Comparing the costs of abstraction for DL frameworks

Maksim Levental*
mlevental@uchicago.edu
University of Chicago

Elena Orlova
eorlova@uchicago.edu
University of Chicago

ABSTRACT

High level abstractions for implementing, training, and testing Deep Learning (DL) models abound. Such frameworks function primarily by abstracting away the implementation details of arbitrary neural architectures, thereby enabling researchers and engineers to focus on design. In principle, such frameworks could be "zero-cost abstractions"; in practice, they incur translation and indirection overheads. We study at which points exactly in the engineering life-cycle of a DL model the highest costs are paid and whether they can be mitigated. We train, test, and evaluate a representative DL model using PyTorch, LibTorch, TorchScript, and cuDNN on representative datasets, comparing accuracy, execution time and memory efficiency.

ACM Reference Format:
Maksim Levental and Elena Orlova. 2020. Comparing the costs of abstraction for DL frameworks. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Deep Learning (DL) frameworks represent neural network models as dataflow and computation graphs (where nodes correspond to functional units and edges correspond to composition). In recent years, there has been a proliferation of DL frameworks [**? ? ? ?**] implemented as domain-specific languages (DSLs) embedded in "high-level" languages[1] such as Python, Java, and C#. These DSLs serve as *abstractions* that aim to map the DL graphs to hardware pipelines. That is to say, they hide (or *encapsulate*) details of DL models that are judged to be either irrelevant or too onerous to consider. By virtue of these design decisions the frameworks trade-off ease-of-use for execution performance; quoting the architects of PyTorch:

> *To be useful, PyTorch needs to deliver compelling performance, although not at the expense of simplicity and ease of use. Trading 10% of speed for a significantly simpler to use model is acceptable; 100% is not.*

Trading off ergonomics for performance is manifestly reasonable[2], especially during the early phases of the DL engineering/research process (i.e. during the hypothesis generation and experimentation phases). Ultimately one needs to put the DL model into production. It is at this phase of the DL engineering process that every percentage point of execution performance becomes critical. Alternatively, there are many areas of academic DL where the research community strives to incrementally improve performance [**? ? ?**]. For example, in the area of super-resolution a deliberate goal is to be able to "super-resolve" in real-time [**?**]. Similarly, in natural language processing, where enormous language models are becoming the norm [**?**], memory efficiency of DL models is of the utmost concern. In such instances it's natural to wonder whether ease-of-use trade-offs that sacrifice execution performance, or memory efficiency, are worthwhile and whether their costs can be mitigated.

Thus, our intent here is to investigate the costs of some of the abstractions employed by framework developers. In particular we focus on the PyTorch ecosystem (chosen for its popularity amongst academic researchers) deployed to Graphics Processing Units (GPUs). To that end, we implement a popular and fairly representative[3] DL model at four levels of abstraction: conventional PyTorch, LibTorch, cuDNN, and TorchScript. We argue, in the forthcoming, that these four implementations span considerable breadth in the abstraction spectrum. Furthermore we train, test, and evaluate each of the implementations on four object detection datasets and tabulate performance and accuracy metrics.

The rest of this article is organizing as follows: Section (2) quickly reviews the germaine background material on GPUs and DL frameworks, **??** describes the implementations and our profiling methodology, **??** presents our results and a comparative discussion thereof, **??** discusses broad lessons learned, **??** proposes future work, and **??** speculates wildly about the future of DL systems more generally.

2 BACKGROUND

2.1 Abstraction

subsec:abstraction

2.2 GPUs

We briefly review NVIDIA GPUs[4] in order that the performance criteria we measure in **??** are legible.

A GPU consists of many simple processors, called streaming multiprocessors (SMs), which are comprised by many compute *cores* that run at relatively low clock speeds[5]. Each compute core in

*Both authors contributed equally to this work.
[1]For the purposes of this article, we take "high-level" to mean garbage collected and agnostic with respect to hardware from *from the perspective of the user.*

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[2]"The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming." [**?**]
[3]In the sense that the functional units constituting the model are widely used in various other models.
[4]A more comprehensive introduction to GPUs themselves and CUDA programming is available in [**?**].
[5]For example, individual NVIDIA GTX-1080 Ti cores run at ∼1500MHz.

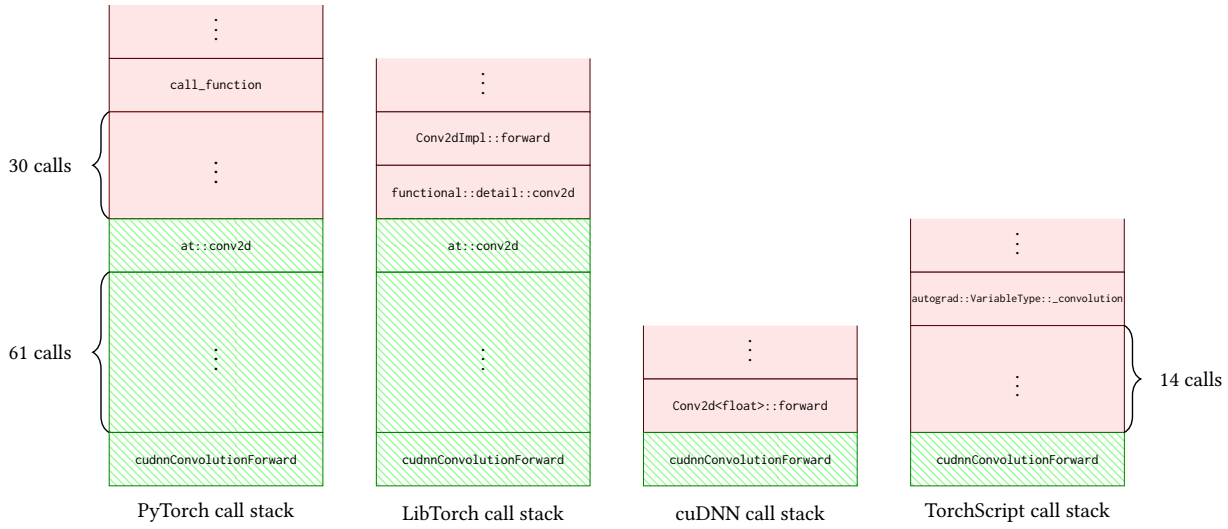


Figure 1: Call stacks representing the number of calls between `Conv2d.forward` at the level of abstraction and the ultimate execution of the convolution `cudnnConvolutionForward` on the GPU. ☐ represents calls where the implementations diverge and ▨ represents calls where the implementations coincide. Note that program setup calls are omitted. Complete stacktraces are available on GitHub at main/tex/stack_traces.

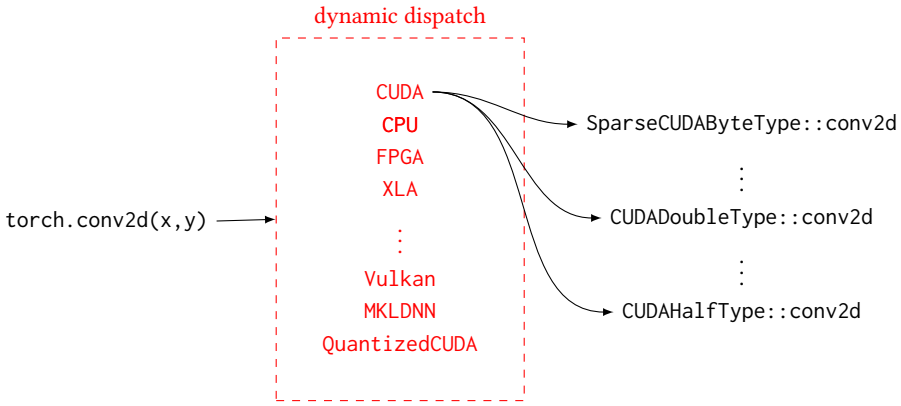


Figure 2: How the `torch.conv2d` operation on tensors `x, y` is implemented in PyTorch.

an SM can execute one floating-point or integer operation per clock cycle. See Figure (3) for a diagram of NVIDIA's Fermi architecture, where each SM consists of 32 cores, 16 load/store (LD/ST) units, four special-function units (SFUs) which compute transcendental functions (such as sin, cos, exp), a relatively large register file[6], and thread control logic (to be discussed in the proceeding). Each SM has access to local memory, several cache levels, and global memory. In the Fermi architecture (and subsequent architectures) local memory is configurable in software; a fraction of it can be apportioned as either local memory or L1 cache (for workloads that query global memory in excess of local memory). One final feature worth mentioning, though irrelevant for us here, is the L2 cache's atomic `read-modify-write` facilities; this enables sharing data across groups of threads more efficiently than possible in conventional CPUs[7].

Such an architecture, particularly suited to maximizing throughput, necessitates a programming model distinct from that of a conventional, general purpose processor architecture. A unit of computation deployed to a GPU is called a *kernel*; kernels can be defined using NVIDIA's Compute Unified Device Architecture (CUDA) extensions to C, C++, and FORTRAN[8]. Compiled kernels are executed by many *threads* in parallel, with each thread starting at the same

[6]For example, Intel's Haswell architecture supports 168 integer and 168 floating-point registers.

[7]On a CPU, atomic `test-and-set` instructions manage a semaphore, which itself manages access to memory (therefore incurring a cost of at least two clock cycles).

[8]In fact CUDA compiles down to a virtual machine assembly code (by way of `nvcc`) for a virtual machine called the Parallel Thread Execution (PTX) virtual machine. So, in effect, it's compilers all the way down.

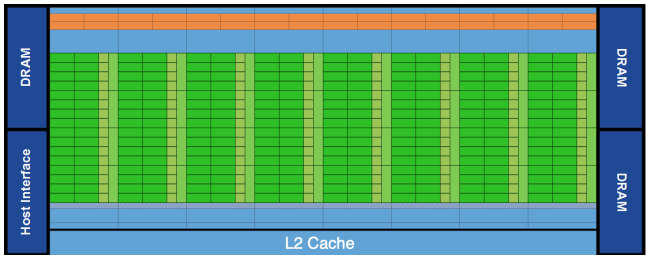
instruction; NVIDIA describes this addition to Flynn's taxonomy [**?**] as Single Instruction Multiple Thread (SIMT)[9]. The large register file enables very fast thread context switching (~25 microseconds on the Fermi architecture [**?**]), performed by a centralized hardware thread scheduler. Multiple threads are grouped into blocks (SMs are single tenant with respect to blocks) and blocks are grouped into *grids* (grids execute a single kernel). All threads in a block, by virtue of running on the same SM, coordinate (execute in arbitrary order, concurrently, or sequentially) and share memory. Thread blocks are partitioned into *warps* of 32 threads; it is these warps that are dispatched by the warp scheduler (see Figure (3b)) and starting with the Fermi architecture two warps can be executed concurrently on the same SM in order to increase utilization[10].

We present an example CUDA program in Figure (4) to illustrate some of the artifacts of the CUDA threading model. The premise of the program is performing an element-wise sum of two 32×48 entry matrices. Note that all of the data weighs in at $3 \times 32 \times 48 \times 4 = 18$ kilobytes (well within the bounds of shared memory on any one SM). The actual work of summing is partitioned across a grid of 6 thread blocks, each containing 16×16 threads. Such a partitioning means each thread can be logically responsible for exactly one sum and therefore the kernel is quite simple (see Figure (4a)). Within the context of a kernel, each thread is uniquely identified by it's multi-index in the thread hierarchy (`threadIdx` and `blockIdx`). Hence, to carry out the sum, the kernel maps this multi-index to the physical address of the data[11]. This (grid, block, thread)-to-data mapping is, in effect, the mechanism that implements the SIMT architecture. Note that, since each block is allocated to exactly one SM, this sum will take $(16 \times 16) \div 16 = 16$ clock cycles on the Fermi architecture; better throughput could be achieved by increasing the number of blocks (and therefore the number of SMs assigned work).

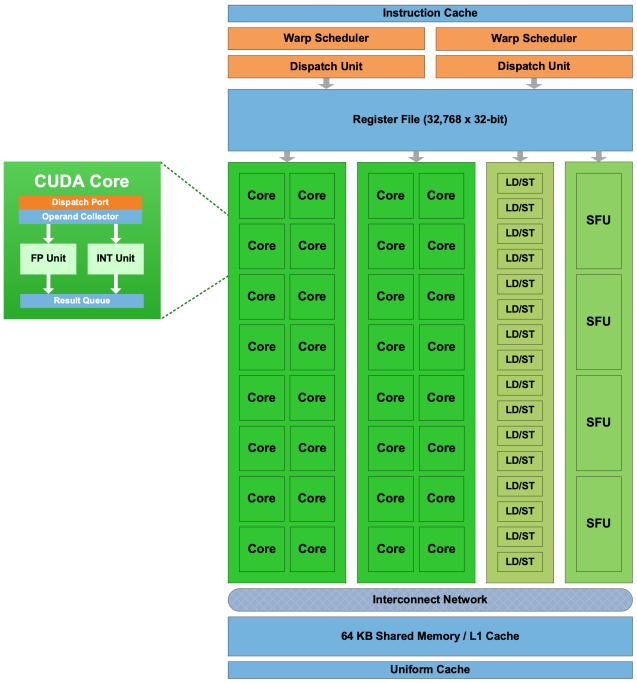
2.3 Graph compilers

DL frameworks primarily function as graph compilers; they compile abstract dataflow and compute graphs into sequences of operations that execute on various hardware architectures. They typically also implement some "quality of life" abstractions like `Tensor`[12] and include utilities useful for the training of DL models (e.g. optimizers and data loaders). Here we focus in particular on the graph compiler functionality.

DL graph compilers are distinct from other dataflow compilers (such as VHDL and Verilog[13]); in addition to keeping account of how the data streams through the compute graph, they also keep account of how the gradients of the data stream through the graph (i.e. the *gradient-flow*). This is called *automatic differentiation* (often



(a) Eight (of 16) SM in the Fermi architecture (remaining 8 are symmetrically placed around the L2 cache)



(b) An individual Fermi SM

Figure 3: NVIDIA Fermi Architecture [?]

shortened to *autodiff*). In principle autodiff is implemented by using the rules of Newton's calculus to calculate the derivatives of primitive functions and the chain rule to calculate derivatives of compositions of primitive functions. There are two types of autodiff: *forward mode* (or *forward accumulation*) and *reverse mode* (or *reverse accumulation*)[14]. Reverse mode autodiff enables the framework to effectively calculate the gradients of parameters of a neural network with respect to some relevant loss or objective function. Note that such gradients can be *back-propagated* through the neural network in order to adjust the parameters of the neural network such that it minimizes the loss[15] or maximizes the objective.

[9] They key difference between SIMD and SIMT is that while in SIMD all vector elements in a vector instruction execute synchronously, threads in SIMT can diverge; branches are handled by predicated instructions [**?**].

[10] That is, one warp can occupy the compute cores while the other occupies the SFUs or Load/Store units.

[11] In CUDA C/C++ data is laid out in row-major order but this is not fixed (in CUDA FORTRAN the data is laid out in column-major order).

[12] A tensor in this context is a data structure similar to a multidimensional array that supports some useful operations (e.g. slicing, flattening, index permutation). Most DL frameworks also abstract memory layout on hardware behind this abstraction.

[13] Verilog and Very High Speed Integrated Circuit Hardware Description Language (VHSIC-HDL or VHDL) are specification languages for specifying circuits on field programmable gate arrays.

[14] Briefly, for a composition of functions $y = f(g(h(x)))$, forward mode evaluates the derivative $y'(x)$, as given by the chain rule, inside-out while reverse mode evaluates the derivative outside-in. For those familiar with functional programming, these operations correspond to `foldl` and `foldr` on the sequence of functions with ∂_x as the operator.

[15] In which case, it is in fact the negatives of the gradients that are back-propagated.

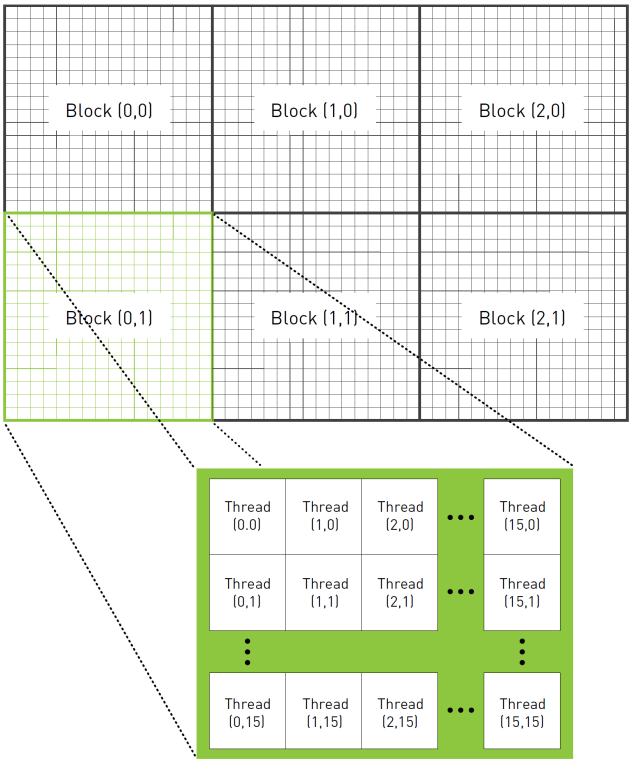
```cuda
__global__ void matrix_sum(
    float *A,
    float *B,
    float *C,
    int rows,
    int cols
) {
    // blockDim is short for block dimension
    // blockDim.x == blockDim.y == 16 threads
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    if (x < cols && y < rows) {
        int ij = x + y*m; // row-major order
        C[ij] = A[ij] + B[ij];
    }
}

int main() {
    int rows = 32, cols = 48;
    float A[m][n], B[m][n], C[m][n];

    // initialization and cudaMemcpy
    // ...

    // dim3 is a 3d integer vector
    // dimensions omitted in the constructor
    // (e.g. z like here) are set to 1
    dim3 numBlocks(3, 2);
    dim3 numThreads(16, 16);
    matrix_sum<<blocks, threads>>(A, B, C, rows, cols);
}
```

(a) CUDA code to be compiled by nvcc. Note differences `__global__` and `matrix_sum<<,>>` from standard C.



(b) Mapping from thread and block to matrix element [?].

Figure 4: Canonical CUDA "hello world" kernel (matrix addition).

Dataflow graphs (and their corresponding gradient-flow graphs) can be specified either statically, with fan-in and fan-out for all functions predetermined, or dynamically, where compositions of functions are determined "on-the-run". There are advantages and disadvantages to both specification strategies. Static specifications tightly constrain[16] the intricacy of the dataflow graph but, obversely, can be leveraged to improve performance and scalability [? ?]. TensorFlow (prior to v2.0) is an example of a DL framework that compiles statically specified graphs. Conversely, dynamic specifications can be very expressive and user friendly, including such conveniences as runtime debugging, but are much more difficult to optimize. PyTorch is an example of a DL framework that supports dynamic specification. Both PyTorch and TensorFlow also support just-in-time (JIT) compilation strategies (TorchScript and XLA respectively); such JIT compilers strike a balance between fluency and scalability. In this work we investigate TorchScript (see ??).

It warrants mention that, in addition to vertically integrated DL frameworks (i.e. specification language and hardware compiler), recently there has been work on intermediate byte code representations for dataflow graphs that arbitrary compiler "frontends" can target. The Multi-Level Intermediate Representation (MLIR) [?] project has goals that include supporting dataflow graphs, optimization passes on those graphs and hardware specific optimizations[17]. Stripe [?] is a polyhedral compiler[18] that aims to support general machine learning kernels, which are distinguished by their high parallelism with limited mutual dependence between iterations. Tensor Comprehensions [?] is an intermediate specification language (rather than intermediate byte code representation) and corresponding polyhedral compiler; the syntax bears close resemblance to Einstein summation notation and the compiler supports operator fusion and specialization for particular data shapes. Finally, Tensor Virtual Machine (TVM) [?] is an optimizing graph compiler that automates optimization using a learning-based cost modeling method that enables it to efficiently explore the space of low-level code optimizations.

[16] For example, branches and loops are cumbersome to specify statically.
[17] Interestingly enough, the project is headed by Chris Lattner who, in developing LLVM, pioneered the same ideas in general purpose programming languages.
[18] A polyhedral compiler models complex programs (usually deeply nested loops) as polyhedra and then performs transformations on those polyhedra in order to produce equivalent but optimized programs [?].