calls to CUBLAS. The derivatives with respect to the output units and the averaging units are likewise computed with a few matrix multiplications.

6.5 Derivatives with respect to convolutional hidden units

Given the derivatives with respect to the outputs of the averaging units, it is conceptually simple to compute the derivatives with respect to the outputs of the convolutional units. The averaging units simply locally average the convolutional unit outputs, so the derivative with respect to a given convolutional unit output will be the same as the derivative with respect to the averaging unit assigned to its region. This means we require a supersampling kernel. This step is essentially the reverse of the step of step 6.3. Figure 8 depicts the goal of this step.

There is one minor complication – that is that the CUBLAS SGEMM (matrix multiplication) routine outputs its result in column-major order. Since I used CUBLAS to compute the derivatives with respect to the activities of the averaging units, I will have to deal with this (because I want the output of the kernel to be in row-major order for the next step). Figure 9 depicts the format of the input matrix.

Figure 10 explains the implementation of the algorithm. All reads and writes are coalesced and there are no bank conflicts. When 4 threads read the same value, that value should be broadcast so there should be no bank conflict there. This kernel as well achieves 100% occupancy and accounts for less than 1% of the time spent training the convolutional net. This kernel is 58x faster than its CPU counterpart when $F = 64, N = 128$.

This kernel computes derivatives with respect to the outputs of the convolutional units. Computing derivatives with respect to the *inputs* to the convolutional units is accomplished with a simple vector element-wise multiplication, since from equation (1) we know

$$\begin{aligned} \frac{\partial E}{\partial x_k^l} &= \frac{\partial E}{\partial y_k^l}\frac{\partial y_k^l}{\partial x_k^l} \\ &= \frac{\partial E}{\partial y_k^l}(y_k^l)(1 - y_k^l), \end{aligned}$$

the latter part is due to the fact that $y_k^l = \frac{1}{1+e^{-x_k^l}}$.

6.6 Derivatives with respect to weights between data and convolutional units

Once we have the derivatives with respect to the inputs to the convolutional units, the final step is to compute the derivatives with respect to the weights between the data units and the convolutional units. For a particular filter and image pair, the derivative with respect to weight w_{ik} is the average over all applications of the filter of the derivative with

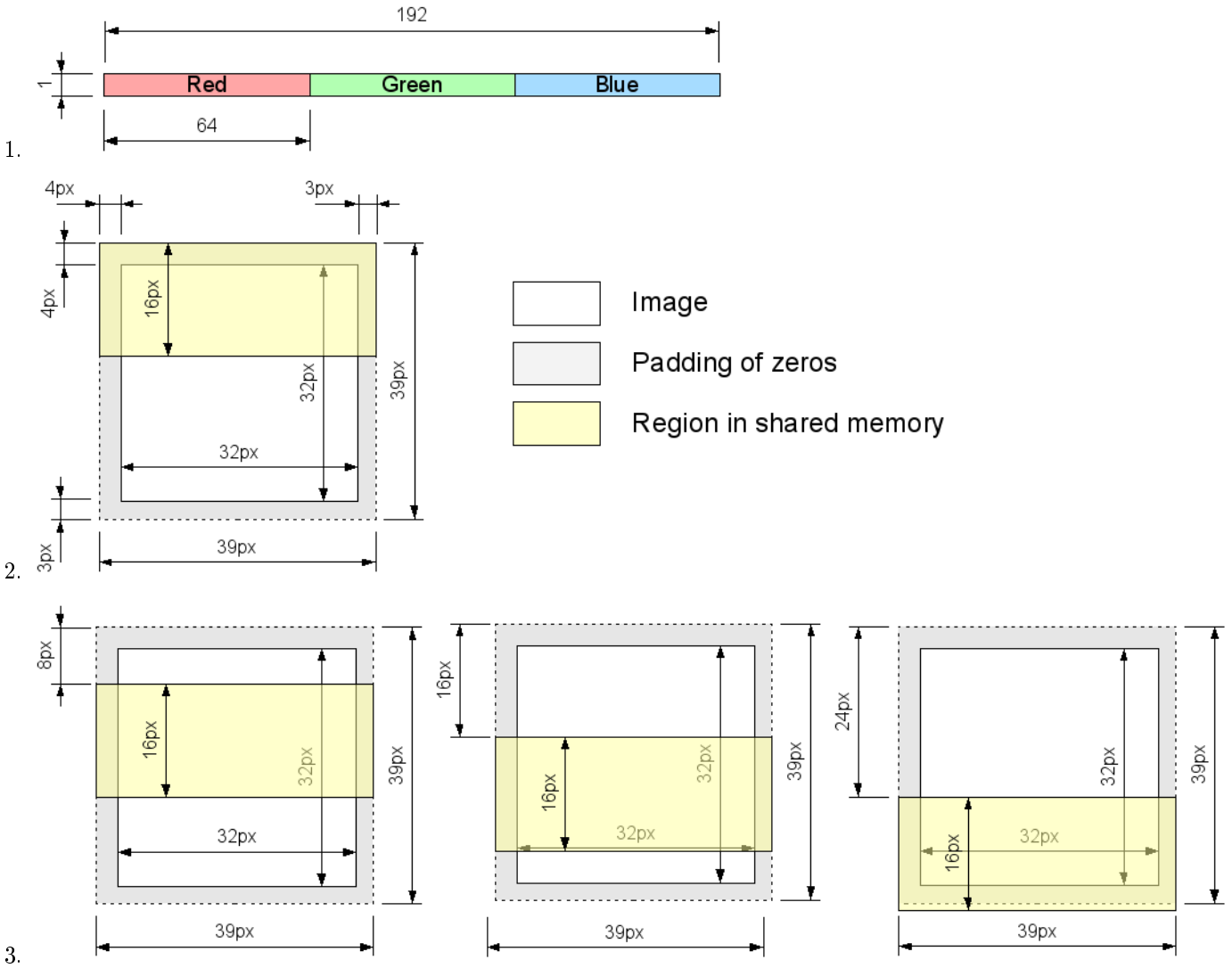

Figure 5: The algorithm for convolving 8×8 colour filters with 32×32 colour images with 4px padding of zeros on the left and top edges and 3px padding on the right and bottom edges. The block size is $(y, x) = (8, 32)$. Each block convolves one filter with one image.

(1) 3/4 of the threads load the entire filter into shared memory. Although they do not need all three colour channels at once, there is a slight advantage to loading them all together, and the shared memory is available.
(2) The threads allocate a 16×39 region of shared memory (highlighted) and initialize it with zeros. The threads then load the 16×32 chunk of the (red colour channel of the) image that corresponds to the non-border area. The 8×32 threads then compute the 8×32 dot products with this chunk of the image and store the result in a local variable. Each thread computes one dot product. This result will later have to be added to the result from the blue and green colour channels and then written out to global memory.
(3) The threads shift the shared memory window down by 8 pixels and perform the 8×32 dot products with the new image chunk. They repeat the process twice more. Each time, there are 8×32 dot products to perform. At the last step, the window is seen to exceed the size of the padding because there is only 3px of padding at the bottom. This points out the fact that the threads really only need a 15×39 region of shared memory, but the extra row doesn't hurt.

After the above steps, each thread has 4 local variables with the results of the 4 dot products that the thread computed. The threads now load the green channel of the image and repeat the above steps for the green channel of the filter, adding to the 4 local variables. Then the threads repeat the steps for the blue channel. Finally, each thread writes its 4 outputs to global memory.

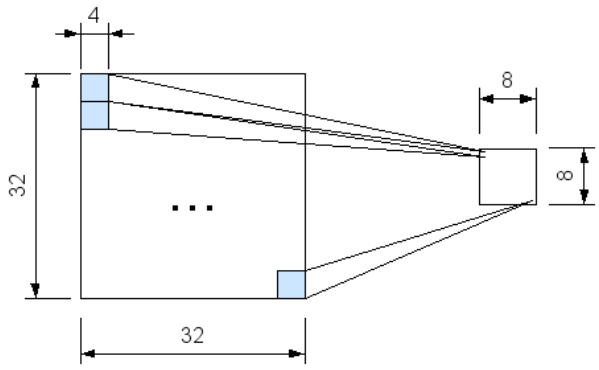

Figure 6: The goal of the local averaging kernel: to go from the 32×32 array of floats (left) to the 8×8 array (right) by averaging the highlighted 4×4 regions.

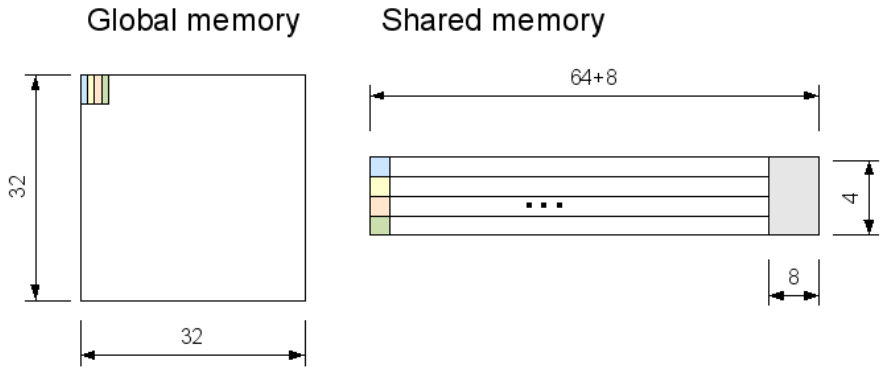

Figure 7: The local averaging algorithm. The block size is $(y, x) = (8, 32)$. Each block subsamples one 32×32 plate of hidden unit outputs (pictured left). The threads allocate a 4×64 array in shared memory with 8 padding columns, to avoid bank conflicts later.

The values are summed as pictured. Thread 0 sums the 4 values in the blue column and writes the result to the corresponding cell in shared memory. Thread 1 sums the yellow column, thread 2 the orange, and thread 3 the green. The rest of the threads proceed in an analogous fashion. Notice that this produces 2-way bank conflicts since threads 0 and 2 write to the same bank, as do 1 and 3, 4 and 6, etc. The reads from global memory are of course coalesced.

After this step is complete, the threads in the first 8 warps do the remaining reduction by summing out the columns of the shared memory array, 2 threads per column. So each warp sums out 16 columns. No synchronization is necessary in this step. At this step there are no bank conflicts because thread i reads from bank $(i\%2)\times 8+i/2$ (modulo 16).

Once all the final values are written to the first row of the shared memory array, the first two warps divide those values by 16 and write them out to global memory. This step also has no bank conflicts because thread i accesses bank i (modulo 16). The writes to global memory are coalesced.

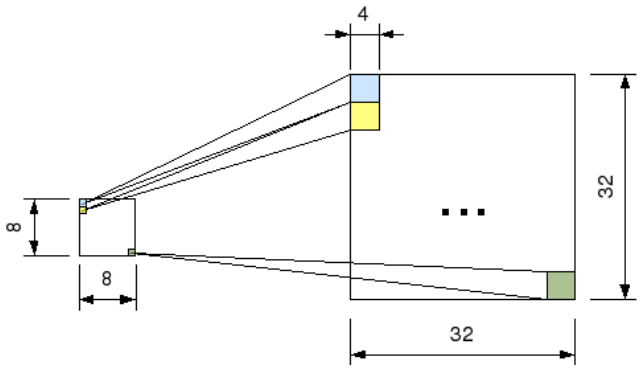

Figure 8: The goal of the kernel of step 6.5 for one image and one filter. We need to supersample the derivatives with respect to the averaging units (left) by a factor of 4×4. So each value on the left must get written to 16 locations on the right.

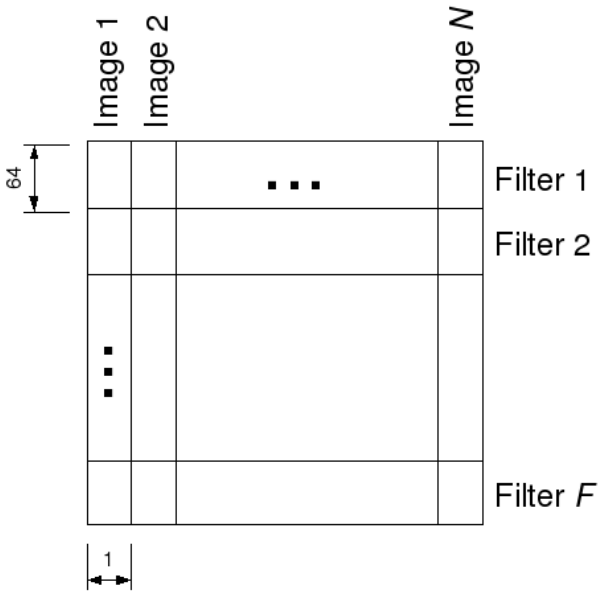

Figure 9: The layout of the input matrix for step 6.5. These are derivatives with respect to the outputs of the averaging units. The images are in columns because the CUBLAS SGEMM routine outputs in column-major order.

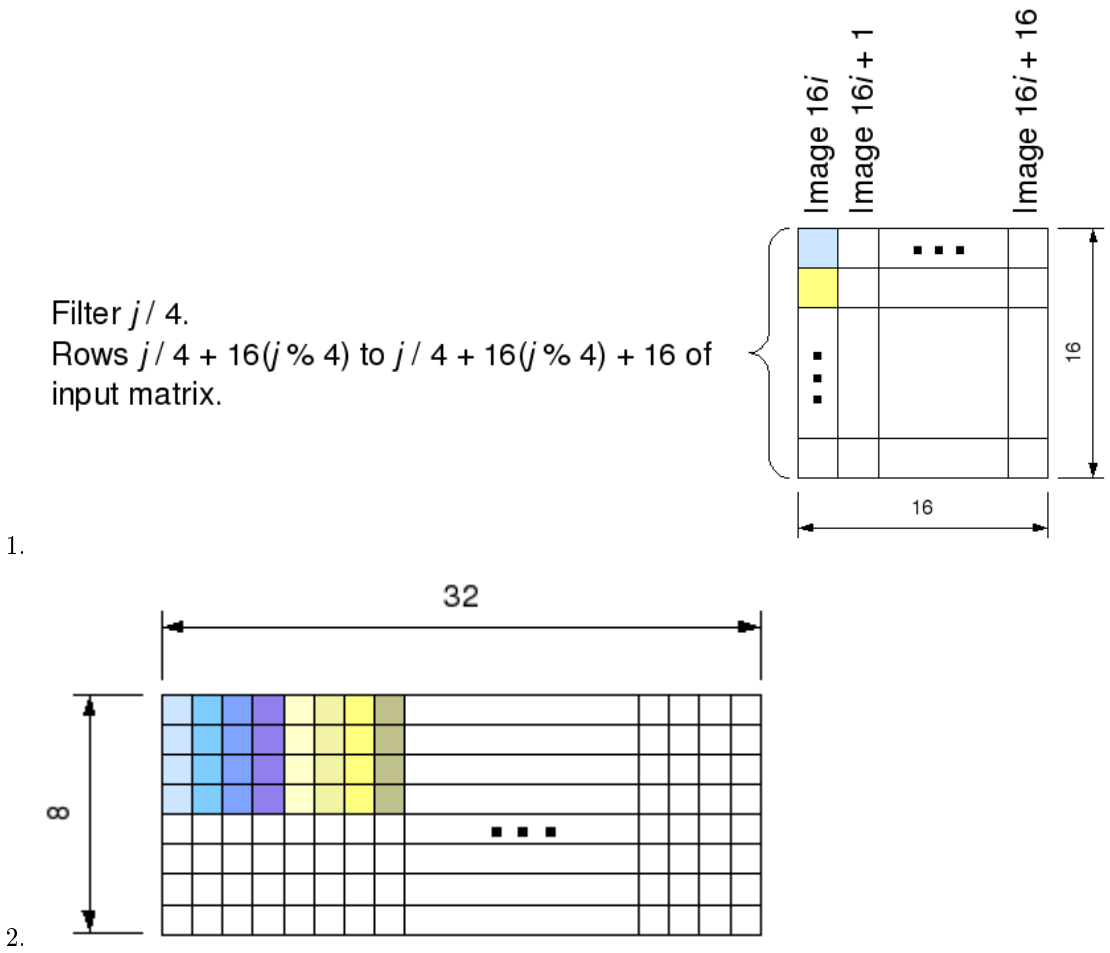

Figure 10: The supersampling algorithm of step 6.5. The block size is 16×16. The grid size is $(y, x) = (64F/16, N/16)$. The grid simply partitions the input matrix into 16×16 chunks (refer to Figure 9 for the input layout).

(1) Each block allocates a $16 \times (16 + 1)$ region of shared memory and reads its chunk of the input matrix into shared memory, without transposing it. The threads supersample the matrix in the following way: Threads 0, 1, 2, 3 read the blue value $(0, 0)$ of **(1)**. Threads 4, 5, 6, 7 read the yellow value $(1, 0)$. The rest of the threads proceed analogously. Note that there are only enough threads to read four of the columns of the shared memory array in this way.
(2) Each thread then writes the value it read to four different rows of the output matrix. The hue in **(2)** indicates the thread that is doing the writing. Thread 0 writes the first column, thread 1 the second, etc. *(Note that the threads that read the second column of the shared memory array will write to a different image (output matrix) than the ones that read the first column. Only one image is pictured in* **(2)***.)* The threads repeat **(2)** three more times, each time processing four more columns of the shared memory array.

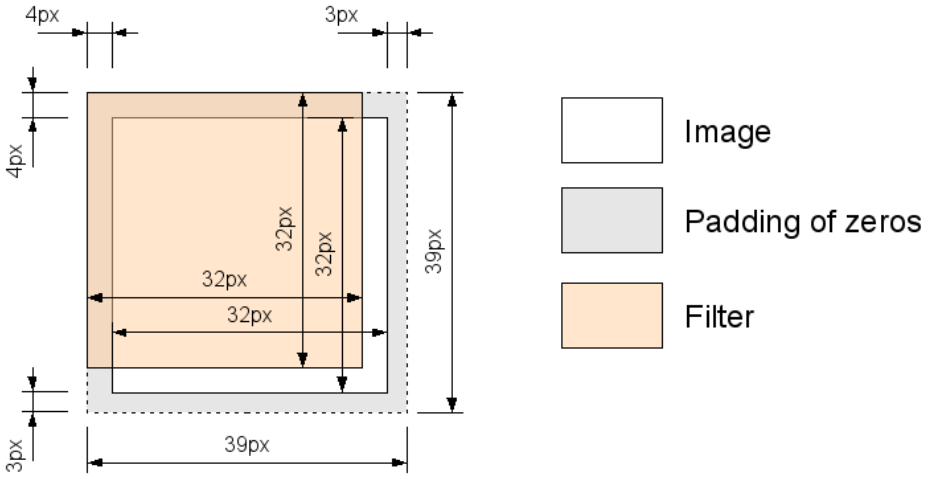

Figure 11: The convolution required to compute the derivatives with respect to the weights between the data units and the convolutional units. This convolution produces 8×8 outputs per image-filter pair. Each image must be convolved with F filters, but unlike in the convolution of step 6.2, the F filters are different for each image.

respect to w_{ik}. This amounts to the convolution depicted in Figure 11. The "filters" now are the 32×32 derivatives with respect to hidden unit inputs which we computed in the previous step.

Figure 12 describes the algorithm. This kernel is 25% faster than the convolution kernel of section 6.2 even though the kernels require an identical number of FLOPs. This kernel uses a block size of $8 \times 8 \times 8$, 5856 bytes of shared memory, and 16 registers, so it too achieves 100% occupancy. All global memory accesses are coalesced. There are no bank conflicts due to the one column of padding depicted in Figure 12. There are no warp-splitting if statements. I have checked that removing all global memory loads and their if statements makes the kernel only 5% faster. Here too I obtained a slight speedup from incrementing pointers at each loop iteration rather than recomputing indices. As can be discerned from Figure 12 (bottom), on the second iteration the kernel loads a chunk of the image matrix into shared memory, but 64% of that chunk is already in shared memory from the previous load. Despite the redundancy, it turns out that this is the fastest alternative, due to the simplicity fo the code. This kernel is 156x faster than the equivalent CPU kernel when $F = 64, N = 128$. This convolution takes 10.54ms on the GPU.

6.7 Other derivatives

Once we have the above derivatives, we need to do a simple reduction to average over all images in the batch. I won't go into detail here because it's straightforward. The bias derivatives are likewise computed by averaging the derivatives with respect to the hidden unit inputs.

7 Results

7.1 Testing methodology

I have tested each of my kernels against its CPU equivalent. Each kernel produces a matrix of results. I compared the maximum absolute difference between the matrix produced by the CPU and the matrix produced by the GPU. Observing that the maximum difference was something like 10^{-5} instilled confidence that, at the very least, the GPU code was doing an excellent job of replicating the bugs in the CPU code.

7.2 Experiments

I have trained the convolutional neural net on my dataset of 60,000 images from 10 classes; it takes about 20 minutes to fully train with 64 convolutional hidden units. It eventually learns to guess the correct class about 58% of the time on a test set. The best result I have achieved with other methods is about 65%, so the convolutional net is not doing too badly. A non-convolutional neural net with one hidden layer with 10,000 hidden units gets about 50% of test cases right.

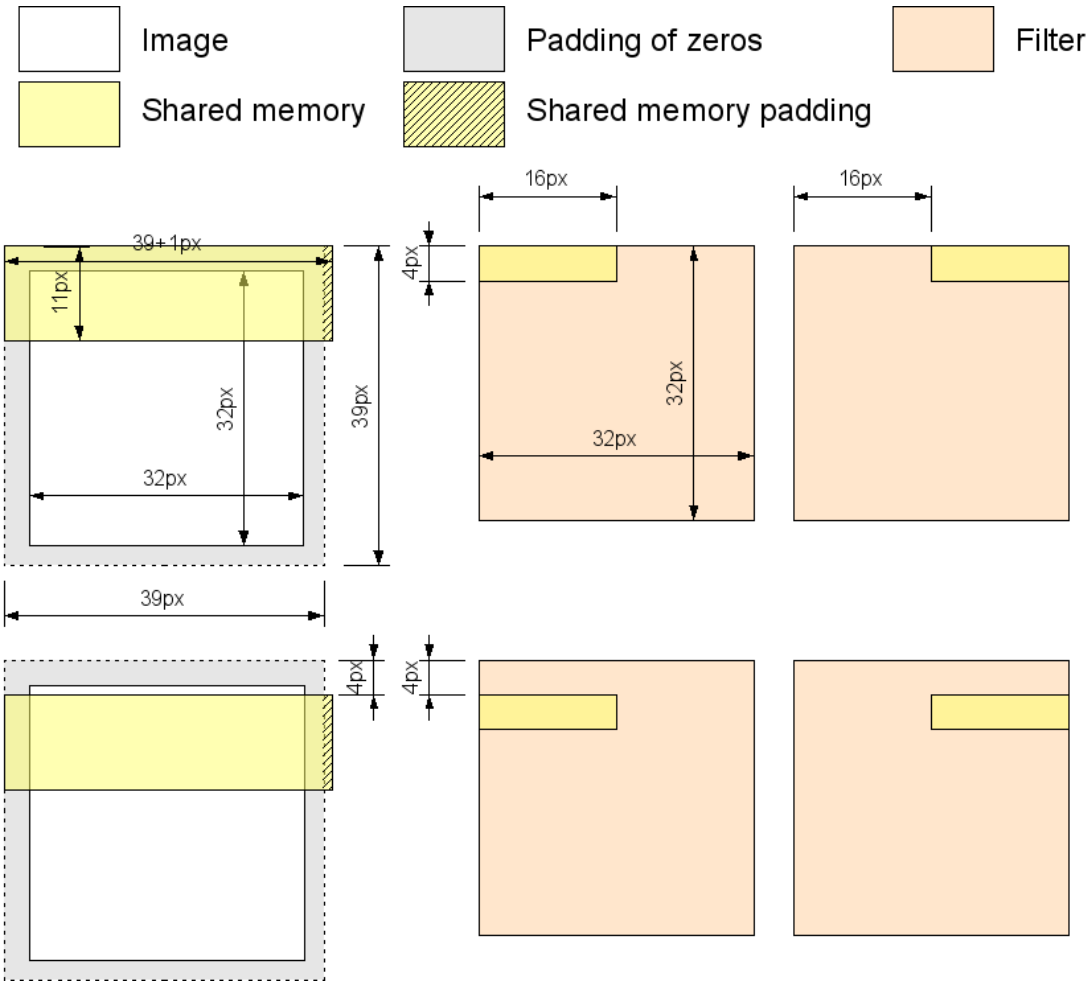

Figure 12: The algorithm for performing the convolution of Figure 11.

The block size is $8 \times 8 \times 8$. The x and y dimensions of the thread index indicate which of 64 dot products (see Figure 11) the thread is to compute. Each block convolves one image with 16 filters. The z dimension indicates which two filters the thread is concerned with. Here, I have drawn just one image and one filter. The "filters" are the 32×32 derivatives that we computed in step 6.5. The images are the input data. They are colour, but only one colour is shown. The threads process each colour in turn (it is the outer-most loop).

The threads initialize the shared memory matrix with zeros.
(Top-left) The first 11×32 of 512 threads load an 11×32 chunk of the image into shared memory.
(Top-centre) The threads load a 4×16 chunk of the filter into shared memory. Each thread computes the length-64 dot product between the filter and image that it needs to compute here.
(Top-right) The threads shift to the right by 16 pixels the shared memory window on the filter and add to the dot product from the previous window.
(Bottom) The threads shift the shared memory window on the image down by 4 pixels and repeat the above steps.
They continue shifting the windows until all chunks of the filter and image have been processed. Then they write the output to global memory and repeat all of the above for the next colour.

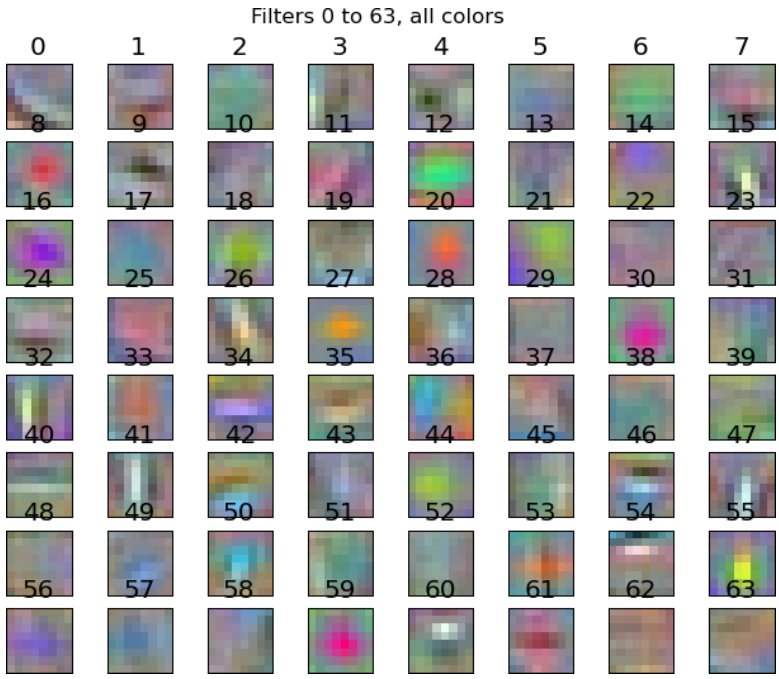

Figure 13: The 64 filters learned by the convolutional neural net. Each filter is applied by overlaying it on a 32×32 colour image and taking the dot product.

Figure 13 shows the 64 filters that this convolutional net learned. The filters have $8 \times 8 \times 3$ weights, and each weight is connected to a pixel in the image. So we can visualize these filters as if they were 8×8 colour images. I initialized the weights randomly, so this at least confirms that the network learns some sensible filters.

8 Possible improvements

I would have liked to have had more time to explore kernels in which each block convolves with multiple filters. It may be hard to maintain full occupancy in such kernels, but it may not be needed. I suspect that giving each thread more work, even at the expense of occupancy, may produce some speedups. Maybe an algorithm of the sort used for step 6.6 could work for step 6.2. The only difference would be that the threads would be able to load the entire filter into shared memory.

It may also be useful to consider alternative data representations. For example, instead of storing all the red pixel values, then all the green values, then all the blue values, I could store the red, green, and blue values for each pixel one after the other. Or, instead of storing the filters and images in row-major order, I could store one of them in column-major order. This would make it easy for one thread to compute a whole bunch of convolutions at once since it would be able to get the nth pixel of a bunch of images in just one global memory transaction.

9 Conclusion

It works! The next step is to build convolutional nets with multiple layers of convolution.

References

[1] Figure from the Wikipedia article http://en.wikipedia.org/wiki/Logistic_function