Boda: A Holistic Approach for Implementing Neural Network Computations

Matthew W. Moskewicz, Ali Jannesari and Kurt Keutzer
University of California, Berkeley
moskewcz,jannesari,keutzer@eecs.berkeley.edu

Abstract

Neural networks (NNs) are currently a very popular topic in machine learning for both research and practice. GPUs are the dominant computing platform for research efforts and are also gaining popularity as a deployment platform for applications such as autonomous vehicles. As a result, GPU vendors such as NVIDIA have spent enormous effort to write special-purpose NN libraries. On other hardware targets, especially mobile GPUs, such vendor libraries are not generally available. Thus, the development of portable, open, high-performance, energy-efficient GPU code for NN operations would enable broader deployment of NN-based algorithms. A root problem is that high efficiency GPU programming suffers from high complexity, low productivity, and low portability. To address this, this work presents a framework to enable productive, high-efficiency GPU programming for NN computations across hardware platforms and programming models. In particular, the framework provides specific support for metaprogramming and autotuning of operations over ND-Arrays. To show the correctness and value of our framework and approach, we implement a selection of NN operations, covering the core operations needed for deploying three common image-processing neural networks. We target three different hardware platforms: NVIDIA, AMD, and Qualcomm GPUs. On NVIDIA GPUs, we show both portability between OpenCL and CUDA as well competitive performance compared to the vendor library. On Qualcomm GPUs, we show that our framework enables productive development of target-specific optimizations, and achieves reasonable absolute performance. Finally, On AMD GPUs, we show initial results that indicate our framework can yield reasonable performance on a new platform with minimal effort.

CCS Concepts **•Computing methodologies → Neural networks; •Software and its engineering → Source code generation;**

Keywords computer vision, code generation, neural networks, mobile computing, convolution

1 Introduction

Productive, efficient parallel programming remains a challenging task with no general solution. In this work, we focus on a single facet of this broad issue: implementing neural network operations on GPUs. Taking a vertical approach spanning from high-level application to low-level programming, we present several contributions:

- A framework that provides a novel unified methodology, based on metaprogramming and autotuning, for productive development of portable, efficient implementations of a broad class of numerical functions targeting GPUs or similar platforms.
- Support for metaprogramming with ND-Arrays as a key data type, using named dimensions for improved productivity and type checking.
- A proof-of-concept use of the framework to implement the core set of operations needed for deploying three common image-processing neural networks, i.e., AlexNet, Network-in-Network, and Inception-V1 across three different GPU targets.
- An experimental evaluation of the resulting implementation, including a comparison to highly-tuned vendor library.

An additional contribution is that we provide a platform for future research, further experiments, and benchmarking related to GPU portability and metaprogramming. While our current focus is on neural network operations, any numerical operations that operate over ND-Arrays should be reasonably well supported by our approach. The entire framework, including support for automated replication of all results presented here, is made available as open source with a permissive license[1].

However, note that we specifically do not attempt to address the general problems of parallel programming, such as language and compiler design. We instead take the more pragmatic approach of layering over existing languages and compilers that are available on the platforms we target. Currently, creating efficient implementations of the types of operations we consider in this work is *extremely* difficult and time consuming. While the semantics of the operations are generally easy to express in a few lines of code in any language, efficient implementations for GPUs often require many *programmer-years* of effort. We provide one alternative method to develop such implementations. We show that our approach to such development represents a novel tradeoff among portability, speed, and productivity.

We achieve this with the careful application of both metaprogramming and autotuning in our proposed framework. We demonstrate the approach via a case study of mapping a core set of NN deployment computations, particularly convolutions, pooling, and activation functions, to NVIDIA, Qualcomm, and AMD GPUs. We show that we can target the same NVIDIA GPU hardware using either OpenCL or CUDA and achieve similar, high-efficiency results. This portability is possible due to the metaprogramming support provided by the framework, which allows syntactic compatibility between the core languages of the programming models (i.e. OpenCL and CUDA). Additionally, the framework abstracts away details related to compilation, allocation, scheduling, and execution that differ between OpenCL and CUDA. Also, we show that our approach eases the cumulative burden of targeting NVIDIA, Qualcomm, and AMD GPUs. Using the NVIDIA-tuned code as a starting point, we were able to achieve reasonable performance on a Qualcomm mobile GPU with only a few weeks of part-time effort by a programmer unfamiliar with the target. Then, using all the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CF'17, Siena, Italy
© 2017 ACM. 978-1-4503-4487-6/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3075564.3077382

[1] https://github.com/moskewcz/boda

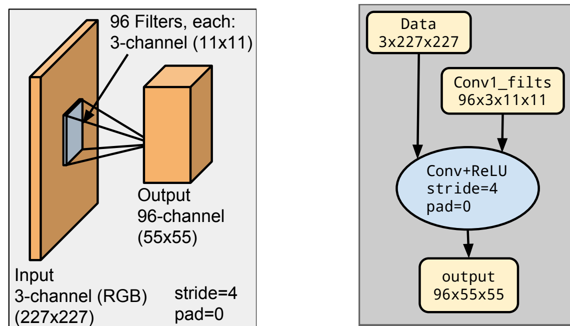

Figure 1. An illustration of a typical NN convolution (left) and the corresponding compute graph fragment (right).

code developed for the NVIDIA and Qualcomm platforms, we show the initial results of applying autotuning to target another new hardware platform, AMD GPUs. With no manual effort, we achieve a modest level of performance on AMD hardware, and argue that the profiling and tuning capabilities of the framework provide a great starting point for platform-specific optimizations.

2 Background and Motivation

Convolutional neural networks (CNNs) are NNs which make heavy use of 2D *convolutions* over multi-channel 2D images. CNNs have been quite successful in many computer vision applications such as object detection [4] and video classification [6]. In this work, the proof-of-concept set of operations we implement using our framework is drawn from three common CNNs: "AlexNet" [7], "Network-in-Network" [10], and the first version of Google's "Inception" network [16].

In addition to *convolutions*, CNNs commonly contain other operations such as *pooling* and *nonlinear activation functions*. However, for CNNs, convolution operations typically dominate the computation. Typically, convolutions require many operations (100s to 1000s or more) to produce a single output pixel, as each output pixel depends on all input pixels across all input channels within a convolution kernel-sized window of the input. In our discussion, we focus on convolution operations, as they are the most challenging operations to implement. However, note that we do implement *all* the operations necessary for deployment of our three considered CNNs, including the pooling and activation operations.

ND-Arrays, or collections of numbers with N indexes (sometimes also called N-D Matrices or tensors), are the main data type used for CNN computations. In particular, the input image, the filtered images produced by each layer (and fed as input to the next layer), and the filters themselves are all ND-Arrays. That is, each layer of convolutions in a CNN can be defined as the function *output = conv (input, filters)*, where *output*, *input* and *filters* are all ND-Arrays. The left side of Figure 1 shows an example of a single convolutional layer. Each output value of the convolution is the result of a dot-product between one of the 96 filters and an 11×11 region of the input image.

2.1 Problem Statement

Convolution, as used in neural networks, has simple-to-express semantics but is very difficult to implement efficiently. In particular, evidence suggests that such efforts invariably involve both low level programming and a significant degree of metaprogramming [9] [2]. Thus, rather than try to shield the programmer from such issues, we embrace both metaprogramming and direct, low-level programming in our approach, and attempt to make both activities as productive as possible.

Ideally, programmers could express NN operations such as convolution in a simple forms, such as a sets of simple nested loops, in the languages of their choice. Then, the compiler (or entire development toolchain) would create or provide an efficient implementation for the target platform. However, this has always been an elusive goal. At best, it simply shifts the fundamental implementation problems from the end developer to the developer of the toolchain. At worst, it adds substantial new problems, since the toolchain must solve a much more general problem than that of implementing a specific, known operation. In general, creating high-efficiency GPU implementations of numerical operations is no simple task. Many algorithmic and implementation issues must be considered, and a wide design space must be explored. In the end, the result of such work may be captured in many forms: libraries, frameworks, languages, or compilers. However, in this work, we are concerned more with how the initial algorithmic work and exploration is performed in the first place, rather than how it is eventually captured for final use (although this is also important). That is, let us say one believes that a compiler or language should handle some general case of creating efficient numerical code for some platform. However, in that case, it is still a prerequisite that *it is known* how to create efficient code for such operations for the given platform.

Our high level task is, given a NN and its inputs, efficiently compute its outputs. We can define a NN as a directed acyclic graph of (stateless) functions and ND-Arrays. We refer to this type of graph as a *compute graph*. Figure 1 shows an example of a convolution operation and its corresponding compute graph fragment. We term the process of converting from some description of a NN to the corresponding compute graph the *NN front-end*. In this work, as we are focused on the implementation of core computations, we are NN front-end *neutral*; as long as a suitable compute graph can be produced, any NN front-end can be used with our approach. Further, as mentioned earlier, while there are various operations in the compute graph, convolution is the most computationally challenging. Due to space limitations, we limit our discussion here primarily to the implementation of convolution.

2.2 Key Problems of Efficient GPU Convolutions

When implementing convolutions across multiple types of GPUs, there are two categories of problems. First, there are the fundamental challenges of implementing efficient convolutions on any GPU. Second, there are the issues of *portability* that arise when targeting multiple GPUs. Together, the full set of high-level problems we address with our approach are:

- Incompatible GPU programming models across different hardware: OpenCL and CUDA.
- GPU-hardware-specific constraints: memory size and access methods, organization and control of compute primitives.
- Data movement: getting data from off-chip to compute units and back, sizes and bandwidths of storage locations.
- Parallelism: what computations happen when and where.
- Managing overheads: conditionals, control flow, indexing.

To the best of our knowledge, our approach is the first to address these concerns in a unified, vertical framework for implementing NN convolutions on GPUs. In Section 3, we will discuss how our approach addresses each of these concerns using metaprogramming, autotuning, and other techniques.

2.3 Why portability?

One might ask, why not simply target each GPU separately to avoid portability issues? In short, there are many downsides to reimplementing convolutions for every GPU target. Aside from the initial duplicated effort, which is perhaps the primary issue, separate implementations complicate testing and maintenance. In practice, the bulk of high-performance, high-efficiency GPU code currently resides inside highly tuned libraries. Such libraries are generally tuned for only a small subset of GPUs – typically only those from a single vendor. As they are developed independently, they are often incompatible and support different sets of operations. They are also generally not extendable by end users. So, having a portable approach helps ensure compatibility and functional correctness across all platforms, both existing and new. Further, it encourages collaboration, which helps ease both extensiblity and the ability to efficiently target new hardware platforms.

3 Approach

Our general approach is to create a vertical framework that provides support for solving exactly the set of problems encountered when trying to efficiently implement convolutions on GPUs. We choose a middle path between the traditional library and compiler approaches to the mapping problem:

- Compared to a traditional library, our approach is more complex but much more flexible.
- Compared to general-purpose compilation, our approach is more limited but much simpler.

Compared to a full compiler, we do not aspire to support general purpose programming, and we avoid *all* mandatory, pre-existing, already-specified intermediary layers present in typical compilation flows. Thus, we can avoid much of the complexity of full, general purpose compilers. Although less general-purpose than a traditional compiler, we embrace compiler-like metaprogramming techniques, where the programmer writes code to dynamically generate and transform other code. Thus, we have much more flexibility to generate customized code for every possible combination of specific operations and hardware targets encountered at runtime, unlike a traditional pre-compiled library. Overall, using this approach, we can achieve high efficiency for the set of operations required for our application, while keeping overall complexity manageable. One key to efficiency and reduced complexity is that, at runtime, we need only handle the specific operations present in the input. Unlike a traditional library, we need not write and pre-compile code to handle the general case, and we are free to use *all* input-specific runtime information to aid in optimization. In particular, for each operation, we need only handle the specific input sizes used. As the number of possible input sizes is very large, such specialization is cumbersome and/or limited in the traditional library approach. Additionally, we can use information about the specific sequences of operations present to perform additional optimizations, such as fusion, as discussed in Section 3.4.

While we believe our approach is perhaps the most productive way to achieve our particular goals, we also realize the benefits of the traditional library and compiler approaches as well. In fact, we would argue that, as progress is made on the key problems of implementing efficient GPU convolutions (using our approach), it is then natural to: (1) generalize the techniques and embed them in a compiler, or (2) apply additional effort (and perhaps compromise speed) to allow for a fixed-library implementation.

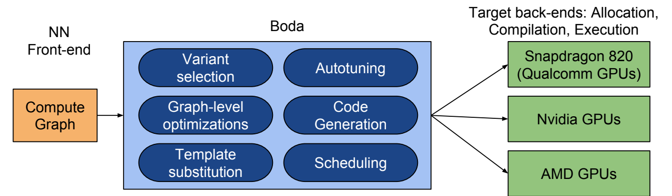

Figure 2. Overall structure of Boda.

3.1 Framework Structure

An overview of our framework for mapping NN computations to GPU hardware is shown in Figure 2. A compute graph is input to Boda, which performs various tasks to map it to different *target back-ends*. As with the front-end, our framework is back-end neutral. We require only that the target platform provide mechanisms for:

- Run-time Compilation (for metaprogramming/specialization),
- Memory allocation and execution of code, and
- Per-function-call timing (for profiling/autotuning).

Note that we do not support arbitrary languages or programming models throughout our framework, but only what is necessary for the back-ends we wish to target. Conveniently, all modern GPUs support similar programming models and input languages. NVidia hardware supports both CUDA [12] and OpenCL [14]. Other hardware vendors, such as AMD and Qualcomm, support only OpenCL. Both OpenCL and CUDA offer comparable interfaces for memory allocation, compilation, and execution of code. Further, the core language for describing computation supported by both OpenCL and CUDA has C-like syntax and semantics.

Programming model portability with CUCL. Now, we discuss how our approach abstracts away the incompatibilities between OpenCL and CUDA.While CUDA and OpenCL share C as a base, they use different syntax for various GPU-programming-specific concepts. We start with the cross-compatible intersection of CUDA and OpenCL to form a language we call *CUCL*. Then, we abstract away the syntactic differences (e.g. CUDA's `threadIdx` versus OpenCL's `get_local_id`) by adding special syntax to CUCL for them (e.g CUCL's `LOC_ID_1D` maps to `get_local_id(0)` in OpenCL and *(threadIdx.x)* in CUDA; see the relevant back-end sources at in `ocl_util.cc` and `nvrtc_util.cc` for a full list. When necessary or desired, though, our framework allows the use of back-end specific languages or features. Of course, use of back-end specific features limits the portability of any code that uses them. Yet, a far more limiting and important issue is that of *performance portability*. While it is convenient to share a common syntax and semantics for computation (i.e. C) across targets, this ensures only functional equivalence. This is very helpful for development, testing, and debugging. However, it does not address our goal of achieving *high efficiency* across all back-ends. Currently, GPU compilers are unable to produce efficient runtime code from high-level, portable descriptions of convolutions. So, we instead aim to minimize the effort needed in order to optimize and specialize (to whatever degree in necessary) operations of interest across our limited set of target back-ends.

ND-Arrays. Our first guiding observation is that the majority of NN operation inputs and outputs are ND-Arrays. Hence, ND-Array specific support, particularly for metaprogramming, forms a cornerstone of our approach. Typically, ND-Arrays consist of a single contiguous block of memory filled with a flat array of elements.

Importantly, in our application, the sizes of all such arrays are known and fixed at the compute graph level. Thus, we may *statically specialize all operations* based on the sizes of their input and output ND-Arrays. All indexing and bounding calculations on such ND-Arrays may be reduced to multiplication, division, and modulo by constants. The resulting expressions are amenable to efficient implementation and various optimizations. Further, in user-written templates, we require that all dimensions of each ND-Array must be named. This use of mnemonic, semantically-significant names for array dimensions helps clarify code using ND-Arrays. By analogy, imagine code that used C structures where each field was simply referred to by index rather than name. Not only do named ND-Array dimensions improve readability, but they are used to implement a form of type checking for all ND-Array arguments. All ND-Array arguments passed to a function must have the same number of dimensions *with the same names* as given in their argument declarations. For example, a function expecting a 4D-Array with dimension names *in_chan:out_chan:y:x* (i.e. a set of filters) could not be passed a 4D-array with dimension names *img:chan:y:x* (i.e. a batch of images).

3.2 Metaprogramming

As discussed earlier, metaprogramming is, by necessity, commonly used to create high efficiency GPU implementations of NN operations. Thus, the novelty of our approach is not merely the usage of metaprogramming, but in the specific design choices made to balance speed, portability, and productivity. We start with allowing the user to write only mildly restricted native GPU code in our CUDA/OpenCL subset language, CUCL. Compared to directly using CUDA or OpenCL, CUCL:

- provides language-neutral idioms to replace those from CUDA and OpenCL, and
- requires all ND-Array function arguments to be decorated with their dimension names, and
- requires access to ND-Array metadata (sizes, strides) to use a special template syntax: %(*myarray_mydim_size*).

Many simpler operations can be directly written as a single *CUCL function template*. To produce OpenCL or CUDA functions from a CUCL function template, the framework: (1) replaces CUCL idioms with OpenCL or CUDA ones, and (2) replaces references to ND-Array sizes and strides with either (at the user's explicit choice) (2a) constants for the specific input ND-Array sizes, or (2b) references to dynamically-passed ND-Array metadata. Typically, we care most about the case where the sizes are replaced with constants, as this gives the most possibility for optimizations and therefor efficiency. However, this does require instantiation of the given CUCL template for *every* unique set of called argument sizes. Sometimes, for a given operation, this is unnecessary for performance, and perhaps even creates prohibitive overhead. Thus, at the user's selection, our framework also allows dynamically passing the sizes and strides of ND-Arrays as automatically-generated function arguments. Note, however, that CUCL code insulates the user from this issue, since the same syntax is used to refer to ND-Array metadata regardless of if it is dynamic or static, allowing easy experimentation with both methods for each ND-Array.

In general, our approach does not require a GPU programmer to learn any new languages. CUCL is simply a set of *optional* idioms to allow portability between OpenCL and CUDA. For metaprogramming, the user can *optionally* write unconstrained C/C++ to generate CUCL using a simple string-based template system. But, to be clear: a user can, as a starting point, take existing OpenCL or CUDA functions and run them inside Boda with only minor changes (to meet the interface of Boda for inputs and outputs), without writing any metacode or using any CUCL idioms. However, to get programming model portability, they will need to update their code to use only shared subset of OpenCL and CUDA. In this case, the CUCL idioms serve to aid the process, as they allow portable usage of various OpenCL/CUDA features that exist in both languages but with different syntax. Similarly, to get performance portability, they will likely need to employ metaprogramming. But, here, Boda's simple string-based metaprogramming system eases the programmer into learning this skill.

Metaprogramming for NN Convolutions. NN convolution can be viewed as generalized matrix-matrix multiplication. In fact, in early approaches, NN convolution was often implemented using BLAS (Basic Linear Algebra Subroutines) library SGEMM (Single-precision General Matrix-Matrix multiply) calls for the bulk of the computation. But, as discussed in Section 5, the use of special-purpose code for NN convolutions is currently the dominant approach. However, writing efficient NN Convolution code is difficult, as it requires:

- large blocks consisting of many moves or multiplies,
- supporting many regimes of input sizes and modes, and
- having fine-grained control over data movement and execution.

All of these issues share a common solution: metaprogramming. With metaprogramming, one can easily write loops at the metacode level to generate long sequences of moves or multiplies. Multiple input regimes can be handled with metacode level case-splits that do not incur runtime overhead. Finally, one can generate specific memory and register indexing patterns without repetitive, error-prone manual effort. Prior efforts have indeed uniformly used metaprogramming to varying degrees to address these issues; see Section 5 for more discussion and a detailed comparison with this work. At a high level, we choose to take a very general and flexible approach to metaprogramming. Rather than use some language-level metaprogramming facility, we choose to directly write code generators in our framework's host language of C++. We use our framework's native support for ND-Arrays at the metacode layer to (when desired) allow code generation to exploit fixed, exact sizes for all inputs and outputs. For example, when cooperatively loading data across threads on a GPU, one must typically employ a loop with a conditional load. If there are N threads loading W words, the loop must iterate $\lceil W/N \rceil$ times. For each iteration, the load must be guarded on the condition that $i * N + thread_id < W$. In CUCL, OpenCL, or CUDA, here is a simplified version of how such a loop might appear:

```
for(int i = 0; i < ((W-1)/N)+1; ++i) {
  int const ix = i*N + thread_id;
  if(ix< W){filts_buf[ix] = filts[ix];}
}
```

However, if N and W are fixed, we know we need exactly $\lceil W/N \rceil$ individual loads. Further, only the last load need be conditional, and then only if (W mod N) is non-zero. In some cases, just making W and N constant may allow the platform-specific compiler to unroll the loop and eliminate unneeded conditionals without additional effort. We show our framework's support for this simple metaprogramming approach here, where we have replaced the W and N variables with template variables that will be expanded to integer string constants:

```
#pragma unroll
for(int i = 0; i < ((%(W)-1)/%(N))+1; ++i) {
  int const ix = i*%(N) + thread_id;
  if(ix<%(W)){filts_buf[ix] = filts[ix];}
}
```

Although simpler metaprogramming approaches (such as C++ templates, discussed more in Section 5) might be sufficient to handle this case, we have observed that the platform-specific compiler often does not successfully unroll the loop and/or remove unneeded conditionals (we provide an example later in this section). In such cases, our framework allows us to smoothly and easily shift more complexity to the metacode level and directly emit the sequence of desired loads. To do this, we move the loop to the metacode level, and replace it entirely with a template variable in the CUCL code:

```
%(filts_buf_loads);
```

Then, at the metacode level, we write code to generate the needed sequence of loads, which is similar in structure to the original loop:

```
string ix_str, load_str;
for(int i = 0; i < ((W-1)/N)+1; ++i) {
  int const max_ix = i*N + (N-1);
  ix_str = str(i*N)+"+thread_id";
  load_str  = "filts_buf["+ix_str+"]";
  load_str += "=_filts["+ix_str+"];";
  if(max_ix >= W){ // need bound check
    load_str = "if("+ix_str+"<"+str(W)
             + "){"+load_str+"}";
  }
  emit( "filts_buf_loads", load_str );
}
```

While metaprogramming clearly adds complexity, the virtue of a string-based C++ approach is simplicity. If the programmer can write GPU-style C code, they can certainly write C (or C++) that *prints* the same GPU-style C code. Then, they can easily *promote* code from the code to the meta-code level to exploit run-time information to specialize the final generated code. And, in the event of errors at the generator level, or for profiling, they can easily inspect the generated code. We argue that, compared to compiler-style approaches, our approach is both valid and one that some fraction of the rare programmers expert in efficient low-level numerical programming favor. Returning to our example, when this metacode is run for the case of (N=96,W=256), the result is exactly the desired sequence of loads:

```
filts_buf[0+thread_id] = filts[thread_id];
filts_buf[96+thread_id] = filts[96+thread_id];
if(192+thread_id<256){
  filts_buf[192+thread_id] = filts[192+thread_id];
}
```

In one case (with N=128,W=512), this approach resulted in 4 assembly-level load instructions. In contrast, a loop-based approach failed to remove the conditional guarding the load, and yielded dozens of instructions in including four conditional jumps. The details of this example are too long to include here, but are available in the Boda source as `test/meta-smem-load-example.txt`.

Further, generation of global-to-shared memory load sequences (where access patterns are critical), and generation of register-blocked, unrolled sequences of fused multiply-adds (which are often hundreds of instructions long) were tasks that significantly benefited from metaprogramming. Although too complex to discuss in detail here, the reader is referred to our full metacode implementation in the Boda source code in `src/cnn_codegen.cc`.

In summary, it is not easy to determine what sequences of C-level code will execute well on a given platform, but our framework aims to make the process easier. Further, metaprogramming allows the programmer to exploit run-time knowledge to make many values (such as sizes, strides, loop bounds, and offsets) constant, and to reduce the usage of loops and conditionals. Generally, this allows the platform-specific compiler to generate more efficient binary code. But, perhaps more importantly, when the compiler fails to automatically generate efficient code, metaprogramming allows for the ability to emit very low-level code, so that the final instruction sequence can be carefully guided. This allows the ability to productively experiment with different compute and memory access patterns without needing to manually rewrite large sections of target-specific code. Access to detailed documentation, disassemblers and/or instruction-level profiling tools for each given platform make this process much more productive. However, it is perhaps when such aids are not available that Boda's ability to speed the cycle of experimentation is most vital.

Now, we discuss our overall meta-programming flow, which includes the framework layers shown in Figure 3.

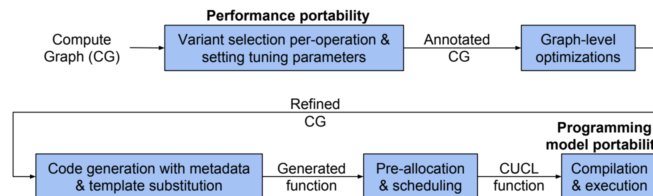

Figure 3. Boda flow: from compute graph to code.

3.3 Variant Selection and Autotuning

As mentioned, NN Convolutions have a wide range of possible input sizes and parameters. It is difficult to write a single function, even with metaprogramming, that runs well over a broad range of input sizes. Furthermore, each back-end target may need specific optimizations, which may be difficult to combine in a single function. Perhaps one target can use a single function for many input sizes, but requires special techniques for memory access. On the other hand, perhaps a range of targets can share code, but only for certain ranges of input sizes. Thus, depending on their specific goals, we expect the user will create multiple *variants* of certain important operations (such as convolution). Further, each *variant* may have various *tuning* parameters that affect code generation, so they can run well in more cases. Such tuning parameters might control thread blocking, memory access patterns, or load/store/compute vector widths. Consider a typical set of tuning parameters and their values: *MNt=4:4,MNb=16:16,Kb=4,vw=4*. These parameters specify 4×4 register blocking, 16×16 thread blocking, an inner-loop-unroll-factor of 4, and a vector/SIMD width of 4. Given an input size and target platform, it may be tractable to manually or heuristically choose a variant and its tuning parameters – particularly when variants are written with specific targets and input size ranges in mind. However, when considering many operations across many input sizes across many target platforms, this task becomes at best onerous and at worst intractable. Thus, an important complimentary technique is *autotuning*, where such parameters can be selected automatically by the framework. By performing a brute-force, guided, or sampled exploration of the space of variants and tuning parameters, we can both: (1) find the best parameters for a given operation, as well as (2) learn much about a new target platform.

Figure 4 demonstrates the key features of autotuning: automatic per-platform variant selection and automated sweeps over tuning parameters. Currently, we apply a simple brute-force search combined with some heuristic parameter selection, which is tractable given the relatively small number of operations, variants, and tuning parameters. For example, in the experimental evaluation of Section 4, which considers 43 operations on 3 targets, we needed to compile and execute a total of 1150 functions. This took on the order of 1 hour, with compilation time being the dominant cost. As

future work, we plan to explore a wider tuning space, over which using brute-force would be intractable. In that case, we plan to use techniques such as those from OpenTuner [1] to limit the number of points in the space that must be tested.

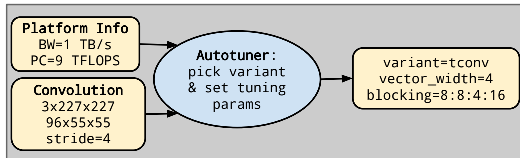

Figure 4. Autotuning in Boda.

3.4 Graph-level Optimizations

Next, we discuss graph-level optimizations: a critical but relatively simple part of our flow. In particular, there are two important graph-level optimizations for NN compute graphs:

- Fusing of adjacent convolution and activation operations, and
- Inserting any needed data-format-conversion operations.

Convolution operations are commonly followed by the application of some element-wise activation function. In some cases, the overhead to read and re-write the output ND-Array to apply the activation function is significant. In these cases, one may inline the code for the activation function in the convolution operation to avoid a read-modify-write of the output. While this may increase the code size of the output-writing part of the convolution operation, it is generally still favorable to do this, as activation functions such as ReLU add only a few instructions per existing output store. So, our framework simply always performs this fusion when possible, using string substitution to insert an application of the activation function for all output-value writes. The insertion of data-format-conversion operations is necessary due to the fact that some variants may use different layouts or padding of their input or output ND-Arrays. That is, since we are able to freely choose the format of most internal ND-Arrays, we can exploit this to achieve higher efficiency within each variant. While the user must generally manually pick data layouts chosen to work well for a given case, the framework's support for ND-Array access and metadata handling eases the burden of creating transformation functions and experimenting with different layouts. Also, as long as different layouts are distinguished by different ND-Array signatures (different dimension cardinality or naming), the framework can error-check that all ND-Arrays are in the proper format prior to each operation. In many cases, data-format-conversion operations can be inserted automatically, based on the context in which an ND-Array is used.

3.5 Code Generation, Scheduling, and Execution

Once we have generated and compiled callable functions for each needed operation, we execute the compute graph. For this, we must first perform operation scheduling and ND-Array allocation. For our current target application, scheduling is not difficult. The bulk of execution time is spent on operations that can each individually saturate the target hardware's compute capacity by themselves. So, we need not attempt to run multiple operations in parallel; any topological sort of the compute graph yields a reasonable execution order. Further, for our current use cases, we are generally not limited by GPU memory. Hence, we can employ a naive allocation strategy and simply pre-allocate all ND-Arrays in the compute graph. However, with some additional work, our framework should be easily capable of supporting more complex scheduling and allocation policies if needed or desired. After allocation and scheduling, we issue the resultant sequence of function calls to the target backend, which in turn performs all the desired computations. The output ND-Arrays are then resident in GPU memory, ready to be read back to the CPU or processed further as desired.

4 Results

We now report per-convolution-operation runtime results across hardware targets and programming models, organized to illustrate the key contributions of our work. The benchmark set of operations was chosen by extracting the unique convolutions from three common DNNs: "AlexNet" [7], "Network-in-Network" [10], and the first version of Google's "Inception" network [16]. Further, we choose to report a selection of 43 operations with:

- a batch size of 5, which models a streaming deployment scenario with some latency tolerance, and
- have more than $1e8$ FLOPS (as we focused our optimization efforts on these more computationally intensive sizes).

As show in Table 1, we organize the operations by sorting them by FLOP count, which is a reasonable proxy for the difficulty of a given operation. However, depending on the exact convolution parameters, two operations with similar FLOP counts may substantially differ in both:

- their theoretical maximum efficiency for a given hardware platform (based on Roofline [19] analysis), as well as
- the empirical performance of any given convolution algorithm.

So, while one expects a general trend that operations with larger FLOP counts will take longer to execute, there is no expectation of smoothness. Of particular note, the two operations with large spikes in runtime in most graphs are *Fully Connected* layers, where each filter is the size of the full input image and thus there is only one output pixel per image. Compared to other convolutions with similar FLOP counts, such operations offer less opportunity for parallelism and data reuse, and thus tend to be slower to execute. However, these fully connected layers can be handled with a faster, less general version of convolution. This special case is not fully implemented in Boda yet, and it appears cuDNN does not properly invoke its specialized version for these cases, perhaps since they are not explicitly marked as fully connected (though this can be easily deduced). Adding optimizations for these special cases to Boda is a good subject for future work.

The NVIDIA GPU used is a Titan-X(Maxwell). The AMD GPU used is an R9-Nano. The Qualcomm GPU used is the Adreno 530 GPU portion of the Snapdragon 820 System-on-Chip (abbreviated "SD820" hereafter). For the CUDA platform, we use the NVIDIA-provided nvrtc library to allow run-time compilation for CUDA. All timings are performed using CUDA and OpenCL level timing functions, and thus should include only time spent on the GPU, and should not depend on the host CPU or other machine configuration details. The input data given to the convolutions is all-non-zero pseudo-random noise. Note that runtimes should not (in general) depend on the input data, as long as it has proper range and sparsity. All outputs are cross-checked for numerical correctness using a hybrid relative/absolute tolerance of 1e-3. It is generally the case that NN calculations do not rely on precision for values with very small magnitudes (i.e. large negative exponents). So, when comparing each value between known-good and under-test implementations, we calculate a relative difference, but clamp it to the maximum absolute value of the two values being compared. Thus, as the values to compare become smaller than the specified relative error tolerance, the tolerance becomes absolute instead of relative. See

KSZ	S	OC	B	$input\ X{\times}Y{\times}Chan$	FLOPs
5	1	32	5	28×28×16	1e+08
5	1	64	5	14×14×32	1e+08
1	1	256	5	7×7×832	1.04e+08
1	1	112	5	14×14×512	1.12e+08
1	1	128	5	14×14×512	1.28e+08
1	1	64	5	28×28×256	1.28e+08
1	1	64	5	56×56×64	1.28e+08
1	1	128	5	14×14×528	1.32e+08
1	1	144	5	14×14×512	1.45e+08
1	1	96	5	28×28×192	1.45e+08
1	1	384	5	7×7×832	1.57e+08
1	1	160	5	14×14×512	1.61e+08
1	1	160	5	14×14×528	1.66e+08
1	1	4096	5	1×1×4096	1.68e+08
1	1	192	5	14×14×480	1.81e+08
5	1	128	5	14×14×32	2.01e+08
3	1	320	5	7×7×160	2.26e+08
1	1	384	5	13×13×384	2.49e+08
1	1	128	5	28×28×256	2.57e+08
1	1	256	5	14×14×528	2.65e+08
1	1	96	5	54×54×96	2.69e+08
3	1	384	5	7×7×192	3.25e+08
3	1	208	5	14×14×96	3.52e+08
1	1	1000	5	6×6×1024	3.69e+08
1	1	1024	5	6×6×1024	3.77e+08
6	1	4096	5	6×6×256	3.77e+08
3	1	224	5	14×14×112	4.43e+08
1	1	256	5	27×27×256	4.78e+08
3	1	256	5	14×14×128	5.78e+08
5	1	96	5	28×28×32	6.02e+08
3	1	288	5	14×14×144	7.32e+08
3	1	128	5	28×28×96	8.67e+08
3	1	320	5	14×14×160	9.03e+08
11	4	96	5	224×224×3	1.02e+09
11	4	96	5	227×227×3	1.05e+09
7	2	64	5	224×224×3	1.18e+09
3	1	1024	5	6×6×384	1.27e+09
3	1	256	5	13×13×384	1.5e+09
3	1	384	5	13×13×256	1.5e+09
3	1	192	5	28×28×128	1.73e+09
3	1	384	5	13×13×384	2.24e+09
3	1	192	5	56×56×64	3.47e+09
5	1	256	5	27×27×96	4.48e+09

Table 1. List of benchmark convolution operations. KSZ: kernel X/Y window size; S: X/Y stride; OC: # of output channels; B: # input images per batch

the function `min_sig_mag_rel_diff()` in `src/boda_base.cc` for the exact implementation.

Programming model portability – OpenCL vs. CUDA. On NVIDIA hardware, we show that we can achieve almost identical per-operation runtime, using the same CUCL code, regardless of which programming interface we use (programming model portability). This is contrary to the common perception that CUDA offers higher performance than OpenCL for NVIDIA hardware. Although this may often be true in practice, the fact that Boda emits only low-level code insulates the user from the differences between OpenCL and CUDA. Instead of using complex programming methods at the level of OpenCL and CUDA, Boda instead shifts much of the implementation complexity into the metacode layer, which is relatively programming platform neutral. Thus, the resulting OpenCL and CUDA code is quite simple and portable, using little beyond basic C constructs and the (common to OpenCL and CUDA) GPU threading model. Also, we abstract away various higher-level issues in terms of compilation, allocation, scheduling, and execution that differ between the two platforms. This is (to the best of the author's knowledge) a novel illustration of the lack of importance of using CUDA versus OpenCL for a high-efficiency, difficult-to-implement GPU programming task. A comparison of CUDA vs. OpenCL efficiency on our benchmark set of operations is given in Figure 5. In the figure, all runtimes are for running each operation using the best function generated by Boda for that operation, selected by autotuning. The two plotted cases differ only in the choice of backend (OpenCL or CUDA) for compilation and execution; the generated CUCL code for both cases is identical. In both the OpenCL and CUDA backends, it is possible to output the compiled "binary" code (in this case, NVIDIA PTX portable assembly code). For several cases that were inspected, the same CUCL source code yields the nearly the same PTX when compiled using either OpenCL or CUDA. However, there are some minor differences: the addressing modes and internal LLVM compiler versions appear to slightly differ between NVIDIA's internal OpenCL and CUDA compilation flows. These issues, combined with normal runtime variation/noise, can easily explain the remaining small differences in runtime between the OpenCL and CUDA cases.

As a gauge of the overall absolute quality of our results, in Figure 6, we show the performance of Boda relative to the highly-tuned vendor CNN library cuDNN version 5. Note that Boda is particularly slower in cases with 3x3 kernel sizes, where cuDNN is using Winograd convolution [8], which is not yet implemented in Boda. A case study to determine the effort/speed tradeoff of implementing Winograd convolution in Boda is a key topic of future work. However, overall, we are reasonably competitive, and even faster than the reference library in a few cases.

Tuning for Qualcomm Mobile GPUs . In Figure 7, the *boda-initial* values show the initial (poor) performance when running the general-case fallback convolution variant on the SD820 platform. When starting work on this platform, the general-case fallback variant was the only variant that could be run, since bugs in the Qualcomm OpenCL implementation and portability issues (primarily related to usage of shared memory and high register usage) prevented any of the existing optimized-for-NVIDIA variants from running at all. The few missing bars in the *boda-initial* series denote cases where even the simple fallback variant failed to compile or run. However, with a few weeks of effort, we were able to create a new convolution variant that both worked around bugs in the Qualcomm platform as well as used some platform-tailored optimizations for memory access. Additionally, based on analysis and experimentation, we added new points in the space of tuning parameters (specific thread and register blocking constants) to be searched over. The final results of using the combination of the new variant and expanded tuning space are shown in the figure as *boda-autotuned*, with the same meaning as in other figures: the values show the runtimes of the best variant and tuning parameters for each operation.

Improving efficiency by autotuning. We now move to some initial results on AMD hardware that demonstrate the value of autotuning. Using the expanded library of variants and tuning space from targeting NVIDIA and Qualcomm hardware, we perform an experiment to isolate the effect of autotuning. In Figure 8, we compare two cases. First, we consider the runtimes one might achieve without autotuning. In this case, it is too time consuming to select

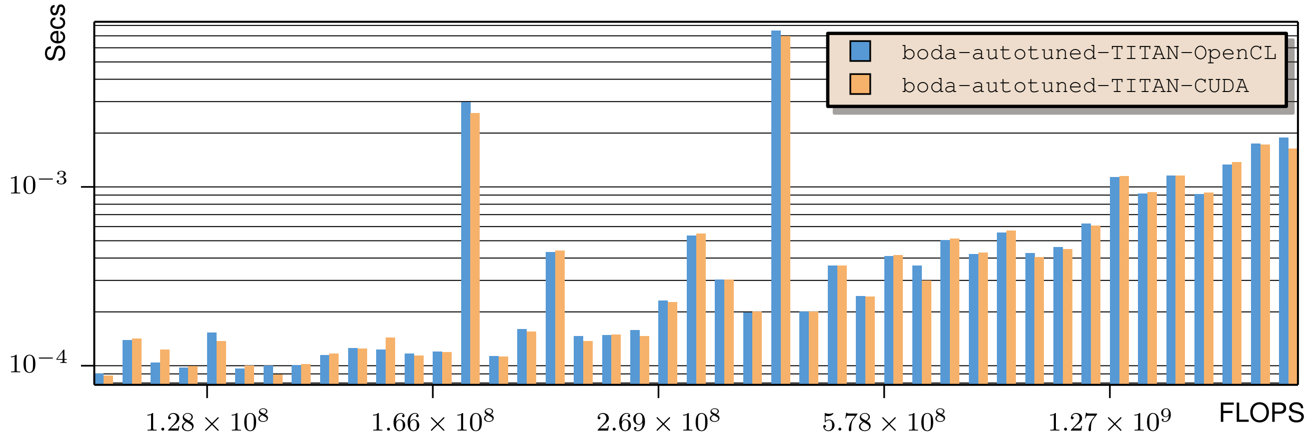

Figure 5. OpenCL vs CUDA. Runtime on NVIDIA Titan-X (Maxwell)

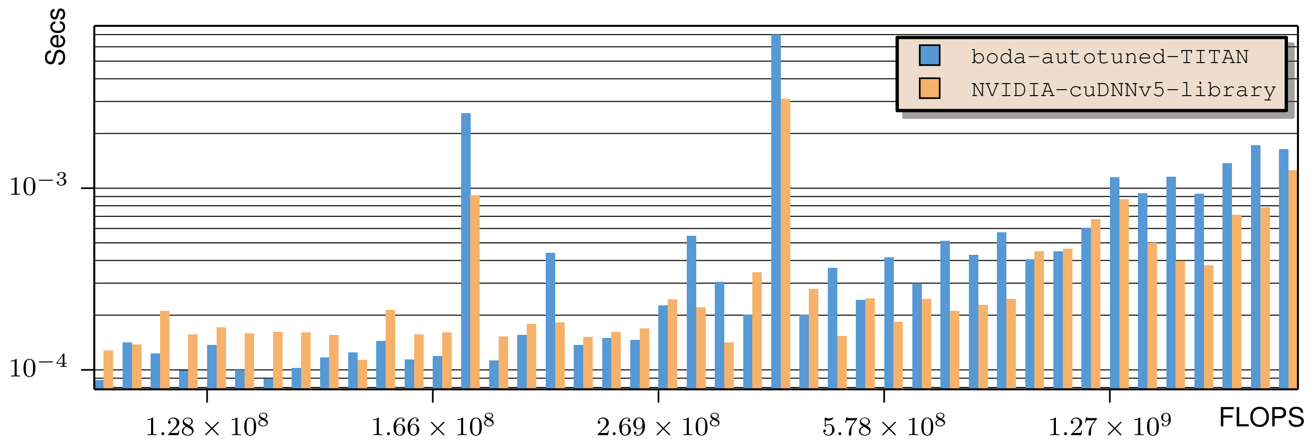

Figure 6. Comparison of Boda with cuDNNv5 on NVIDIA Titan-X

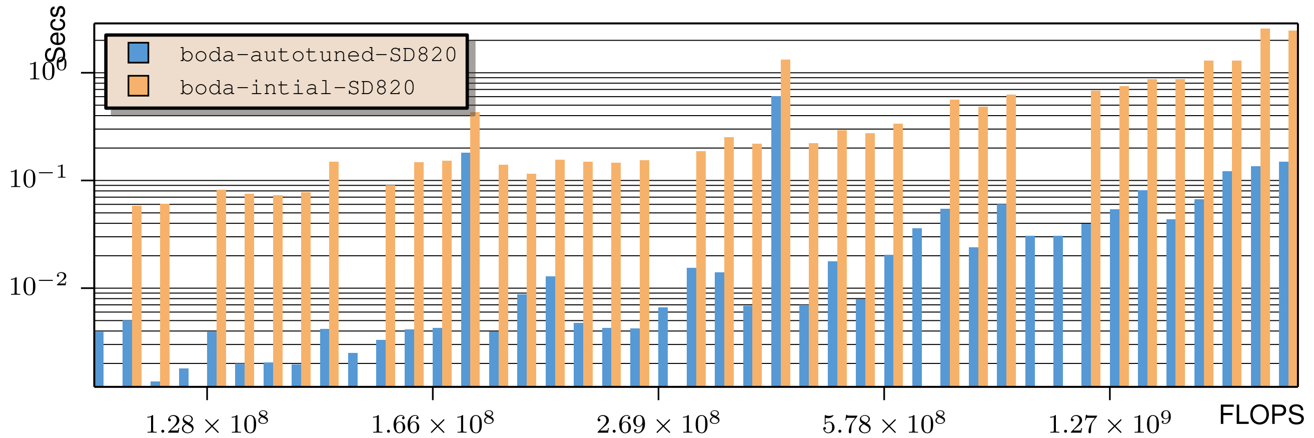

Figure 7. Initial vs. optimized results on Qualcomm Snapdragon 820

the best variant and tuning parameters for each operation individually. Instead, the *boda-manual-tune* values show the runtimes that result from:

- using a simple “choose-most-specialized-possible” heuristic to select the per-operation variant, and
- choosing the *single overall best* setting for tuning parameters, judged by the sum of runtime over all cases.

The second step in this process, while automatic, is designed to mimic the actual process and results of previous efforts at manual tuning that we performed prior to having autotuning support in our framework. Thus, in addition to giving better results, autotuning requires *much less* effort than manual tuning. Additionally, the overall result of exploring the tuning space provides significant insight into this new platform. By seeing which variants and tuning parameter settings work well, *and which do not*, and comparing results across platforms, we can more quickly determine where to focus future optimization efforts. As with all new platforms, it is difficult to predict how much speed improvement is possible with a given amount of optimization effort. However, we are now well positioned to explore this question for the AMD platform as future work.

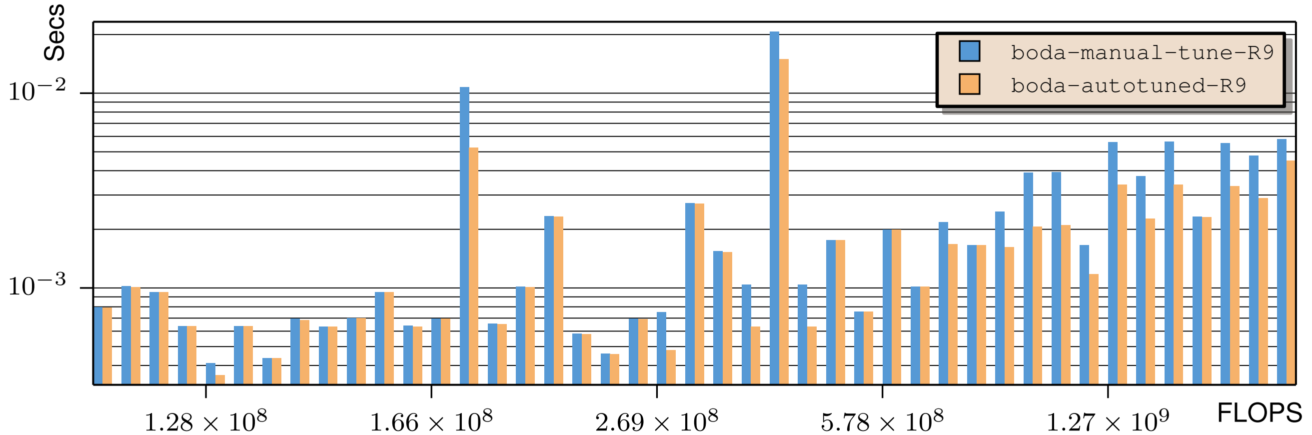

Figure 8. Manually-tuned and autotuned runtime on AMD R9-Nano (Fiji)

Performance portability on different targets. In Figure 9, we show the overall portability of our benchmark convolution operations across three different platforms. Using a single framework and library of variants and tuning parameters, we achieve reasonable performance across three different hardware platforms (AMD, NVidia, and Qualcomm) and different two programming platforms (OpenCL and CUDA). Note that the generated code has no dependencies on any platform-specific libraries (or any libraries at all), and all code is generated and compiled at run-time specific to each operation instance. In particular, for testing, the framework can run the same operation on all platforms supported *within a single process* and compare full results across platforms on the fly. Currently, the results for the AMD platform are significantly slower than those on the NVIDIA platform, especially for the smaller (lower FLOP count) operations. OpenCL is presented as a standard for portable parallel computing across many types of hardware. This leads to a common perception that OpenCL provides *general* (both functional and performance) portability. However, these results clearly demonstrate that, for these operations, OpenCL does not provide performance portability even between two relatively similar platforms (AMD and NVIDIA) with comparable peak computational and memory performance. Of course, the intent of Boda is to allow programmers to close this portability gap, and proving that this can be done for the AMD platform is an important topic for future work. Similarly, while the SD820 results are much slower than the NVIDIA results (by perhaps 2 orders of magnitude), it must be remembered that the SD820 GPU is (by design) a much smaller device with much lower power usage and correspondingly lower peak performance. At this time, we present these results mainly to show the functional portability of our entire framework, including testing and profiling, and not to directly compare these platforms. However, with modest additional optimization efforts on the AMD and Qualcomm platforms, one may be able to draw fairer comparisons between these disparate platforms.

5 Related Works

Early NN frameworks such cuda-convnet [7] and Caffe [5] performed CNN convolutions by leveraging Nvidia's cuBLAS [11] matrix math (BLAS) library. However, this BLAS-based approach is limited in that it: (1) does not reuse data between spatially overlapping input windows, (2) sometimes requires expensive input and output transformations to convert 4D-Arrays into 2D matrices, and (3) does not allow fusion of an activation function with the convolution operation. Additionally, the underlying matrix-matrix multiply function may not be well optimized for the problem sizes required.

Various purpose-built libraries to perform NN convolution have improved speed and efficiency over BLAS-based approaches. NVIDIA's popular cuDNN [2] library achieves much higher efficiency than BLAS-based approaches [3], but is closed-source and limited to NVIDIA hardware. Thus, it is not extensible to support new operations or to target other hardware platforms.

With similar performance to cuDNN, a more open family of libraries based on an assembly-language-level metaprogramming flow is embodied in Nervana System's "neon" framework [9] [15]. However, as with cuDNN, this approach is limited to NVIDIA hardware. Further, the use of perl-based metaprogramming to generate low-level GPU assembly code creates significant hurdles to extending this approach for new operations or platforms. We operate instead at the higher abstraction level of CUCL, and use a C++-hosted string-template based metaprogramming approach. We argue that our approach of writing C++ code that generates C code is relatively easier to work with and extend than writing perl to generate assembly. In particular, using C, many constructs look roughly the same at the metacode and code levels. As shown in the example in Section 3, to statically unroll a loop, one simply moves the loop from the code to the metacode, and "escapes" the body of the loop to print the code it previously contained. In essence, we claim the similarity and compatibility between the metacode and code languages eases the burden on the programmer to operate across both levels. Further, rather than simply creating a convolution library, we span the entire flow from compute graph to execution, which allows for additional freedom and optimizations.

One common approach to metaprogramming is to use built-in language level metaprogramming facilities. In particular, C++ templates are commonly used for high performance GPU metaprogramming with CUDA. However, C++ templates have the following disadvantages as compared with this work:

- C++ template support for OpenCL is only starting to become available.
- All C++ template programs must run at compile time, and thus cannot use run-time information.
- Like perl, C++ templates are a different language than C, and are generally considered difficult to use.
- C++ templates do not offer the practical ability to implement complex, significant operations tasks at the meta level.
- C++ templates do not allow the ability to inspect the generated C level code for a given instantiation for debugging and analysis.

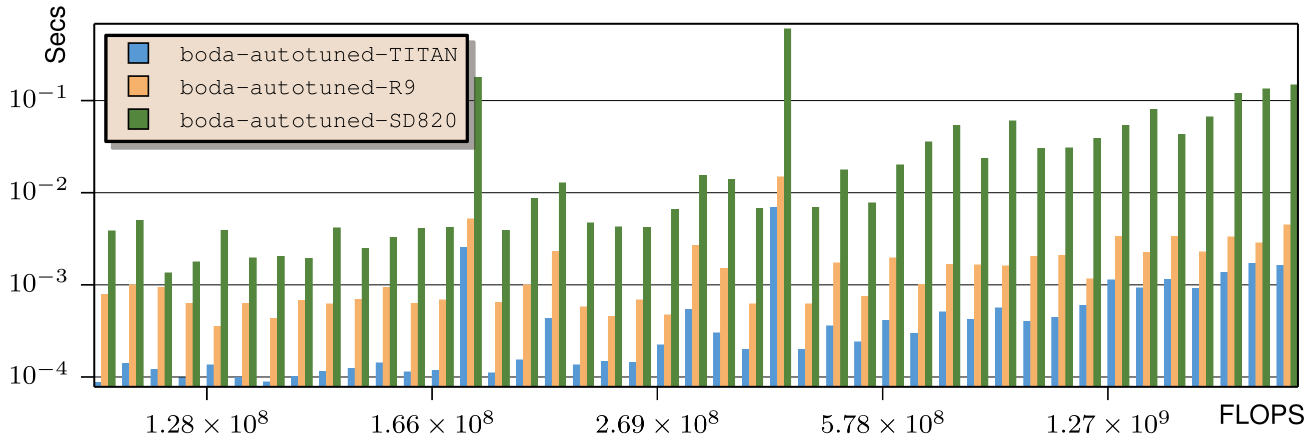

Figure 9. Autotuned runtime on NVIDIA Titan-X, AMD R9-Nano, and Qualcomm Snapdragon 820

In general, there is a lack of mature related work with which to compare our work against, especially for targeting mobile GPUs. In particular, Greentea LibDNN [18] and cltorch [13] do not have published results for mobile GPUs, and it is not clear that they even support such platforms. Further, the lack of maturity of the codebases makes them difficult to independently benchmark. Again, we do admit that such comparisons are important and are a good subject for future work and collaboration if available. Also, based on published results [13] [3], cltorch and Greentea do not appear to be currently competitive with cuDNN on NVIDIA platforms (unlike this work).

On the topic of programming model portability, any comparison must consider both performance portability and programming model portability at the same time, which requires a common benchmarking methodology. The OpenCL-based cltorch project also provides a comparison with a similar CUDA based approach (CUDA torch). However, being separate projects, this comparison does not imply programming model portability for cltorch – nor is the speed of cltorch and CUDA torch particularly close [13].

One compiler-style approach, Latte [17], focuses more on front-end generality and the ability to support arbitrary NN code. However, since it targets Intel CPUs and accelerators, as opposed to popular GPUs, direct comparison is difficult.

6 Conclusions

Boda is a new approach for productive development of efficient GPU code for NN operations. In particular, it supports metaprogramming and autotuning to enable programming model and performance portability. Experimental results show that Boda eases the path to portable, efficient implementations. On NVIDIA hardware, we achieve performance competitive with the vendor library using *either* OpenCL or CUDA. On Qualcomm hardware, we show that we can quickly develop new variants and otherwise tune our generated code to achieve reasonable performance on a mobile GPU. On AMD hardware, we show that autotuning and profiling pre-existing code on a new platform provides a good foundation for platform-specific optimization efforts as future work. Further, as an open, vendor-neutral framework, we avoid dependencies on any specific hardware platforms or inextensible vendor libraries. Thus, our framework provides a productive method for implementing existing and new NN operations while targeting various hardware platforms.

Acknowledgments

Research partially funded by DARPA Award Number HR0011-12-2-0016, the Berkeley Deep Drive (BDD) Industry Consortium, ASPIRE industrial sponsors and affiliates Intel, Google, Hewlett-Packard, Huawei, LGE, Nvidia, Nokia, Oracle, Samsung and German Academic Exchange Service (DAAD).

References

[1] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 303–316.

[2] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *arXiv:1410.0759* (2014).

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

[4] Ross Girshick, Forrest Iandola, Trevor Darrell, and Jitendra Malik. 2015. Deformable part models are convolutional neural networks. In *Computer Vision and Pattern Recognition*.

[5] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv:1408.5093* (2014).

[6] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. 2014. Large-scale video classification with convolutional neural networks. In *Computer Vision and Pattern Recognition*.

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

[8] Andrew Lavin. 2015. Fast algorithms for convolutional neural networks. *arXiv preprint arXiv:1509.09308* (2015).

[9] Andrew Lavin. 2015. maxDNN: An Efficient Convolution Kernel for Deep Learning with Maxwell GPUs. *arXiv:1501.06633* (2015).

[10] Min Lin, Qiang Chen, and Shuicheng Yan. 2013. Network In Network. *arXiv:1312.4400* (2013).

[11] NVIDIA. 2016. cuBLAS. https://developer.nvidia.com/cublas. (2016). [Online; accessed 27-May-2016].

[12] NVIDIA. 2016. CUDA. https://developer.nvidia.com/cuda-zone. (2016). [Online; accessed 01-Oct-2016].

[13] Hugh Perkins. 2016. cltorch: a Hardware-Agnostic Backend for the Torch Deep Neural Network Library, Based on OpenCL. *arXiv preprint arXiv:1606.04884* (2016).

[14] John E Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering* (2010).

[15] Nervana Systems. 2016. 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. 2014. Going Deeper with Convolutions. *arXiv:1409.4842* (2014).

[17] Leonard Truong, Rajkishore Barik, Ehsan Totoni, Hai Liu, Chick Markley, Armando Fox, and Tatiana Shpeisman. 2016. Latte: a language, compiler, and runtime for elegant and efficient deep neural networks. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 209–223.

[18] Fabian Tschopp. 2015. Efficient convolutional neural networks for pixelwise classification on heterogeneous hardware systems. *arXiv preprint arXiv:1509.03371* (2015).

[19] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* (2009).