Productive Generation of Portable, Efficient Code for Convolutional Neural Networks

Matthew W. Moskewicz Forrest N. Iandola Kurt Keutzer
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94402
{moskewcz,forresti,keutzer}@eecs.berkeley.edu

***Abstract*—The popularity of neural networks (NNs) spans academia [1], industry [2], and popular culture [3]. In particular, convolutional neural networks (CNNs) have been applied to many image based machine learning tasks and have yielded strong results [4]. The availability of hardware/software systems for efficient training and deployment of large and/or deep CNN models has been, and continues to be, an important consideration for the field [5] [1]. Early systems for NN computation focused on leveraging existing dense linear algebra techniques and libraries [6] [7]. Current approaches use low-level machine specific programming [8] and/or closed-source, purpose-built vendor libraries [9]. In this work, we present an open source system that, compared to existing approaches, achieves competitive computational speed while achieving higher portability. We achieve this by targeting the vendor-neutral OpenCL platform [10] using a code-generation approach. We argue that our approach allows for both: (1) the rapid development of new computational kernels for existing hardware targets, and (2) the rapid tuning of existing computational kernels for new hardware targets. Results are presented for a case study of targeting the Qualcomm Snapdragon 820 mobile computing platform [11] for CNN deployment.**

I. Introduction and Motivation

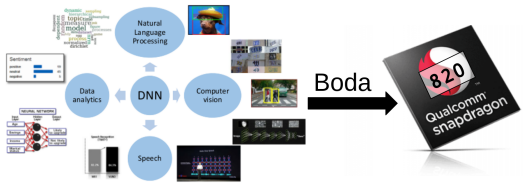

Figure 1. Boda overview: from CNNs to mobile computing platforms.

To support ongoing research, development, and deployment of systems that include NNs, it is desirable to continue to nurture a diverse enabling ecosystem of hardware and software tools and approaches. Imagine that, for a given task, high-performance vendor libraries exist for at least one platform. Currently, for CNNs, this vendor is nVidia, the platform is Maxwell, and the library is cuDNN [9]. Why not simply use that vendor's platform and libraries for the task and be satisfied? One issue is quite simple: in industrial use cases, choice of platform may be dictated by business concerns. Further, those same business concerns may preclude dependence on any single vendor. For example, the flagship Samsung Galaxy S7 mobile phone ships in two versions: one using a Samsung-proprietary Exynos 8890 System-on-Chip (SoC), the other using the Qualcomm Snapdragon 820 [11] SoC. Neither of these SoCs contains nVidia GPUs or are otherwise capable of running cuDNN. Further, nVidia, Qualcomm, and Samsung have engaged in a long running patent dispute over GPU technologies. Based on the uncertainties associated with such litigation, SoC and/or GPU alternatives are subject to constant change. Further, business needs may dictate the desired task to perform. In an effort to provide differentiation, vendors have a strong desire to implement novel functionality not present in existing publicly available libraries. Together, these uncertainties about both target hardware and particular use-case create a strong pressure for *portability*: the ability to quickly achieve reasonably performance for a variety of tasks across a variety of platforms.

This work focuses on the challenges associated with achieving portable, efficient computation of the key primitive operations needed by convolutional neural networks. We frame the problem as follows:

For a particular use-case/flow consisting of a specific combination of:

- a target computational device (i.e. a hardware architecture),
- a set of convolutional neural network primitives (i.e. a CNN architecture),
- a set of problem sizes of interest, and
- a set of performance requirements,

it is, in general, a difficult task to achieve good computational efficiency. Building systems based on existing computational libraries (e.g. BLAS) generally achieves only limited efficiencies [12]. Improving on such approaches requires tuning multiple computational kernels for the particular use-case at hand. This generally requires months of effort by a very specialized programmer. This programmer must be one that is both capable of producing high-efficiency code for a target platform as well as being familiar with the details of CNN

computations. Such programmers are not common and thus their time is a very limited resource.

In this work, we present an open-source vertically-integrated framework for developing and deploying CNN computations. The framework combines parts of the functionality of CNN middleware frameworks such as Caffe [7] or TensorFlow [13] with the functionality of CNN computation libraries such as nVidia's cuDNN [9] or Nervana's nervanagpu/neon [14] [15]. "Out-of-the-box", our framework does not attempt to have the breadth or depth of features of a typical general-use middleware such as Caffe or TensorFlow. Also, it does not attempt to achieve the same efficiency as a highly-target-specific computational library such as cuDNN. Instead, we aim to allow for the rapid development and deployment of new use-cases/flows of the form described above. In particular, the framework enables productive, flexible, modular OpenCL code generation of the full set of needed CNN operations for any given use case.

This allows for both:

- the rapid development of new computational kernels for existing hardware targets, and
- the rapid tuning of existing computational kernel for new hardware targets.

To provide support for these claims, we present a case study of using the framework to target the Qualcomm Snapdragon 820 mobile computing platform [11] for CNN deployment (see Figure 1). We show that the framework enabled a quick path to initial functional correctness, and from there helped navigate a smooth path to reasonably efficient computational performance.

The rest of the paper is organized as follows. In Section II we review the semantics of the key computational operations needed to support CNNs, focusing particularly on convolution. Then, in Section III we discuss performing CNN calculations on modern computational hardware, focusing particularly on the issue of data reuse. We take a historical approach to this explanation, citing related work along the way. In Section IV we introduce our framework using the example kernel of single-precision matrix-matrix multiply (SGEMM). The use of such a relatively simple, existing computational kernel allows us to:

- illustrate in general what our framework does and how to apply it,
- to show in particular how we approach a new hardware target, and
- to provide a good baseline for general performance evaluation.

Then, we present our core contribution of productive, portable, efficient, code generation for CNN operations (via case study) in Sections V and VI. Finally, we conclude in Section VII.

II. Introduction to CNN Computations

Different CNN architectures can contain a wide variety of computational primitives. For completeness, it is important that any CNN computation system supports a reasonably broad set of such primitives. Further, it is desirable that support for new operations, particularly those that are simple, can be easily added. The most common operations needed for deployment/testing include *convolution*, *pooling* and *softmax*. However, across many common networks, such as AlexNet [6], GoogLeNetV1 [16], and the VGG networks [17], *convolution* operations dominate the overall computation. Thus, in this section, we focus on the convolution operation. In particular, we consider the commonly occurring forms of convolutions from the above (and similar) CNNs. Given the nature of this work, we will focus on practical, operational definitions of the relevant computations and the data on which those computations act. Although various types of data are used in CNNs, the most common are 32-bit (and, increasingly, 16-bit) floating point numbers. Regardless of exact type or precision, the numeric values used by CNNs are grouped into named ND-Arrays (i.e. collections of numbers with N indexes, sometimes also called N-D Matrices or tensors). Thus, we define convolution operationally as the function $out = conv(in, filts)$ where out, in, and $filts$ are all N-D arrays. For simplicity, we omit discussion of the common practice of special-case handling of biases here. Further, we will restrict the discussion to convolution over 2D images, which is the only type used in the above example networks. Thus, for a single 2D multi-channel image, both in and out are 3D arrays, with their dimensions being the image height, the image width, and the number of image channels. For example, the overall input to a network might be an RGB (3 channel) image with a size of 205x205 pixels. Thus the dimensions of the 3D array storing this image ($dims(in)$) could be written compactly as "YxXxC = 205x205x3", where the H, W, and C *name* the dimensions, and the 200, 200, and 3 are the *concrete sizes* of those dimensions. Often, particularly for training, it is desirable to process a *batch* of multiple images. In this case, an additional "B" dimension is added to in and out. For the example of a 32 image batch, $dims(in)$ becomes "BxYxXxC = 32x205x205x3". The results of the convolution are fully independent across input/output image pairs. Thus, for simplicity, we assume a single image for the remainder of this discussion. The semantics of convolutions are determined by several architecturally specified values: the number of output channels, the kernel size, and the stride. While in general the kernel size and stride can be different in each spatial dimension (i.e. X and Y in the 2D case), for simplicity, here we consider only the case where kernel size and stride are the same for all dimensions (i.e. are scalars). The kernel size and number of output channels, combined with the number of channels in in, determine $dims(filts)$. In our running example, recall that the number of input channels $IC = 3$. If the number of output channels $OC = 96$, and the kernel size $KSZ = 7$, then $dims(filts)$ will be "OCxKSZxKSZxIC = 96x7x7x3". Finally, the width of the output will be given by $out_X = 1 + (in_X - KSZ)/stride$

(and similarly for the height). Thus, if the stride for this example is 2, then we have $out_X = 1 + (205 - 7)/2 = 100$, and $dims(out)$ will be "YxXxC = 100x100x96". Note that in is often padded by $\lfloor KSZ/2 \rfloor$ elements (usually zeros) in each spatial dimension. When such padding is applied (assuming $stride = 1$), out will have the same spatial dimensions (height and width) as in. We can calculate each output channel oc using in and the 3D slice of $filts$ where $OC = oc$. That is, we use each slice $filts[OC = oc]$ (dims "KSZxKSZxIC = 7x7x3") to compute each slice $out[OC = oc]$ (dims "YxX = 100x100"). Then, for each output point $out[OC = oc, Y = oy, X = ox]$, we extract a KSZxKSZ window of in (sometimes called an *input patch* or *patch*) starting at $Y = oy * stride, X = ox * stride$, across all input channels, to yield the slice $in[Y = [oy * stride, oy * stride + KSZ), X = [ox * stride, ox * stride + KSZ)]$ (dims "YxXxC = 7x7x3"). The final value of each output point is the sum of all elements of the element-wise product of the per-output-channel slice of $filts$ and the per-output-x-y-point slice of in. Typically some *activation function*, such as ReLU ($max(0, x)$), is next applied to each output value, and this operation is often fused into the convolution operation itself for efficiency. For later reference, note here that each in slice is reused across all output channels, and each $filts$ slice is reused across all output x-y points. Similarly, note that if multiple images are processed in a batch, the number of input-patches/output-x-y-points is multiplied by "B". Lastly, note that the computation performed by an entire "standard" fully-connected NN layer can be viewed as a special case of the convolution operation where the stride is 1, the kernel size is equal to the input size, the number of output channels is the number of neurons in the layer. Thus there is a single input slice per layer, consisting of the entire input, and the output has a width and height of 1.

III. CNN Computations on Modern Hardware And Related Work

In the prior section, the final computation of each (scalar) output value in a convolution consisted of an element-wise product of two ND-Arrays (one a slice of the input, one a slice of the filters, with exactly equal dimensions), followed by a sum over all the elements of the product. If we were to conceptually *reshape* or view the two 3D slices as 1D arrays (i.e. vectors, with size equal to the product of the 3 3D-Array dimension sizes), we can see that this operation is simply a vector dot-product. Thus, the core computational operation of a large class of NNs (including CNNs) is to perform many dot-product operations between a large set of input vectors and model parameter vectors (also known as *filter weights*, *filters*, or *weights*). In NNs with multiple *layers*, the outputs of one operation may become inputs to another. In CNNs, the slices that form the inputs to each dot-product may spatially overlap and thus share data with each other. Recall that in our running example, the we have $dims(out)$ of "YxXxC = 100x100x96", yielding $100 * 100 * 96 = 960000$ output points, each requiring one dot-product to compute. If we consider the set of all input-slice/filter-slice pairs to these 960000 dot-products, we see that it is formed from the cross product of $100 * 100 = 10000$ input slices and 96 (per-output channel) filter slices. Thus, each dot-product input is reused across many dot-products: 96 dot-products for each input-slice, and 10000 dot-products for each filter-slice.

At least as early as 2004, work using GPUs to accelerate NNs made the key observation that this data reuse pattern of NNs makes then well suited for calculation using GPUs [5]. Based on the amount of calculation hardware available, and the rate at which it can perform calculations, each computational system has a *peak* computational rate. Computations which can achieve this peak rate on a given system are termed *compute limited*. But, on modern computational systems (CPUs and GPUs), computations such as batches of independent dot-product operations are instead limited by the time taken to transfer operands between different storage locations in the system. This is termed a *bandwidth limited* computation. Depending on the ratio of communication resources to compute resources in a system, it is necessary to reuse data to varying degrees at different levels of the storage hierarchy of the system to avoid becoming bandwidth limited [18]. Over time, the relative cost of communication has increased compared to that of communication, so systems that have high absolute peak computational rates (for a given power level) require increasing amounts of data reuse to achieve those rates [19].

Returning to our working example, recall that the size of each input-slice is $7 \times 7 \times 3$, or 147 elements. If we form a 2D matrix using all 10000 input slices as rows, we will have a 10000×147 matrix $inmat$. Similarly, we can form 2D matrix using all 96 filter slices as columns, yielding a 147×96 matrix $filtsmat$. We can now express the entire convolution operation as a single matrix-matrix multiply operation: $outmat = inmat * filtsmat$. Note that $outmat$ has $10000 * 96 = 960000$ elements, so we can reshape it to the desired $dims(out)$ of "YxXxC = 100x100x96". Due to the dot-product-input reuse discussed earlier, such matrix-matrix multiplies are generally amenable to high-efficiency GPU implementations [20]. Early CNN frameworks such as cuda-convnet [6] and Caffe [7] originally performed CNN convolutions in exactly this fashion, leveraging nVidia's cuBLAS matrix library.

However, there are several limitations of this BLAS-based approach:

- When $in_X, in_Y >> KSZ$, $inmat$ is roughly $(KSZ/stride)^2$ larger than in; thus explicitly creating the matrix $inmat$ may require significant intermediate memory, and to a lesser extent, non-negligible time.
- It does not allow for data reuse between spatially overlapping input slices.
- The underlying matrix-matrix multiply library may not

be well optimized for the problem sizes required.
- It is not possible to fuse an activation function with the convolution operation.
- It is not possible to use various other optimizations, such as Winograd convolution [21].

Regardless of the exact reasons, it became clear that BLAS-based approaches were often falling far short of peak compute. Overall community efforts then turned to purpose-built libraries for convolution, so it remains an open research question what the limits of BLAS-based approaches are in various cases.

In particular, nVidia soon released the cuDNN [9] library, which exposes an API for directly performing convolutions using a variety of approaches. Due to the closed-source nature of the library, and the fact that it has evolved rapidly, it is difficult to analyze in detail. However, it is clear that it achieves much higher efficiency than BLAS-based approaches in many cases, and is within 50% of peak compute for many use cases. Concurrently, a family of libraries based on an assembly-language-level programming flow appeared [8] [14] [15], offing similar performance to cuDNN. Historically, the assembly language level approaches have offered the best performance at any given time, with cuDNN catching up in its next release. Since the assembly-language kernels are open source, one could speculate that nVidia is copying or reimplementing them internally. If true, it would imply that the entire high-performance CNN computation ecosystem is anchored by small cluster of approaches and programmers – perhaps as few as one or two key figures. One result of this state of affairs is that there is effectively only one usable hardware platform for high efficiency CNN computations: nVidia Maxwell GPUs. In particular, there is a lack of support for OpenCL on AMD or any other GPU platforms, due to the lack of an OpenCL library comparable to cuDNN or nervanagpu. Currently, in the caffe framework, support for OpenCL uses only the BLAS-based approach, and even that support is marginalized and fragmented across various forks [22] and pull requests. For completeness, we briefly consider the state of the art for performing convolution on CPUs. Recently, Intel has made significant advances in supporting convolutions, and is competitive with GPUs at large scale [23]. However, in practice, most use cases are still single socket, or at most single node, and current CPUs cannot offer competitive performance at this scale, particularly using consumer (as opposed to server) parts. Also, it is unclear that CPUs are currently competitive with GPUs on a power or cost basis for CNN computation. Note that a detailed comparison of CPUs vs. GPUs is outside the scope of this paper.

We feel that, for a healthy ecosystem, it is desirable to support many hardware platforms to enable new applications across many areas, including mobile, IoT, transportation, medical, and others. With this work, we attempt to bring OpenCL support for CNN computations to a more even footing with existing high efficiency approaches. While we focus on OpenCL targeting GPUs, extending our results to CPUs is a reasonable topic for future work.

IV. Warmup: Code Generation for SGEMM

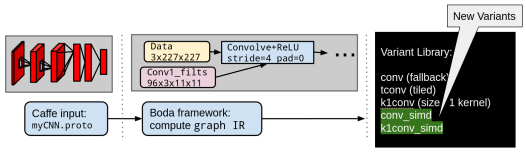

Figure 2. Boda overall flow from CNN description to compute graph (to be matched against variant library).

At a high level, our framework employs a code generation flow summarized in Figure 2. To explain the details of how this flow works in practice, we start with the application of our approach to a simple example: matrix-matrix multiplication. As discussed earlier, early approaches to CNN convolutions employed matrix-matrix multiplication as the key computational primitive. This allowed the usage of high-quality vendor provided BLAS libraries (such as cuBLAS from nVidia) for the bulk of the computation. In particular, one key BLAS function, single precision matrix-matrix multiply, or SGEMM, implements the core computation. Also, following the Caffe naming convention, an auxiliary function $inmat = im2col(in)$ provides the conversion from the ND-Array in to the 2D-matrix-of-input-slices $inmat$. Note that $filtsmat$ is typically simply a reshape of $filts$. As mentioned earlier, the expansive nature of $im2col()$ can be problematic. In particular, for large batches of images, the size of $inmat$ (for a single layer) may exceed GPU memory. To avoid this, batches may be handled with separate per-image calls to $im2col()$ and $SGEMM()$. SGEMM is a simple and well studied function across many hardware architectures. The main required tasks for achieving an efficient SGEMM implementation for a given hardware target are as follows:

1) Arranging accesses to storage to make best use of system's communication potential.
2) Achieving sufficient data reuse at each level of the system's storage hierarchy to avoid being bandwidth limited.
3) Ensuring computational units are continually active; (2) is necessary but not sufficient for this.
4) Repeating (1), (2), and (3) for all interesting input sizes.

Unfortunately, these goals are often both interrelated and in conflict with each other. For example, it is often the case that accessing storage contiguously, or in certain patterns, achieves higher bandwidth than others. Thus, there may be a tradeoff between achieving good communication bandwidth and reading the best set of data for reuse at the next level. In general, (1) and (2) directly balance each other. So, if a factor of N additional reuse can be achieved at anything less

than a factor of N cost in bandwidth, it is favorable to do so. The primary goal is (3); if the maximum compute rate can be achieved, other concerns are secondary. However, (1) and (2) effect system power, and code that achieves (3) at first, might, over time, become thermally limited. Thus, even after (3) is achieved for a nominal situation, further optimization to (1) and (2) may be important. For (4), if the set of input sizes is small (or a small set of ranges of sizes) and known in advance, it may be tractable to create an SGEMM variant for each input size or range of sizes. Further, a single variant can often handle multiple input sizes, particularly if the set of sizes is somehow restricted. Typically, the input sizes for SGEMM are expressed as M, K, and N, all scalars, where if $c = SGEMM(a, b)$, then c is an $M \times N$ matrix, a is $M \times K$, and b is $K \times N$. For example, it might be reasonable to design an SGEMM variant that can handle the case where $M \approx K \approx N$, $1024 < M, N, K < 4096$, and $M, N, K \bmod 32 = 0$. If needed, sizes that are not multiples of 32 can be handled by padding and/or remainder-processing sub-functions. Typically, it is difficult to achieve good efficiency across a range of sizes and hardware targets without some form of *metaprogramming*. We define metaprogramming as the collection of techniques where, instead of directly writing code, some higher level facility is used to create the desired final code. C Macros, C++ templates, and ad-hoc code generation are all examples of metaprogramming, and all offer similar key benefits:

- Values that are known ahead of time can be constant in the final code without (potentially repeatedly) hard-coding them at the source level.
- Many variants of a single version of an algorithm can be generated simply by varying parameters at the meta level.
- Repetitive sections of code can be generated, rather than manually written. This is particularly important when simpler techniques such as compiler-driven loop unrolling are insufficient or cumbersome.

The usage of metaprogramming for SGEMM for GPUs seems to be commonplace; as far the authors are aware, it is used to varying degrees by all modern, efficient GPU BLAS libraries.

Our framework uses a combination of string-replacement templates, specific support for known-size ND-Array access, and ad-hoc unrestricted programmatic code generation. This places it toward the more flexible/extreme end of the space of metaprogramming techniques. Although metaprogramming is inherently complex, we nonetheless attempt to achieve a good balance between flexibility, power, simplicity, and easy of use. To implement SGEMM in OpenCL for GPU targets, we generally employ the following standard techniques (for example when targeting nVidia GPUs):

- Register Tiling [24]
- Explicit Local Memory Blocking
- Inner Loop Unrolling

The basic template (in simplified pseudocode form) for our SGEMM is show in Listing 1.

Listing 1. SGEMM code template

```
void SGEMM( nda M:K a, nda N:K bt, nda M:N c ) {
  // dims work Mg:Ng:Mb:Nb:Kb:Mt:Nt // blocking values
  local a_lm[%(work_Kb_dim)*%(work_Mb_dim)*%(work_Mt_dim)];
  local b_lm[%(work_Kb_dim)*%(work_Nb_dim)*%(work_Nt_dim)];
  // per-thread tile of output to compute, stored in registers
  float c_t[%(work_Mt_dim)*%(work_Nt_dim)] = {0};
  float a_r[%(work_Mt_dim)];
  float b_r[%(work_Nt_dim)];
  for( int32_t k = 0; k < %(a_K_dim); k += %(work_Kb_dim) ) {
    BARRIER_SYNC;
    // workgroup-wide load of local memory for this iteration
    %(lm_loads);
    BARRIER_SYNC;
    for( uint32_t subk = 0; subk < %(work_Kb_dim); ++subk ) {
      %(loads); // load per-thread slice of a, b into a_r, b_r
      %(fmas); // perform fused multiply-adds
    }
  }
 // iterate over rows of the Mt * Nt registers in c_t
  for( int32_t m = 0; m < %(work_Mt_dim); ++m ) {
    %(transpose_c_t_row); // transpose row of c_t into b_r
    %(store_c_t_row); // store transposed row of c_t to c
  }
}
```

For targeting CNN convolutions, it is acceptable to use this template to generate a variant of SGEMM for every unique input size. However, if this not desirable, the above mentioned techniques can be used to generate variants than can handle ranges of sizes. For a given input size and hardware platform, the first main task of the code generation is to determine a good blocking strategy. This can be accomplished with a combination of heuristic calculations, manual parameter tuning, or automated tuning. In this work, we use only the first two approaches; use of automated tuning is a good subject for future work. The general flow from operation (SGEMM or convolution) to blocking constants is illustrated in Figure 3. For SGEMM (or later for convolution as well),

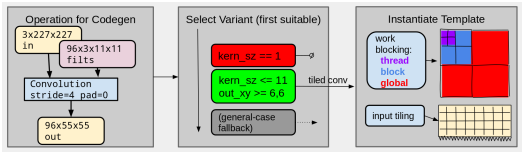

Figure 3. Boda code generation flow from CNN operation to GPU code and work blocking.

the main blocking parameters we must choose are:

- Mt, Nt: The number of output elements computed per thread in the M and N dimensions. Typically in $[1, 8]$. Note that the kernel will require at least $Mt \times Nt$ registers per thread.
- Kb: The inner loop unrolling factor. Typically in $[1, 8]$. If too small, loop overhead becomes excessive. If too large, Instruction or Local memory will be insufficient, as both scale linearly with Kb.
- Mb, Nb: Determines the total number of output elements computed per workgroup. Valid and/or desir-

Table I
SGEMM OPERATION SPEED AND EFFICIENCY: CUBLAS VS. BODA

Size	Communication/Compute			cuBLAS Performance		Boda Performance		
MKN	Bytes	FLOPs	F/B	Runtime	GF/s	Runtime	GF/s	Speedup
128	197KB	4.19MF	21.3	35.8us	117GF/s	53.1us	79.0GF/s	1.48
256	786KB	33.6MF	42.7	54.3us	618GF/s	49.6us	677GF/s	0.91
384	1.77MB	113MF	64.0	88.1us	1.29TF/s	64.4us	1.76TF/s	0.73
512	3.15MB	268MF	85.3	121us	2.22TF/s	104us	2.59TF/s	0.86
768	7.08MB	906MF	128	257us	3.52TF/s	206us	4.40TF/s	0.80
1024	12.6MB	2.15GF	171	605us	3.55TF/s	605us	3.55TF/s	1.00
1536	28.3MB	7.25GF	256	1.64ms	4.42TF/s	1.38ms	5.25TF/s	0.84
2048	50.3MB	17.2GF	341	4.08ms	4.21TF/s	3.52ms	4.87TF/s	0.86

able values vary by hardware architecture, but often $Mb \times Nb \equiv threads\_per\_workgroup$ should be in $[32, 256]$. When $Mb \approx Nb$, both Local memory usage and the Global/Local memory reuse factor scale linearly as Mb (or Nb).

- Mg, Ng: Based on M, N and the prior choices of Mt, Nt, Mb, Nb, we have $Mg = \lceil M/Mb/Mt \rceil$ and $Ng = \lceil N/Nb/Nt \rceil$. $Mg \times Ng$ workgroups will needed to compute the final result. For many targets, the total number of workgroups must be above a threshold to saturate all computational elements in the device. Small values may be subject to performance limitations if they are not a multiple of some particular constant.

After determining the blocking constants, the code generator must emit code for various blocks:

- %(lm_loads): workgroup-wide cooperative global memory → local memory loads
- %(loads):$Mt + Nt$ per-thread local memory → register loads into $a\_r[m], b\_r[n]$
- %(fmas): $Mt \times Nt$ in-register multiply-adds: $c\_t[m][n] += a\_r[m] * b\_r[n]$
- %(transpose_c_t_row), %(store_c_t_row): transpose and write per-thread outputs into c

The general challenges when emitting these blocks are to minimize conditionals and choose particular constructs that can execute efficiently. Careful data layout at the global, local, and register levels may be required, potentially with additional code to reorganize, shuffle, or transpose data at each level. With a few days of effort, reasonable SGEMM performance on an nVidia Titan-X GPU was achieved. See Table I for a comparison of this template's performance with that of the SGEMM from nVidia's highly-tuned cuBLAS library. In short, with moderate effort our framework achieved about 80% of the performance of the vendor library, albeit only for a few simple cases.

Next, we turn our attention to our main focus: our new hardware target, the Snapdragon 820 (SD820). The first key thing we learned about this platform is that manual vectorization of load and stores yields 2X or more increases in load/store bandwidth. Further, at least for SGEMM on the SD820 platform, we find that it is not generally profitable to explicitly move data from global to local memory. Instead, we apply our usual work-blocking strategy, but simply omit the loads and stores to local memory, and read/write global memory directly. Due to the access pattern of the blocking, the hardware cache appears to provide data reuse similar to that of using local memory explicitly, and we avoid the overhead of both the local memory accesses and related synchronization overheads. However, note that the resulting bandwidth amplification is limited, perhaps due to overall limited bandwidth from cache/local-memory. Unfortunately, the SD820 development platform does not provide sufficient profiling information, hardware/ISA documentation, or other tools (such as a disassembler) to perform a more in depth performance analysis. So, a few days of effort was spent on blind experimentation, guesswork, and tuning to improve results. In the end, we adapted out approach for the SD820 using the following techniques:

- Manual vectorization of loads/stores
- Local-memory based output buffers (which seems to a compiler optimization triggered by high register use)
- Direct global memory access with cache Blocking (i.e. don't attempt to use local memory explicitly)

The resulting speed of our SGEMM on the SD820 platform is 2X that of the vendor provided SML/QBLAS library; see Table II. Note, however, that there is currently no vendor provided GPU-based SGEMM, so this comparison is against code running on the CPU portion of the SD820 SoC. Currently, it is not clear if any other OpenCL BLAS libraries can target the SD820 platform without some significant additional efforts. Thus, research of, profiling of, and comparison against other OpenCL-based BLAS libraries is a good topic for future work.

V. CODE GENERATION FOR CNN CONVOLUTIONS

As discussed earlier, the BLAS-based or $im2col()$/$SGEMM()$ approach to performing convolutions has various limitations. Now, we turn our attention to using our framework to generate functions that directly perform convolutions. The first variant we will discuss is a simple fusion of $im2col()$ and $SGEMM()$. We apply the same basic techniques as in the $SGEMM()$ discussion above,

Table II
SGEMM OPERATION SPEED AND EFFICIENCY: QBLAS VS. BODA

Size	Communication/Compute			QBLAS Performance		Boda Performance		
MKN	Bytes	FLOPs	F/B	Runtime	GF/s	Runtime	GF/s	Speedup
128	197KB	4.19MF	21.3	560us	7.5GF/s	264us	16GF/s	2.1
256	786KB	33.6MF	42.7	1.7ms	20GF/s	507us	66GF/s	3.3
384	1.77MB	113MF	64.0	3.8ms	30GF/s	2.1ms	54GF/s	1.8
512	3.15MB	268MF	85.3	8.0ms	34GF/s	3.6ms	74GF/s	2.2
768	7.08MB	906MF	128	23ms	39GF/s	11ms	78GF/s	1.9
1024	12.6MB	2.15GF	171	54ms	40GF/s	27ms	80GF/s	2.0
1536	28.3MB	7.25GF	256	175ms	41GF/s	89ms	81GF/s	1.9
2048	50.3MB	17.2GF	341	401ms	42GF/s	223ms	77GF/s	1.8

but we fold the behavior of $im2col()$ into the %(lm_loads) code block. Or, in other words, we only *implicitly* create the matrix $inmat$; we just read the correct elements from in as needed. While this approach is simple, and avoids the memory overhead of creating $inmat$ explicitly, it makes task (1) much more difficult. In particular, both the overhead of additional indexing logic and the resultant poor access patterns reading global memory can render the performance of this variant less than the BLAS-based one. However, it provides a starting place for further explorations, and can function as a fallback method for convolutions not handled by more specialized variants. We term this the *implicit-SGEMM* or `conv` variant. The next variant we consider exploits the common case where the convolution kernel size KSZ is 1. In this case, various simplification are possible, and it is relatively easy to use a transformation function over in to ensure a good global memory access pattern. Note that, for $KSZ = 1$ convolutions, per-image $im2col()$ is the identity function; thus the `k1conv` variant is quite similar to SGEMM. The final variant we consider, termed `tconv` (tiled convolution), is targeted at the commonly occurring cases of kernel sizes in the range $[2, \sim 11]$, with reasonable widths for in (perhaps in the range $[KSZ * 5, KSZ * 50]$). In this case, we can perform some additional optimizations:

- We can fully unroll over the X dimension of the kernel. This uses significant but limited extra local memory and registers, but allows sharing of in row data in registers across unrollings of the inner loop.
- We can load entire X/Y *tiles* of in at the workgroup level. Even for kernels as small as 2x2, this vastly reduces the amount of data that must be loaded from global memory for in. The reduction is a factor of $(KSZ/stride)^2$; this is naturally the same factor by which $im2col()$ is expansive.

Again, an input transformation must be applied to in to help simplify indexing logic and improve memory access patterns.

For benchmarking, we consider a range of convolutions drawn from three common CNNs: AlexNet [6], NiN [25], and GoogLeNetV1 [16]. For each network, we consider batch sizes "B" of 1, 5, and 20. We then gather all the unique convolution operations, of which there are ~ 180. While some operations are duplicated within some networks, and the mixing together of operations from different batch sizes is perhaps not ideal, this set of operations represents a reasonable set over which low total runtime (summed over all operations) is desired. That is, absolute efficiency is generally less important than absolute runtime, and total runtime tends to be dominated by the larger convolutions. That said, high efficiency across a broad range of problems sizes is still desirable. Each particular network, batch size, and overall use case requires some particular subset of convolutions. As with SGEMM, we initially compare our convolutions against nVidia's cuDNN library on an nVidia Titan-X. Due to space limitations, we do not present the full results of that experiment here. In summary, as with SGEMM, we achieve reasonable performance, albeit with a few weeks of effort rather than a few days. Note that of the ~ 180 convolutions, almost all are handled by either the `k1conv` variant (if they have a size 1 kernel), or otherwise by the `tconv` variant. Only a few cases fall though to the `conv` variant. In general, both `k1conv` and `tconv` provide significant speedups (2X or more) over the fallback `conv` variant. Then, we turn to our main focus in this work of targeting the SD820 platform. As with the SGEMM case, our main task is to manually vectorize loads and stores. Additionally, as with SGEMM, we avoid the explicit use of local memory, and instead rely on cache for global memory bandwidth amplification. So far, we have only implemented two new variants for the SD820 platform: `conv_simd` and `k1conv_simd`, which are manually load/store vectorized versions of their non-simd counterparts. As with the nVidia case, the `k1conv` variant provides significant speedup over the fallback `conv_simd` variant. Detailed speed results are presented for the SD820 platform in the next section.

VI. RESULTS

Here, for the SD820 platform, we show the speed of the OpenCL code generated by our framework for our benchmark set of convolutions. In Figure 4, we show the benefit of the `k1conv_simd` variant for the convolutions to which it can be applied: those with size 1 kernels. In

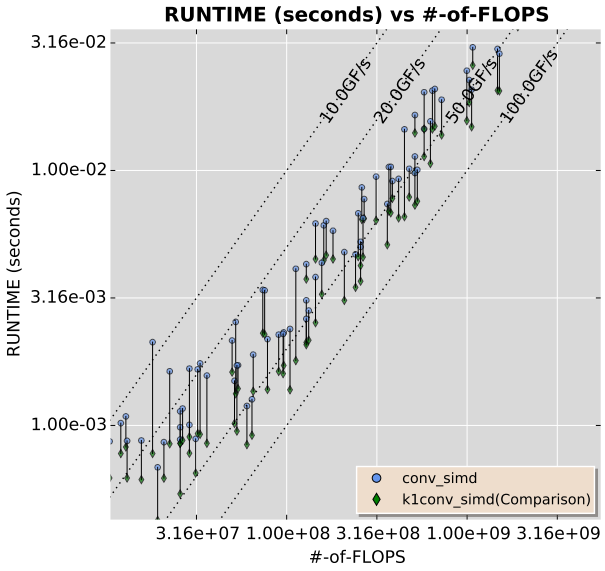

Figure 4. Speed improvement of special-case (k1conv) variant on SD820 platform.

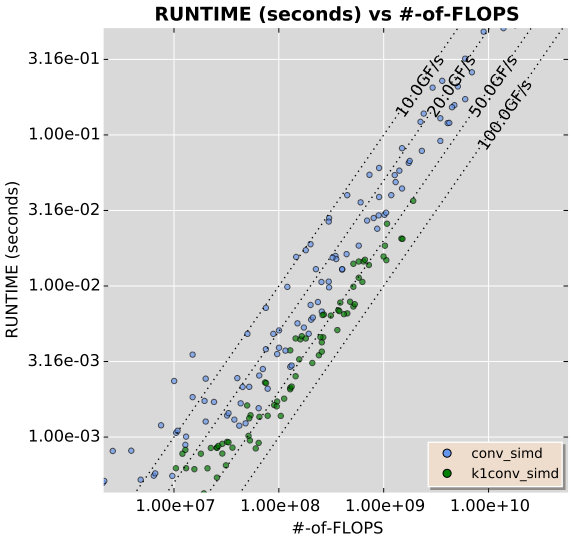

Figure 5. Per-convolution-size speed on SD820 platform.

Figures 5 and 6, we show the absolute performance of our framework's generated code for each benchmark convolution. As per the graph legend, for each convolution, we indicate which variant was selected. Inspecting the results, it can be seen that there are many convolutions with high arithmetic intensity (AI) that perform worse than those with lower AI. This is due to the fact that the higher performing cases are using the higher-efficiency `k1conv` variant. Based on our experience with the nVidia platform, we predict that implementing a `tconv_simd` variant for the SD820 platform will greatly improve the performance of most of

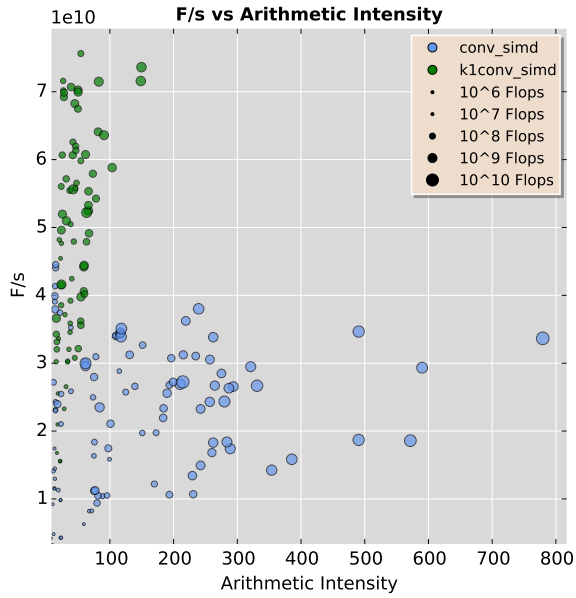

Figure 6. Same data as Fig 5, plotted as speed vs. arithmetic intensity.

the cases currently using the fallback `conv_simd` variant. Given the lack of a vendor CNN library or other libraries to directly compare against, it is difficult to know how close our performance is to optimal. From microbenchmarks, we know there is significant headroom over our results in terms peak compute performance on SD820 platform. However, similar microbenchmarks show that, in many cases, we may be at the limit of cache and/or global memory bandwidth. Thus, usage of smaller data types for storage (e.g. half-precision floats) and/or using hardware support for texture access are natural candidates to achieve additional improvements.

VII. Conclusions

CNNs are increasingly important in research and practice. As the community continues to broaden, interest in productive deployment of CNNs across many platforms and applications domains will only increase. But, the number of programmers capable of efficiently implementing CNN operations is very limited. As a result, support for high efficiency CNN calculation is currently limited to only a few hardware platforms. In this work, we have presented a novel approach to productively generate high efficiently CNN computations for any given hardware platform. We have demonstrated our approach with a case study of tuning CNNs deployment computations for the Snapdragon 820 mobile computing platform. By offering competitive performance and superior portability, we feel this work makes a key contribution to ability of the community to experiment with the deployment of CNNs across a wide range of platforms for a wide range of applications.

REFERENCES

[1] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[2] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. *arXiv preprint arXiv:1512.02595*, 2015.

[3] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[4] Ali Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. Cnn features off-the-shelf: an astounding baseline for recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 806–813, 2014.

[5] Kyoung-Su Oh and Keechul Jung. Gpu implementation of neural networks. *Pattern Recognition*, 37(6):1311–1314, 2004.

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS*, 2012.

[7] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv:1408.5093*, 2014.

[8] Andrew Lavin. maxDNN: an efficient convolution kernel for deep learning with maxwell gpus. *arXiv:1501.06633*, 2015.

[9] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: efficient primitives for deep learning. *arXiv:1410.0759*, 2014.

[10] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(1-3):66–73, 2010.

[11] Qualcomm. Snapdragon 820 processor. https://www.qualcomm.com/products/snapdragon/processors/820, 2016. [Online; accessed 4-April-2016].

[12] Soumith Chintala. convnet-benchmarks. https://github.com/soumith/convnet-benchmarks, 2016. [Online; accessed 4-April-2016].

[13] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL http://tensorflow.org/. Software available from tensorflow.org.

[14] Scott Gray and Nervana Systems. Nervana™ library for gpus. https://github.com/NervanaSystems/nervanagpu, 2016. [Online; accessed 4-April-2016].

[15] Nervana Systems. Fast, scalable, easy-to-use python based deep learning framework by nervana™. https://github.com/NervanaSystems/neon, 2016. [Online; accessed 4-April-2016].

[16] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *arXiv:1409.4842*, 2014.

[17] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[18] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[19] James Demmel. Communication-avoiding algorithms for linear algebra and beyond. In *IPDPS*, page 585, 2013.

[20] Vasily Volkov and James W Demmel. Benchmarking gpus to tune dense linear algebra. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–11. IEEE, 2008.

[21] Andrew Lavin. Fast algorithms for convolutional neural networks. *arXiv preprint arXiv:1509.09308*, 2015.

[22] AMD. Opencl version of caffe developed by amd research lab. https://github.com/amd/OpenCL-caffe, 2016. [Online; accessed 7-April-2016].

[23] Dipankar Das, Sasikanth Avancha, Dheevatsa Mudigere, Karthikeyan Vaidynathan, Srinivas Sridharan, Dhiraj Kalamkar, Bharat Kaul, and Pradeep Dubey. Distributed deep learning using synchronous stochastic gradient descent. *arXiv preprint arXiv:1602.06709*, 2016.

[24] Forrest N Iandola, David Sheffield, Michael J Anderson, Phitchaya Mangpo Phothilimthana, and Kurt Keutzer. Communication-minimizing 2d convolution in gpu registers. In *Image Processing (ICIP), 2013 20th IEEE International Conference on*, pages 2116–2120. IEEE, 2013.

[25] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *CoRR*, abs/1312.4400, 2013. URL http://arxiv.org/abs/1312.4400.