A High-Performance Implementation of ABC-Nets in Flux.jl

James Gilles

Abstract

We implement and evaluate a high-performance implementation of the ABCNets quantization algorithm in Julia with with Flux.jl and CUDAnative.jl.

10 December 2019

1 Introduction and Related Work

In the past decade, deep neural networks have achieved rapid uptake in many fields and applications. However, deep neural networks require large amounts of memory and compute to execute. This prevents them from being deployed in resource-constrained environments like mobile devices (Howard et al., 2017). In addition, network training is expensive; full development of a new deep network architecture has been estimated to have an equivalent carbon footprint to the lifetime footprint of five consumer cars (Hao, 2019).

To address this issue, researchers are looking into techniques for **model compression**. Model compression attempts to reduce the amount of memory or computation needed to evaluate a model. Typically this imposes some loss of accuracy or performance; the challenge is to reduce resource requirements while maintaining performance. Model compression is an old approach, stretching far back into the literature (LeCun et al., 1990). For a survey of the field, see (Cheng et al., 2017)

In this paper we examine a specific style of model compression, **model quantization**. Deep neural networks are typically trained and evaluated using IEEE 754 32-bit floating point numbers (IEE, 2008). 32-bit floats have a large dynamic range and enough precision for general-purpose computing. However, in practice, the weights and activations of deep neural networks have been found to fall into small, predictable distributions. This suggests that they require less than 32 bits of information to accurately represent. This insight leads to the field of model quantization, which investigates different representations for model weights and activations.

Simple quantization techniques substitute floating point numbers with less range, such as 16-bit IEEE 754 floats, or Google's bfloat16 (Tagliavini et al., 2017). More aggressive techniques quantize values to 8, 4, 2, or even single bits, and may use floating-point or fixed-point numbers. (Quantizing to a single bit is termed "binarization", and reflects the observation that artificial neurons can be essentially "on" or "off".)

Research in training networks at these high degrees of quantization is ongoing. However, they all share a problem: they require hardware support to be reasonably efficient. The long development times required for hardware makes evaluating their performance difficult; and the need for that hardware at inference time makes deploying them even more difficult.

It was therefore very exciting when, a few years ago, Courbariaux et al. (2016) introduced a novel quantization technique that can be implemented without custom hardware. Their technique requires binarizing network weights and activations to the values ± 1. They then pack the binary weights and activations into machine words, representing -1 with `0` bits. They introduce a novel algorithm (described in the next section) based on the `xnor` and `popcnt` machine instructions, which allows taking the dot product of a pair of packed vectors in less than 5 machine instructions. This greatly

reduces the necessary compute and memory needed to evaluate binarized networks. Unfortunately, it also imposes significant accuracy losses; it's not obvious how to train binarized networks. Several papers (such as the XNORNets paper from Rastegari et al. (2016)) proposed improved training schemes. These reduce the error due to binarization, but still leave large gaps.

The paper this work implements (Lin et al. (2017)) introduced "ABCNets" (Accurate Binary Convolutional Networks), which allow using multiple bits to represent weights and activations, while still using the efficient `xnor/popcnt` algorithm for convolution. In theory, this allows a clean trade-off between accuracy loss and runtime performance:

Quantization	Weight bits	Activation bits	Top-1	Top-5
Full-Precision	32	32	69.3%	89.2%
BNNs	1	1	42.2%	67.1%
XNORNets	1	1	51.2%	73.2%
ABCNets	5	5	**65.0%**	**85.9%**

(Results are from Lin et al. (2017), evaluating the Resnet18 model on the ImageNet dataset.)

However, Lin et al. (2017) did not actually implement their convolution algorithm using packed machine words. Instead, they used full-precision floating-point numbers and applied equivalent quantization algorithms in floating-point space. This should give nearly identical results, modulo 32-bit floating point precision. However, it does not tell us whether ABCNets are actually faster than full-precision networks on real hardware. It's hard to make the decision about whether to quantize without knowing the resulting accuracy improvement.

In this paper, we write an implementation of the ABCNets algorithm for GPU, using the high-performance scientific programming language Julia and its extensions to support the CUDA programming model.

1.1 The BNN algorithm

Courbariaux et al. (2016) introduced an efficient algorithm for evaluation of Binarized Neural Networks (or BNNs). This technique requires binarizing network weights and activations to the values $+1$ and -1.

They observe that vectors of ± 1 can be packed into machine words by representing $+1$ with 1s and -1 with 0. Given a pair of vectors $\mathbf{a}, \mathbf{b} \in \{-1, +1\}^{64}$, and their packed equivalents `packed_a` and `packed_b`, the dot product of the vectors ($\mathbf{a}^\top \mathbf{b}$) can be computed as follows:

```
dot_packed(packed_a :: UInt64, packed_b :: UInt64) :: UInt64 =
    2 * popcount(not(xor(a, b))) - bitcount(UInt64)
```

Where:

- `popcount` is the standard "population count" machine instruction, which returns the number of 1 bits in a word
- `bitcount` returns the number of bits in a datatype,
- `not` and `xor` operate bitwise.

Why does this work? First, observe the equivalence between multiplication of $(\pm 1) * (\pm 1)$ and `not(xor(0/1, 0/1))`:

$(\pm 1) * (\pm 1)$	`not(xor(0/1, 0/1))`
$-1 * -1 = +1$	`not(xor(0, 0)) = 1`
$-1 * +1 = -1$	`not(xor(0, 1)) = 0`
$+1 * -1 = -1$	`not(xor(1, 0)) = 0`
$+1 * +1 = +1$	`not(xor(1, 1)) = 1`

For a pair of bits, `not(xor(a, b))` performs the equivalent of multiplication. Thus, across a pair of machine words, `not(xor(packed_a, packed_b))` is the equivalent of elementwise multiplication, i.e. the Hadamard product $\mathbf{a} \odot \mathbf{b}$.

To compute the dot product from the Hadamard product, we simply need to sum its elements:

$$\mathbf{a}^\top \mathbf{b} = \sum_{i=1}^{64} (\mathbf{a} \odot \mathbf{b})_i$$

For the sake of example, assume we're using 8-bit integers instead of 64. If we have:

$$\mathbf{a} \odot \mathbf{b} = \begin{bmatrix} +1 \\ -1 \\ -1 \\ +1 \\ +1 \\ -1 \\ +1 \\ +1 \end{bmatrix}$$

Then

$$\mathbf{a}^\top \mathbf{b} = +1 - 1 - 1 + 1 + 1 - 1 + 1 + 1 = +2$$

Addition is commutative, so we can rearrange this sum to split up the positive and negative elements:

$$\mathbf{a}^\top \mathbf{b} = \underbrace{+1 + 1 + 1 + 1 + 1}_{\text{positive elements}} \underbrace{-1 - 1 - 1}_{\text{negative elements}} = +2$$

Now, assume we have $\text{count}_{(+1)}$, the count of the positive elements, and $\text{count}_{(-1)}$, the count of the negative elements. The above sum becomes:

$$\begin{aligned} \mathbf{a}^\top \mathbf{b} &= (+1) * \text{count}_{(+1)} + (-1) * \text{count}_{(-1)} \\ &= \text{count}_{(+1)} - \text{count}_{(-1)} \end{aligned}$$

This is where `popcnt` comes from. If `packed_prod = not(xor(packed_a, packed_b))`, then:

$$\begin{aligned} \text{count}_{(+1)} &= \texttt{popcnt(packed\_prod)} \\ \text{count}_{(-1)} &= \texttt{bitcnt(packed\_prod) - popcnt(packed\_prod)} \end{aligned}$$

Putting it all together:

$$\begin{aligned}\mathbf{a}^\top \mathbf{b} &= \sum \mathbf{a} \odot \mathbf{b} \\ &= \text{count}_{(+1)} - \text{count}_{(-1)} \\ &= \texttt{popcnt(packed\_prod) - (bitcnt(packed\_prod) - popcnt(packed\_prod))} \\ &= \texttt{2 * popcnt(packed\_prod) - bitcnt(packed\_prod)}\end{aligned}$$

This allows us to compute the dot product of up to 64 elements at a time using only 3 machine instructions.

This can be extended to vectors that don't fit into machine words. Simply pack the bits into a sequence of machine words, padding with zero at the end. Then replace `bitcnt(packed_prod)` in the above equation with the element count of your vector.

This dot-product primitive can be extended to implement matrix multiplication and convolution/cross-correlation in the straightforward way.

(Note: From the above example, you can see that the output of this product of binary weights and activations is **not** binary: it must be re-quantized if we want to perform the same operation on its result. This will be important later.)

1.2 The ABCNets algorithm

Lin et al. (2017) introduced the ABCNets algorithm, which represents weights and activations using **linear combinations of binary weight bases**. The underlying observation is fairly straightforward.

Looking at matrix multiplication – which can be generalized to batched convolution – we approximate weights and activations as follows:

$$\mathbf{W} \approx \widetilde{\mathbf{W}} = \alpha_1 \widetilde{\mathbf{W}}_1 + \alpha_2 \widetilde{\mathbf{W}}_2 + ... + \alpha_M \widetilde{\mathbf{W}}_M = \sum_i^M \alpha_i \widetilde{\mathbf{W}}_i$$

$$\mathbf{A} \approx \widetilde{\mathbf{A}} = \beta_1 \widetilde{\mathbf{A}}_1 + \beta_2 \widetilde{\mathbf{A}}_2 + ... + \beta_N \widetilde{\mathbf{A}}_N = \sum_j^N \beta_j \widetilde{\mathbf{A}}_j$$

Where $\widetilde{\mathbf{W}}_i$ and $\widetilde{\mathbf{A}}_j$ are binarized versions of weights and activations, learned during training. (The algorithms to select the weights and activations are described in the following subsections.) Then, we have:

$$\mathbf{W}\mathbf{A} \approx \widetilde{\mathbf{W}}\widetilde{\mathbf{A}} = (\sum_i^M \alpha_i \widetilde{\mathbf{W}}_i)(\sum_j^N \beta_j \widetilde{\mathbf{A}}_j) = \sum_i^M \sum_j^N \alpha_i \beta_j \widetilde{\mathbf{W}}_i \widetilde{\mathbf{A}}_j$$

That is, the product of the approximated weights and activations can be computed in $M * N$ XNOR-matrix-multiplications, where M and N are the bit-widths of weights and activations respectively.

(This is the algorithm we implement in Julia / CUDA, extended to convolution.)

1.2.1 Weight Approximation

The problem of approximating the weights is finding the weight masks $\widetilde{\mathbf{W}}_i$ and coefficients α_i for $i \in 1..M$.

Lin et al. (2017) suggest keeping full-precision weights around during training, and finding weight masks such that the mean squared error between the approximated weights and the real weights is minimized:

$$\alpha_i, \widetilde{\mathbf{W}}_i = \arg\min_{\alpha_i, \widetilde{\mathbf{W}}_i} \left\| \mathbf{W} - \sum_i \alpha_i \widetilde{\mathbf{W}}_i \right\|^2 \tag{1}$$

This could be trained through gradient descent. However, once the network is trained, the weights are fixed; so the authors suggest using a training algorithm that is more expensive during training but less expensive during evaluation. They fix the weight masks:

$$\widetilde{\mathbf{W}}_i = F_{u_i}(\mathbf{W}) = \text{sign}(\mathbf{W} - \text{mean}(\mathbf{W}) + u_i \,\text{std}(\mathbf{W}))$$

Where u_i is a fixed factor chosen a-priori. Then, (1) becomes a system of overdetermined linear equations, which can be solved using least-squares. They repeat this operation during each training step, and use the masks from the final training step for the final network.

It is not obvious how to propagate gradients through this quantization operation, since it is a piecewise constant function without meaningful derivatives. Following Courbariaux et al. (2016), Lin et al. (2017) use the "straight-through estimator":

$$\overline{\mathbf{W}} = \overline{F_{u_i}(\mathbf{W})} = \overline{\widetilde{\mathbf{W}}}$$

Where $\overline{\mathbf{x}}$ denotes the gradient of the loss with respect to $\mathbf{x}$. That is, they pass the gradient of the approximated weights "straight through" to the real weights. This approximation lacks theoretical backing but works well in practice.

1.2.2 Activation Approximation

The problem of approximating the activations is finding the activation masks $\widetilde{\mathbf{A}}_j$ and coefficients β_j for $j \in 1..N$. Running a least-squares solver for every network evaluation would be too slow, so Lin et al. (2017) define a simple quantization operation:

$$\text{actbin}_v(x) : \mathbb{R} \to \{-1, 1\} = \begin{cases} +1 & \text{clip}(x + v, 0, 1) > 0.5 \\ -1 & \text{otherwise} \end{cases}$$

Then, for each basis, we create a separate learned parameter v_i. The basis is then this operation applied to all elements of the input, with that parameter:

$$\widetilde{\mathbf{A}}_i = \text{actbin}_{v_i}.(\mathbf{A})$$

The parameters v_i and the coefficients β_i in the sum are then learned during training using gradient descent.

The gradient through this quantization (for any single activation) is:

$$\overline{x} = \overline{v} = \overline{\text{actbin}_v(x)} * \begin{cases} 1 & x + v \in [0, 1] \\ 0 & \text{otherwise} \end{cases}$$

Which is equivalent to STE when the input is not clipped, and 0 when the input is clipped. (This choice is motivated further in the original paper.)

1.3 Julia

The Julia programming language (Bezanson et al., 2017) is a programming language designed for elegant, high-performance scientific computing. In this project, in addition to the base Julia language, we leverage the CuArrays.jl, CUDAnative.jl (Besard et al., 2019), and Flux.jl (Innes, 2018) libraries.

CUDAnative is an interface which allows Julia code to be compiled to run on the GPU, similar to CUDA C++. CuArrays is a library for GPU-based array operations on top of CUDAnative. Flux is a machine learning framework that builds on that using a source-to-source automatic differentiation system, allowing operations that run across the GPU and CPU to be differentiated with little overhead beyond the actual computation needed for differentiation.

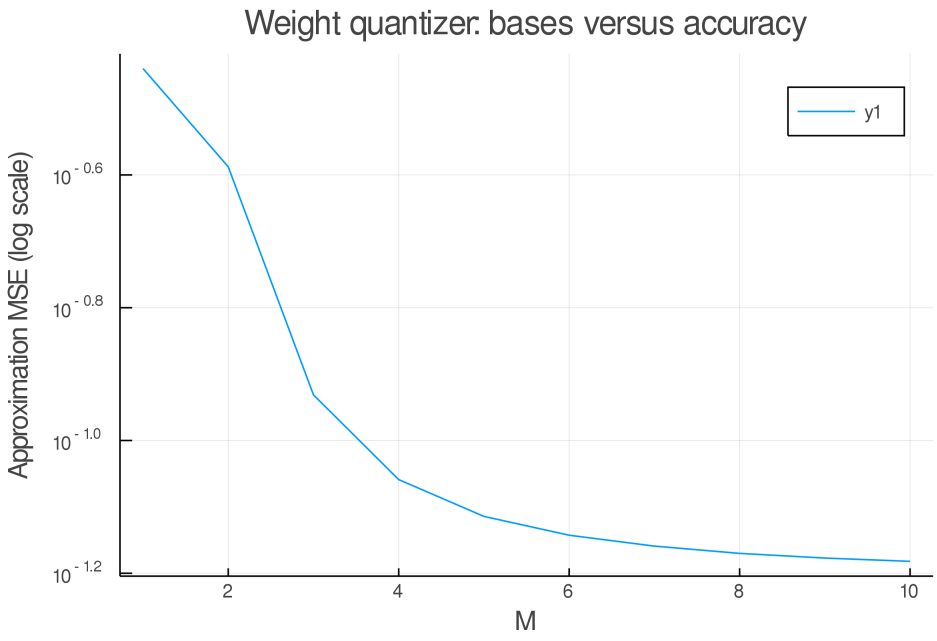
We implement the ABCNets paper using a mix of Flux, CuArrays, and CUDAnative.

2 Implementation

2.1 Floating Point

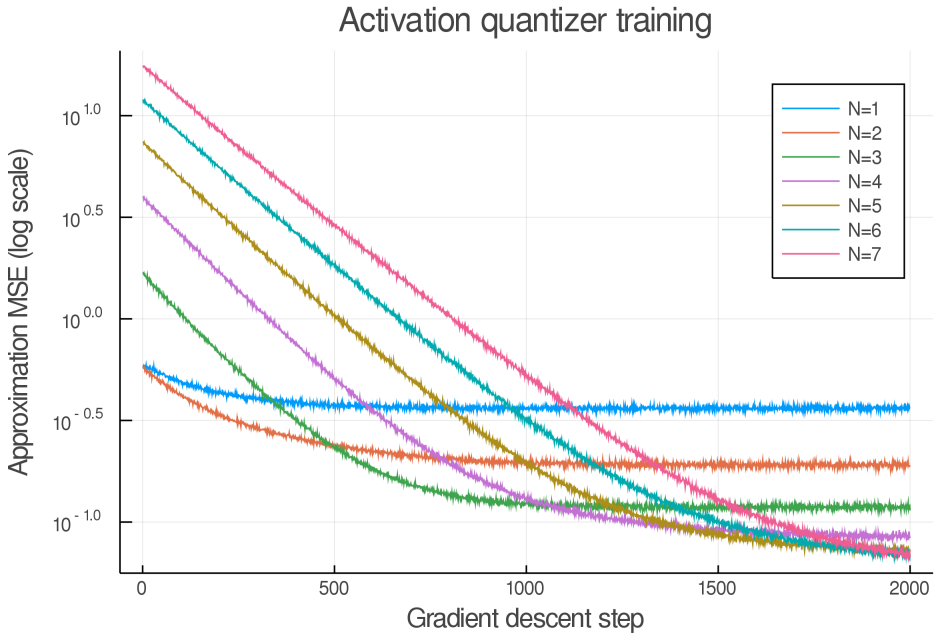
To make sure we understood the source paper and to have a baseline for comparison, we first implemented the ABCNets paper using floating point numbers. We defined the underlying algorithms using CuArrays, and then defined Flux layers for the weight quantization and activation algorithms, as well as their adjoints.

To test the correctness of the weight quantization, we measured the MSE between quantized and unquantized weights for a given number of weight bases. The u_i parameters of the original weight quantization must be chosen a priori; inspired by the original paper, we simply used M values evenly spaced between -1.0 and 1.0 (using only 0.0 in the $M = 1$ case.) We generated a normally distributed 10x10x10x10 block of weights. Given the true weights and u_i, the α parameters (and therefore the quantization error) are fully determined:



We see that the error lowers quickly as M increases, and then plateaus. It's possible that the error could be decreased further by training the u_i parameters; that's a direction for future work.

The activation quantization must be trained through gradient descent. To test this, we train the quantization function on blocks of 10x10x10x10 normally distributed random activations, minimizing the MSE relative to the true activations using the Adam optimizer (Kingma & Ba, 2014). In addition, we vary the number of bases available (N). (Note that each gradient descent step gives a **different** block of activations to approximate, with the same distribution parameters. This models varying activations during evaluation.)



We see that gradient descent is quite good at finding values for the activation quantization parameters. It's non-obvious that gradient should work in this case, when using the straight-through estimator, but it does.

(Do note that there isn't any MSE term in the actual loss during training; instead, the quantization is trained only to minimize the overall loss of the network. It's an open question whether adding other terms might improve performance.)

Finally, we implemented a floating point `ABCCrossCor` layer, which encapsulates quantizing weights and activations and then convolving (in floating point). Note that the operation called convolution in most neural network frameworks is actually cross correlation, since it doesn't flip the kernel along the X and Y axis. Flux bucks this trend by naming things correctly, and we follow suit.

We built a simple network using this layer and trained it on the FashionMNIST (Xiao et al., 2017) dataset, which is a dataset somewhere between MNIST and CIFAR in difficulty. The network is described as follows:

```
N = 3
M = 3

model = Chain(
    CrossCor((3, 3), 1=>16, pad=(1,1)),
    BatchNorm(16), Relu(), ABCCrossCor((3, 3), 16=>32, N, M, pad=(1,1)),
    BatchNorm(32), Relu(), ABCCrossCor((3, 3), 32=>32, N, M, pad=(1,1)),
    MaxPool((2,2)),
    BatchNorm(32), Relu(), ABCCrossCor((3, 3), 32=>32, N, M, pad=(1,1)),
```

```
    BatchNorm(32), Relu(), ABCCrossCor((3, 3), 32=>64, N, M, pad=(1,1)),
    BatchNorm(64), Relu(), ABCCrossCor((3, 3), 64=>64, N, M, pad=(1,1)),
    MaxPool((2,2)),
    BatchNorm(64), Relu(), ABCCrossCor((3, 3), 64=>64, N, M, pad=(1,1)),
    BatchNorm(64), Relu(), ABCCrossCor((3, 3), 64=>64, N, M, pad=(1,1)),
    BatchNorm(64), Relu(), ABCCrossCor((3, 3), 64=>64, N, M, pad=(1,1)),
    MaxPool((2,2)),
    x -> reshape(x, :, size(x, 4)),
    Relu(),
    Dense(64 * 3 * 3, 10),
    softmax,
)
```

Note that the first convolution is **not** quantized, which is standard in quantization studies. Additionally, we perform the BatchNorm + ReLU + quantize operation after max pooling, rather than before, on the recommendation of the original paper.

At full precision, this network achieves 93.1% top-1 accuracy on the FashionMNIST dataset (when trained with the Adam optimizer using Flux default settings, input values in the range $(0, 1)$, and no data augmentation). When quantized and fine-tuned, the network achieves 91.0% top-when accuracy, with M=N=3 bases for weights and activations.

2.2 Packed Binary Operations

The bulk of the work was spent implementing binary packing and convolution kernels using CUDAnative. CUDAnative imitates the CUDA C++ API. When implementing a CUDA program, the challenge is generally to achieve full utilization of all of the compute resources and memory bandwidth available on the device. There are many choices to be made, including how to map an input space to CUDA threads and CUDA blocks, how to coordinate between threads, and how to take advantage of shared memory.

First we implemented a quantization and packing kernel. This kernel is responsible for walking a `Float32` input tensor, applying a quantization operation to each input, and packing the resulting bits into a `UInt32` output tensor. We chose to pack into 32 bits instead of 64 in order to avoid **memory bank conflicts**, which will be discussed shortly. The quantization operation is user-supplied and fused into the kernel, which is easy to do because of how Julia and CUDAnative work.

This kernel also transposes the input. All Flux operations for image processing keep their inputs in WHCB order: that is, x, y, channel, batch, with x walked first because Julia stores arrays in Fortran order. However, to ensure that results fit well into machine words we reordered this to PWHB in the kernel (packed channel, x, y, batch). In the case of multiple bases, the kernel splats the input out to a 5-dimensional array, PWHNB (packed channel, x, y, basis, batch). (Keeping track of all these dimensions can be confusing. To avoid mistakes, we named all functions and variables based on their dimensions; for instance, the helper function that dispatches our quantization kernel is called `quant_pack_WHCB_PWHNB_32`.)

Bit-packing would seem to require communication between threads; that's slow. However, we were able to leverage the CUDA `vote_ballot` intrinsic to make it extremely fast. CUDA thread blocks execute in "warps" of 32 threads; threads are grouped into warps by adjacent thread index. The `vote_ballot` intrinsic takes a boolean and returns an integer with bits set for every element in a warp. If the thread passed in `true`, the bit is 1; if the thread passed in `false`, the bit is 0. By carefully arranging the threads within our kernel, we ensure that warps line up with the regions of the input tensor that map to packed `UInt32s` in the output tensor. Then we used `vote_ballot` to perform the required bit-packing in a single operation.

The performance of our quantization kernel is entirely bounded by global memory bandwidth. Within the kernel, there's very little communication or computation; just invocation of the quantization operation and a single `vote_ballot` per output. The naive implementation turned out quite fast, so we didn't spend much time optimizing it.

We also implemented an unpacking kernel to check correctness of the packing kernel. We didn't optimize the unpacking kernel, though, since it isn't needed during model training or evaluation.

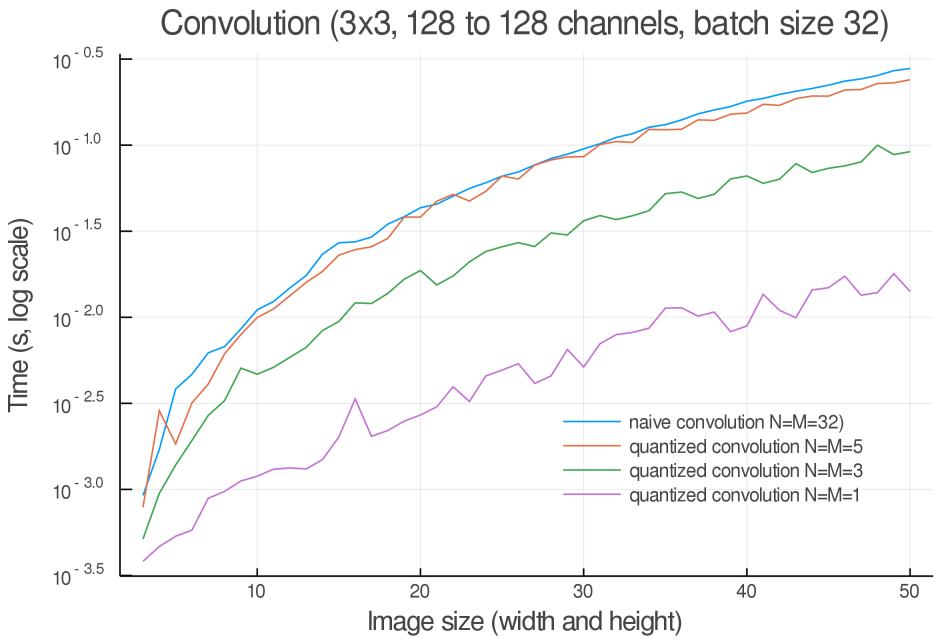
Finally, we implemented a binary "convolution" kernel. This kernel performs convolution (actually cross-correlation) between multiple bases of weights and activations, multiplies the results by the corresponding coefficients, and writes its output to a `Float32` tensor. To start, we implemented this as simply as possible. To verify correctness, we compared the results against CUDNN. CUDNN is an library of CUDA-accelerated neural network kernels from NVIDIA; it serves as Flux's built-in implementation for convolution. We also implemented a naive floating-point convolution kernel, arranged in the same structure as the binary kernel, to serve as a comparison.

This kernel was more interesting to optimize, since there was room for re-using intermediate memory accesses. CUDA provides a programmer-controlled shared memory space to each thread block, which behaves similarly to L1 cache. Our kernel was arranged so that all the threads in a thread block would re-use the same region of the input tensor. We therefore extended our kernel by having all the threads in each block start by collaboratively reading the relevant input region to the shared cache.

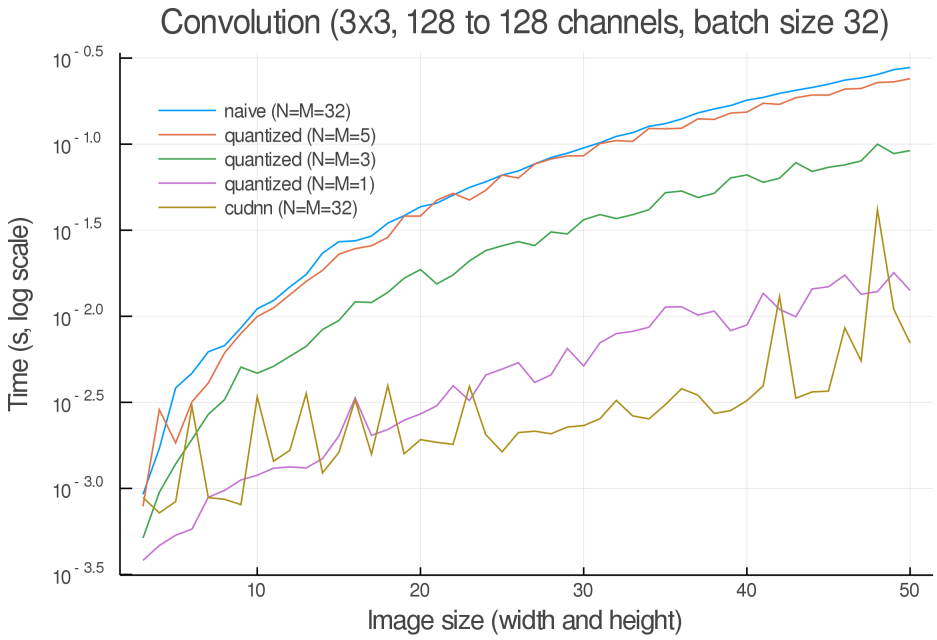
After that, we modified the iteration pattern of different threads to avoid bank conflicts. The shared memory cache has 32 access ports, arranged such that all the threads in a warp access adjacent 32-bit values, all the threads can read from separate ports. However, if some threads access the same value, those accesses must be serialized. We arranged our threads using what we termed a "barber pole pattern", where each thread started its iteration on a separate input region, and would remain mostly non-overlapping throughout.

2.3 Benchmarks

We benchmarked our kernels across a range of image sizes. Here's a comparison of our naive (floating-point) convolution kernel, and the corresponding quantized convolution kernel:



This seems promising: lower precision quantization gives significant speedups. (Note the logarithmic time scale!) Unfortunately, it isn't the whole story. This is what happens when we add 32-bit CUDNN to the mix:



CUDNN beats even the binary version of our kernel! Why is that? Well, partially, CUDNN is just an extremely well-optimized library; but more importantly, we were **unable to use CUDA shared memory**. We ran into issues with our technical stack that prevented shared memory from providing any performance benefit, and were unable to fix these issues in time, despite days of debugging. In the end, we were forced to rip out the optimizations managing shared memory and iteration performance.

Still, we've shown that this algorithm can provide significant speedups over full precision. We hypothesize that we can wring major additional speedups out of the code once the issues preventing the use of shared memory are fixed; but for now, leave that to future work.

3 Julia, CUDAnative.jl and Flux.jl: Benefits and Drawbacks

In this section we reflect on how using Julia, CUDAnative, and Flux.

Some aspects of development were smooth and easy. Flux and CuArrays in particular provide an extremely elegant interface. For example, the floating-point implementation of equation (1) can be essentially copied from the equation listing, modulo some rearranging of axes:

```
function weight_masks(W, us)
    dims = size(W)
    W̄ = W .- mean(W)
    std_W = std(W)
    W̃s = cat((sign.(W̄ .+ (u * std_W)) for u in us)...,
             dims=length(size(W)) + 1)
    W̃s
end
function binarize_weights(W, us)
```

```
    W̃s = weight_masks(W, us) # compute the masks of the weight tensor, given the u_i
    ↪  constants
    dims = size(W)
    Wv, W̃vs = reshape(W, :), reshape(W̃s, :, length(us)) # flatten weights and masks
    αs = W̃vs \ Wv # least squares
    W̃v = W̃vs * αs # compute result
    W̃ = reshape(W̃v, dims...) # reshape output
    # return (floating-point) approximated weights,
    # and the discovered optimal coefficients
    W̃, αs
end
Zygote.@adjoint function binarize_weights(W, us)
    W̃ = binarize_weights(W, us)
    # note: ∇_W̃ should be read "gradient with respect to W̃"
    function adjoint((∇_W̃, ∇_αs))
        # the straight-through estimator.
        # (us are not trainable.)
        (∇_W, ∇_us) = (∇_W̃, nothing)
        (∇_W, ∇_us)
    end
    W̃, adjoint
end
```

Implementing the floating-point version was easy and resulted in very neat code, probably the most "mathematical" machine learning code the author has ever written. (Mathematical to a fault, in fact. One funny bug we found was that our convolution code only gave the same results as Flux when we flipped the kernel. This turned out to be because the Flux `Conv` layer performs **actual** convolution; we were looking for the operator wrongly referred to as convolution in most of the machine learning literature, which Flux correctly terms `CrossCor`).

CUDAnative was unfortunately significantly harder to work with. The primary issue was that some error in the tech stack caused a segfault when our code was run under `nvprof` and `nsys` (NVIDIA's tools for profiling CUDA code). It's not clear whether the issue is in the hardware we used, the OS, the driver, `nvprof`, or Julia itself. Despite several days of debugging, we were unable to resolve it. This meant that we were unable to acquire anything more than basic timing information for our kernels. This made optimizing the kernels extremely difficult, since we weren't sure where the bottlenecks were.

In addition, CUDAnative only supports a subset of Julia. For example, any code which calls string formatting implicitly relies on the Julia runtime, which CUDAnative will reject. This wouldn't seem like a huge problem, except that many error paths rely on string formatting to print errors. This means that a lot of seemingly innocuous code will cause CUDAnative to fail. This issue is compounded by the fact that CUDAnative's diagnostics can also be very difficult to understand. Often they aren't clear about which line of code caused an error, even when tools like `@device_code_warntype` is used.

These issues led to our being unable to use shared memory. We had to reimplement several tools from base Julia (such as `CartesianIndex`) and write some custom helper functions in order to copy chunks of our input arrays to shared memory. This code was **correct**; we were able to verify that it wrote and read the correct values. However, it didn't result in any speedup. Somewhere we were performing an operation that lacked mechanical sympathy. However, since we couldn't use a profiler, we didn't know which operation that was.

CUDAnative is extremely powerful, and allows many low-level operations to be implemented in ways that synchronize harmoniously with Flux and CuArrays. Unfortunately, some issues in the

tech stack prevented us from using it to its full potential. Still, working on the project was very informative, and we look forward to leveraging this tech stack in the future once some of the kinks are ironed out.

4 Future work

If we were to continue working on this project, we would first want to get shared memory working, perhaps by running the code under `nvprof` on several CUDA-enabled machines until we can find one that works. After that, we'd like to optimize the kernels further, and then integrate our binary and floating point implementations, and see what optimizations that enables. One optimization we're particularly curious about would be fusing the convolution, batchnorm, relu, and quantization kernels. This would mean that only packed bits would need to be written to global memory, with all floating-point numbers used in the kernel kept in shared memory, minimizing the necessary global memory bandwidth.

One other interesting direction of development would be the development of a tool like CUB (`https://nvlabs.github.io/cub/`) for Julia/CUDAnative. CUB provides a number of thread-cooperative algorithms that can be used directly within CUDA kernels. Having access to a tool like that would make CUDAnative significantly easier to use, and allow faster development of high-performance kernels.

References

Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pp. 1–70, Aug 2008. ISSN null. doi: 10.1109/IEEESTD.2008.4610935.

T. Besard, C. Foket, and B. De Sutter. Effective extensible programming: Unleashing julia on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 30(4):827–841, April 2019. doi: 10.1109/TPDS.2018.2872064.

Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017. URL https://doi.org/10.1137/141000671.

Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model compression and acceleration for deep neural networks. *CoRR*, 2017. URL http://arxiv.org/abs/1710.09282v8.

Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, 2016. URL http://arxiv.org/abs/1602.02830v3.

Karen Hao. Training a single ai model can emit as much carbon as five cars in their lifetimes. *MIT Technology Review*, Jun 2019.

Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, 2017. URL http://arxiv.org/abs/1704.04861v1.

Mike Innes. Flux: Elegant machine learning with julia. *Journal of Open Source Software*, 2018. doi: 10.21105/joss.00602.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014. URL http://arxiv.org/abs/1412.6980. cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

Yann LeCun, John S. Denker, and Sara A. Solla. Optimal brain damage. In D. S. Touretzky (ed.), *Advances in Neural Information Processing Systems 2*, pp. 598–605. Morgan-Kaufmann, 1990. URL http://papers.nips.cc/paper/250-optimal-brain-damage.pdf.

Xiaofan Lin, Cong Zhao, and Wei Pan. Towards accurate binary convolutional neural network. *CoRR*, 2017. URL http://arxiv.org/abs/1711.11294v1.

Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. *CoRR*, 2016. URL http://arxiv.org/abs/1603.05279v4.

Giuseppe Tagliavini, Stefan Mach, Davide Rossi, Andrea Marongiu, and Luca Benini. A transprecision floating-point platform for ultra-low power computing. *CoRR*, 2017. URL http://arxiv.org/abs/1711.10374v1.

Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.