REPLY

Pete Warden's blog

Ever tried. Ever failed. No matter. Try Again. Fail again. Fail better.

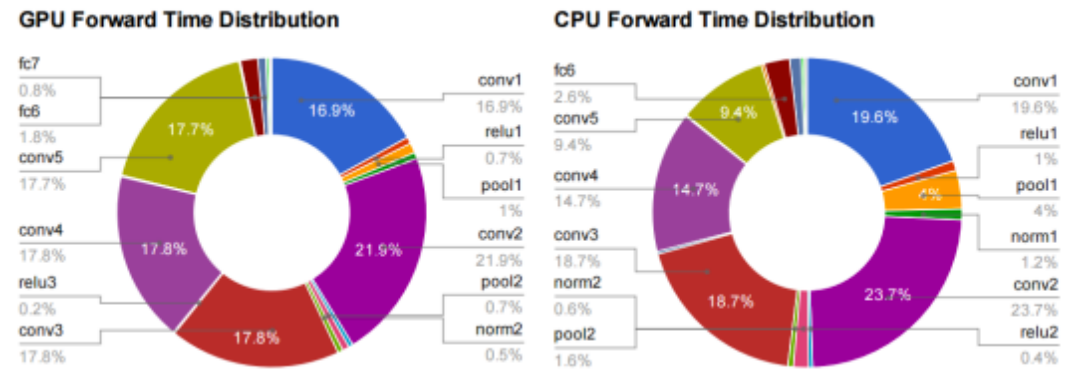
Why GEMM is at the heart of deep learning

APRIL 20, 2015 *By* Pete Warden *in* UNCATEGORIZED 20 COMMENTS

(https://www.flickr.com/photos/badwsky/2139850690/in/photolist-4g6i4q-6vNYCx-9L7GMN-dvygKf-pgwy8E-6kvYtC-5ZB71E-huyTph-47wmM6-6Eg3Fd-67Z4RC-8Q5s4X-3camqf-3cami9-huz1A1-8Q5sWk-vQndM-7SyNrD-8Q8zMf-8Q8zdb-8Q5s1P-8Q8zHj-3daZHT-8Q8z95-8Q8yWJ-doGTcK-7XSRsZ-7W3Akz-huyhr3-47wqcH-8Q8zeE-mF3TC-4uPhTY-FCz46-6kvYzJ-3owuG-68E2ei-5vzh9t-hknp7w-3iJfCq-2Rtb1G-H2vb5-huzVkt-huysN5-huyjoQ-huy8dH-6EdJkL-47wrnc-47AwqN-47AwhJ)

Photo by Anthony Catalano (https://www.flickr.com/photos/badwsky/2139850690/in/photolist-4g6i4q-6vNYCx-9L7GMN-dvygKf-pgwy8E-6kvYtC-5ZB71E-huyTph-47wmM6-6Eg3Fd-67Z4RC-8Q5s4X-3camqf-3cami9-huz1A1-8Q5sWk-vQndM-7SyNrD-8Q8zMf-8Q8zdb-8Q5s1P-8Q8zHj-3daZHT-8Q8z95-8Q8yWJ-doGTcK-7XSRsZ-7W3Akz-huyhr3-47wqcH-8Q8zeE-mF3TC-4uPhTY-FCz46-6kvYzJ-3owuG-68E2ei-5vzh9t-hknp7w-3iJfCq-2Rtb1G-H2vb5-huzVkt-huysN5-huyjoQ-huy8dH-6EdJkL-47wrnc-47AwqN-47AwhJ)

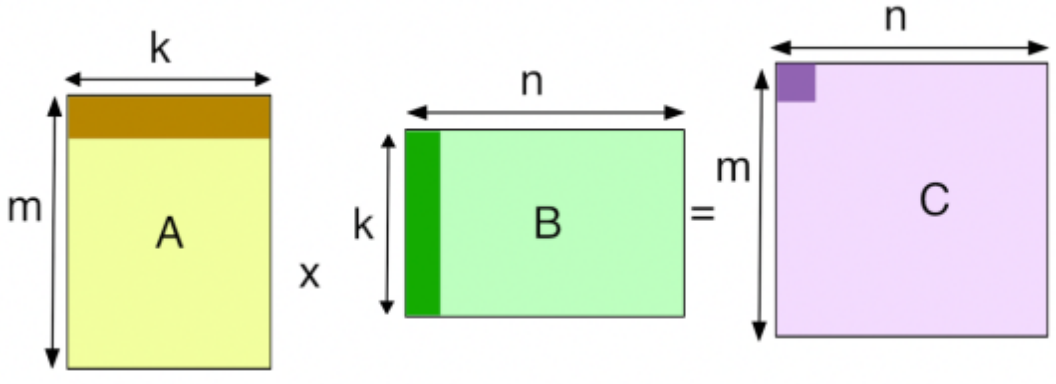
I spend most of my time worrying about how to make deep learning with neural networks faster and more power efficient. In practice that means focusing on a function called GEMM. It's part of the BLAS (Basic Linear Algebra Subprograms) library that was first created in 1979 (http://www.cs.utexas.edu/users/kincaid/blas.pdf), and until I started trying to optimize neural networks I'd never heard of it. To explain why it's so important, here's a diagram from my friend Yangqing Jia's (http://daggerfs.com/) thesis (http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-93.pdf):



(http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-93.pdf)

This is breaking down where the time's going for a typical deep convolutional neural network doing image recognition using Alex Krizhevsky's Imagenet architecture. All of the layers that start with fc (for fully-connected) or conv (for convolution) are implemented using GEMM, and almost all the time (95% of the GPU version, and 89% on CPU) is spent on those layers.

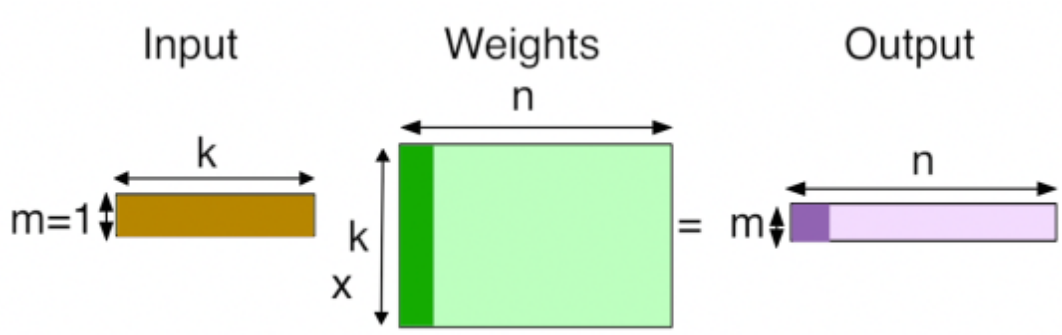
So what is GEMM? It stands for GEneral Matrix to Matrix Multiplication, and it essentially does exactly what it says on the tin, multiplies two input matrices together to get an output one. The difference between it and the kind of matrix operations I was used to in the 3D graphics world is that the matrices it works on are often very big. For example, a single layer in a typical network may require the multiplication of a 256 row, 1,152 column matrix by an 1,152 row, 192 column matrix to produce a 256 row, 192 column result. Naively, that requires 57 million (256 x 1,152, x 192) floating point operations and there can be dozens of these layers in a modern architecture, so I often see networks that need several billion FLOPs to calculate a single frame. Here's a diagram that I sketched to help me visualize how it works:



(https://petewarden.files.wordpress.com/2015/04/gemm_corrected.png)

Fully-Connected Layers

Fully-connected layers are the classic neural networks that have been around for decades, and it's probably easiest to start with how GEMM is used for those. Each output value of an FC layer looks at every value in the input layer, multiplies them all by the corresponding weight it has for that input index, and sums the results to get its output. In terms of the diagram above, it looks like this:



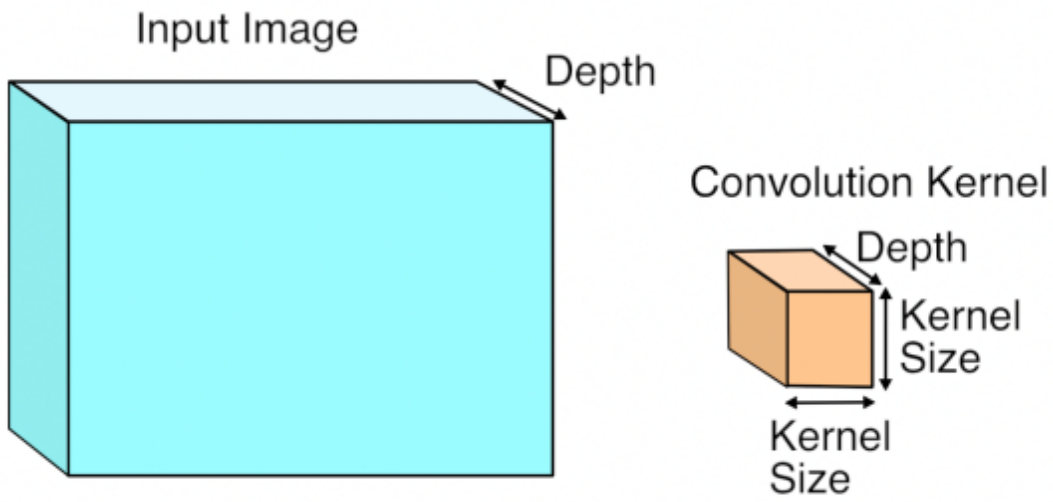
(https://petewarden.files.wordpress.com/2015/04/fcgemm_corrected.png)

There are 'k' input values, and there are 'n' neurons, each one of which has its own set of learned weights for every input value. There are 'n' output values, one for each neuron, calculated by doing a dot product of its weights and the input values.

Convolutional Layers

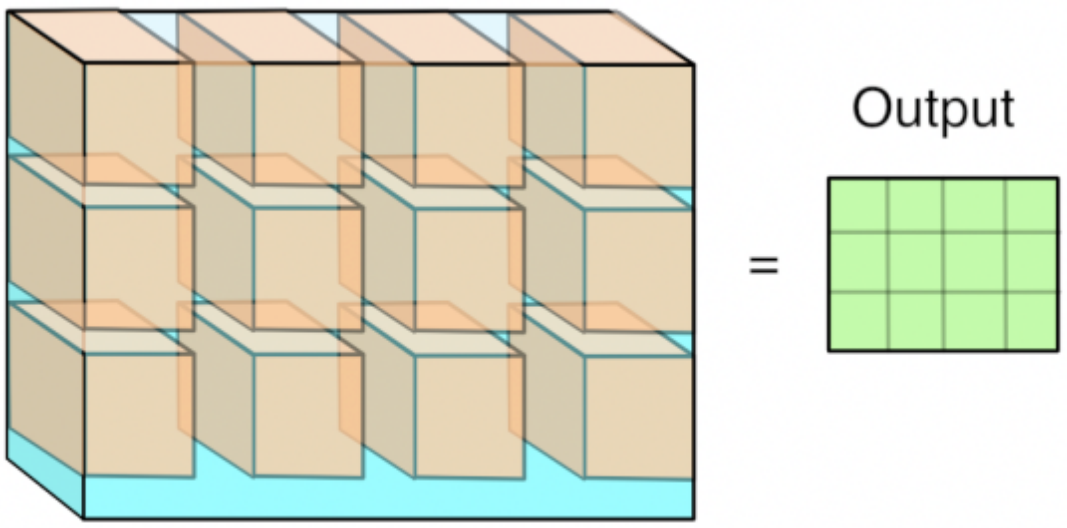
Using GEMM for the convolutional layers is a lot less of an obvious choice. A conv layer treats its input as a two dimensional image, with a number of channels for each pixel, much like a classical image with width, height, and depth. Unlike the images I was used to dealing with though, the number of channels can be in the hundreds, rather than just RGB or RGBA!

The convolution operation produces its output by taking a number of 'kernels' of weights. and applying them across the image. Here's what an input image and a single kernel look like:



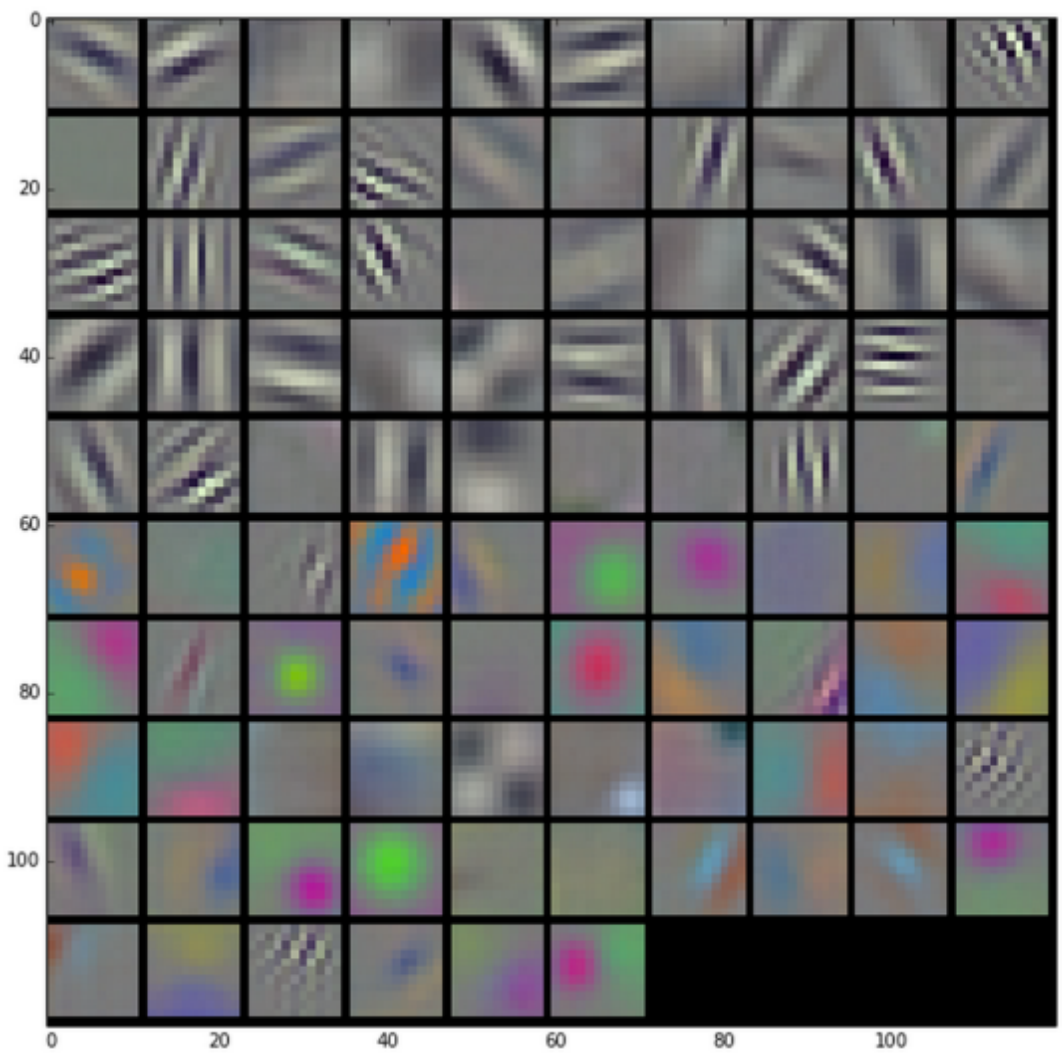
(https://petewarden.files.wordpress.com/2015/04/kernelview.png)

Each kernel is another three-dimensional array of numbers, with the depth the same as the input image, but with a much smaller width and height, typically something like 7×7. To produce a result, a kernel is applied to a grid of points across the input image. At each point where it's applied, all of the corresponding input values and weights are multiplied together, and then summed to produce a single output value at that point. Here's what that looks like visually:



(https://petewarden.files.wordpress.com/2015/04/patches1.png)

You can think of this operation as something like an edge detector. The kernel contains a pattern of weights, and when the part of the input image it's looking at has a similar pattern it outputs a high value. When the input doesn't match the pattern, the result is a low number in that position. Here are some typical patterns that are learned by the first layer of a network, courtesy of the awesome Caffe (http://caffe.berkeleyvision.org/) and featured on the NVIDIA blog (http://devblogs.nvidia.com/parallelforall/deep-learning-computer-vision-caffe-cudnn/):



(http://devblogs.nvidia.com/parallelforall/deep-learning-computer-vision-caffe-cudnn/)

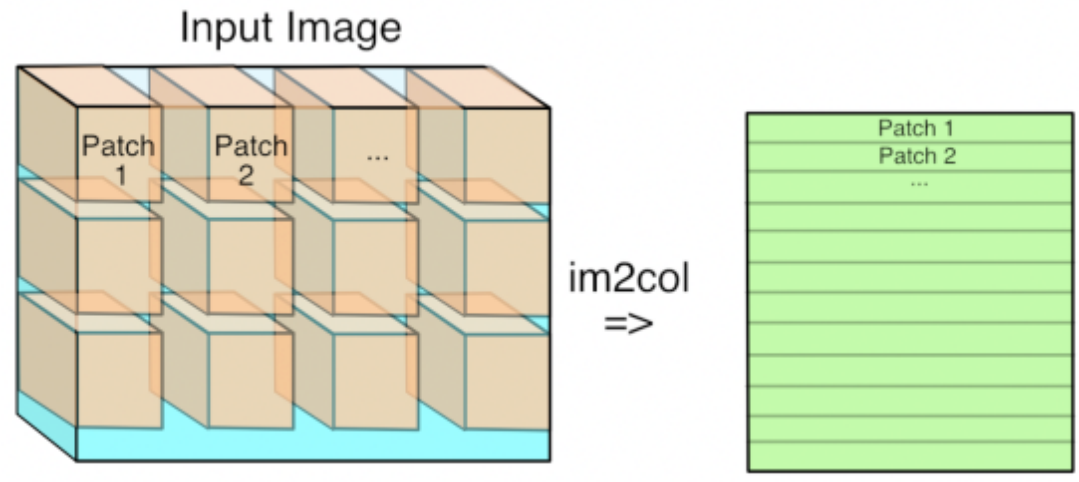
Because the input to the first layer is an RGB image, all of these kernels can be visualized as RGB too, and they show the primitive patterns that the network is looking for. Each one of these 96 kernels is applied in a grid pattern across the input, and the result is a series of 96 two-dimensional arrays, which are treated as an output image with a depth of 96 channels. If you're used to image processing operations like the Sobel operator, you can probably picture how each one of these is a bit like an edge detector optimized for different important patterns in the image, and so each channel is a map of where those patterns occur across the input.

You may have noticed that I've been vague about what kind of grid the kernels are applied in. The key controlling factor for this is a parameter called 'stride', which defines the spacing between the kernel applications. For example, with a stride of 1, a 256×256 input image would have a kernel applied at every pixel, and the output would be the same width and height as the input. With a stride of 4, that same input image would only have kernels applied every four pixels, so the output would only be 64×64. Typical stride values are less than the size of a kernel, which means that in the diagram visualizing the kernel application, a lot of them would actually overlap at the edges.

How GEMM works for Convolutions

This seems like quite a specialized operation. It involves a lot of multiplications and summing at the end, like the fully-connected layer, but it's not clear how or why we should turn this into a matrix multiplication for the GEMM. I'll talk about the motivation at the end, but here's how the operation is expressed in terms of a matrix multiplication.

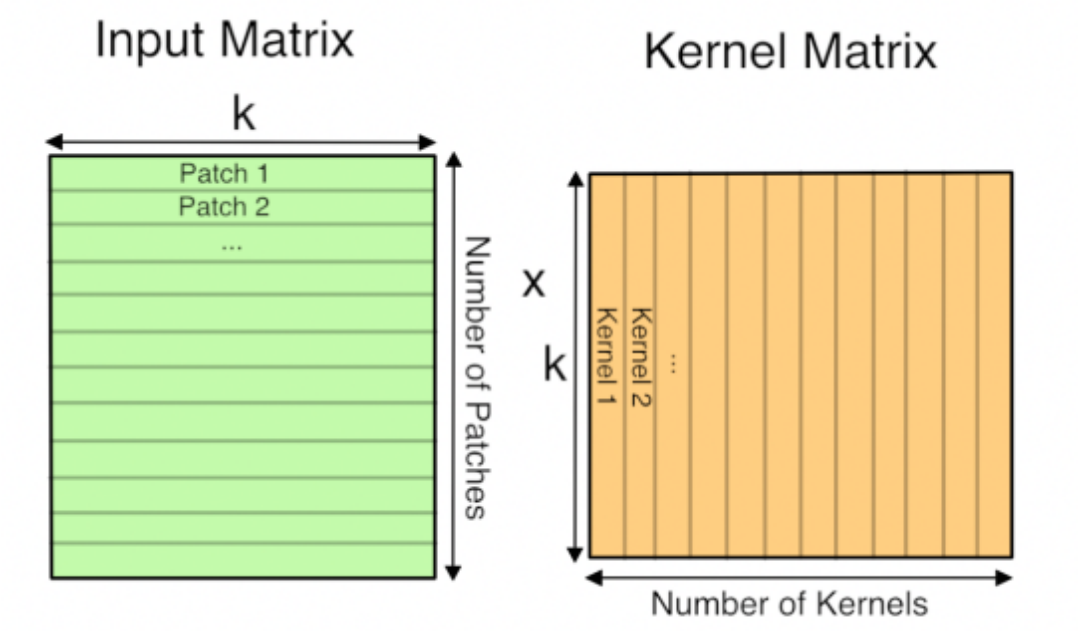
The first step is to turn the input from an image, which is effectively a 3D array, into a 2D array that we can treat like a matrix. Where each kernel is applied is a little three-dimensional cube within the image, and so we take each one of those cubes of input values and copy them out as a single column into a matrix. This is known as im2col, for image-to-column, I believe from an original Matlab function, and here's how I visualize it:



(https://petewarden.files.wordpress.com/2015/04/im2col_corrected.png)

Now if you're an image-processing geek like me, you'll probably be appalled at the expansion in memory size that happens when we do this conversion if the stride is less than the kernel size. This means that pixels that are included in overlapping kernel sites will be duplicated in the matrix, which seems inefficient. You'll have to trust me that this wastage is outweighed by the advantages though.

Now you have the input image in matrix form, you do the same for each kernel's weights, serializing the 3D cubes into rows as the second matrix for the multiplication. Here's what the final GEMM looks like:



(https://petewarden.files.wordpress.com/2015/04/im2colmult_corrected1.png)

Here 'k' is the number of values in each patch and kernel, so it's kernel width * kernel height * depth. The resulting matrix is 'Number of patches' columns high, by 'Number of kernel' rows wide. This matrix is actually treated as a 3D array by subsequent operations, by taking the number of kernels dimension as the depth, and then splitting the patches back into rows and columns based on their original position in the input image.

Why GEMM works for Convolutions

Hopefully you can now see how you can express a convolutional layer as a matrix multiplication, but it's still not obvious why you would do it. The short answer is that it turns out that the Fortran world of scientific programmers has spent decades optimizing code to perform large matrix to matrix multiplications, and the benefits from the very regular patterns of memory access outweigh the wasteful storage costs. This paper from Nvidia (http://arxiv.org/pdf/1410.0759.pdf) is a good introduction to some of the different approaches you can use, but they also describe why they ended up with a modified version of GEMM as their favored approach. There are also a lot of advantages to being able to batch up a lot of input images against the same kernels at once, and this paper on Caffe con troll (http://arxiv.org/pdf/1504.04343v1.pdf) uses those to very good

effect. The main competitor to the GEMM approach is using Fourier transforms to do the operation in frequency space, but the use of strides in our convolutions makes it hard to be as efficient.

The good news is that having a single, well-understood function taking up most of our time gives a very clear path to optimizing for speed and power usage, both with better software implementations and by tailoring the hardware to run the operation well. Because deep networks have proven to be useful for a massive range of applications across speech, NLP, and computer vision, I'm looking forward to seeing massive improvements over the next few years, much like the widespread demand for 3D games drove a revolution in GPUs by forcing a revolution in vertex and pixel processing operations.

(Updated to fix my incorrect matrix ordering in the diagrams, apologies to anyone who was confused!)

20 responses

ZYGMUNT says:
April 20, 2015 at 12:26 pm
Correct me if I'm wrong, but on the first diagram:
k x m * n x k != n x m

You need
n x k * k x m = n x m

KARTIKPODUGU says:
January 5, 2017 at 1:44 am
What you said is right. You need mxk * kxn to produce mxn. The figure also shows the same. Matrix dimentions are always (no.of.rows x no.of.columns). So, as per the first diagram, the matrix dimensions are mxk and kxn. So, you have what you need to multiply.

SCOTT GRAY says:
April 21, 2015 at 6:55 am
I thought this might be a good place to outline my approach. You can find the numbers I'm getting here (NervanaSys):

https://github.com/soumith/convnet-benchmarks

These are basically full utilization on the Maxwell GPU.

I'll use parameters defined here:

http://arxiv.org/pdf/1410.0759.pdf

So instead of thinking of convolution as a problem of one large gemm operation, it's actually much more efficient as many small gemms. To compute a large gemm on a GPU you need to break it up into many small tiles anyway. So rather than waste time duplicating your data into a large matrix, you can just start doing small gemms right away directly on the data. Let the L2 cache do the duplication for you.

So each small matrix multiply is just one position of the filter over the image. The outer dims of this MM are N and K and CRS is reduced. To load in the image data you you need to slice the image as you are carrying out the reduction of outer products. This is most easily achieved if N is the contiguous dimension of your image data. This way a single pixel offset applies to a whole row of data and can be efficiently fetched all at once. The best way to calculate that offset is to first build a small lookup table of all the spatial offsets. The channel offset can be added after each lookup. This keeps your lookup table small and easy to fit in fast shared memory.

So to manage all these small MM operations you utilize all three cuda blockIdx values. I pack the output feature map coordinates (p,q) into the blockIdx.x index. Then I can use integer division to extract the individual p,q values. Magic numbers can be computed on the host for this and passed as parameters. Then in blockIdx y and z I put the normal matrix tiling x and y coordinates. These are small matrices, but they still typically need to be tiled within this small MM operation. I use this particular assignment of blockIdx values to maximize L2 cache usage.

Lastly the kernel itself I use to compute the gemm is one designed for a large MM operation. This gives you the highest ILP and lowest bandwidth requirements. I wrote my own assembler to be able put all this custom slicing logic into a highly efficient kernel modeled after the ones found in Nvidia's cublas (though mine is in fact a bit faster). You can find the full write-up on that here:

https://github.com/NervanaSystems/maxas/wiki/SGEMM

Andrew Lavin also discovered this same approach in parallel with me, though I was about a month ahead of him and had already finished all three convolution operations by the time he released his fprop. But he does have an excellent write-up going into some additional depth. He uses constant memory instead of a shared memory lookup. This is faster but doesn't support padding. Though I understand he may now have figured that out. If so, we'll likely merge kernels at some point. Here is his write-up and source (my source will be released soon):

https://github.com/eBay/maxDNN

-Scott

Pingback: Distilled News | Data Analytics & R

CPUGUY says:
April 21, 2015 at 10:53 am
Nice paper highlighting the importance of high performance linear algebra

T says:
April 21, 2015 at 3:13 pm
The illustration of the matrix multiplication made it really clear1 Also thanks for introducing me to Caffee.

Pingback: Why are Eight Bits Enough for Deep Neural Networks? « Pete Warden's blog

Pingback: 机器学习(Machine Learning)&深度学习(Deep Learning)资料(Chapter 1)丨~ Code flavor ~

Pingback: One Weird Trick for Faster Android Multithreading « Pete Warden's blog

Pingback: An Engineer's Guide to GEMM « Pete Warden's blog

YUEYU.LIN says:
February 23, 2016 at 8:07 am
I'm really happy to see this clear introduction about GEMM and deep learning. Especially I'm strongly interested in the engineering side. From the recent updates, CUDA always is bond with the huge Nvidia graphic card with heavy loaded server mother board. I'm now trying to use Raspberry Pi to build a low engergy deep learning cluster. The most recent RPi 2 B is about only several watts each and event 10 RPi is less than 100 watts, which has 40 cores and 40 Giga bytes rams. What's more, each RPi 2 B has a decent broadcom GPU, I expect I can use it to build a very low energy deep learning cluster.
Actually I found your blog about how to use RPi to accelerate the deep learning process. Do you think it's worth to make Caffe to run in RPi which is heavily relying on CUDA? I'm looking forward to your very insightful opinions. If you think it's worth to port Caffe to RPi, I'll start to dig the treasure!

Pingback: 机器学习(Machine Learning)&深度学习(Deep Learning)资料(Chapter 1) – 软件启示录

MOUSTAFA says:
June 16, 2016 at 6:40 am
Great post ! Thanks

STEVEN YU says:
August 11, 2016 at 7:23 am
Thank you for your great post! Now I understand how GEMM works for convolution.

Pingback: 机器学习(Machine Learning)&深度学习(Deep Learning)资料丨Dotte博客

YUNYU says:
September 11, 2016 at 8:00 am
Thank you for this post!

Pingback: AMD gets into Machine Intelligence with "MI" range of hardware and software - StreamComputing

Pingback: Why Deep Learning Needs Assembler Hackers « Pete Warden's blog

ATCOLD says:
January 9, 2017 at 7:22 pm
"with a stride of 1, a 256×256 input image would have a kernel applied at every pixel, and the output would be the same width and height as the input" hmm… not quite. This is true iff you use padding of (kernel_size – 1) / 2. Otherwise your output will be input_size – kernel_size + 1, squared.

DENDISUHUBDY says:
February 11, 2017 at 3:51 am
Reblogged this on The Secret Guild of Silicon Valley.