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, 14 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 [1–4] 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 (see section (2.1) for a more comprehensive discussion on abstraction in computer science). 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. [1]

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 [6–8]. For example, in the area of super-resolution a high-priority goal is to be able to "super-resolve" in real-time [9]. Similarly, in natural language processing, where enormous language models are becoming the norm [10], memory efficiency of DL models is of the utmost concern. In such instances it is 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 organized as follows: section (2) discusses abstraction and quickly reviews the germaine background material on GPUs and DL frameworks, section (3) describes the implementations and our profiling methodology, section (4) presents our results and a comparative discussion thereof, section (5) discusses broad lessons learned, section (6) concludes and proposes future work, and section (7) speculates wildly about the future of DL systems more generally.

2 BACKGROUND

2.1 Abstraction

What is abstraction? In fact, there are several closely related notions of abstraction. First there is the philosophical notion of abstraction; Locke defines abstraction as follows (bolding ours):

> The acts of the mind, wherein it exerts its power over simple ideas, are chiefly these three: …The third is **separating them from all other ideas that accompany them in their real existence**: this is called abstraction … [11]

*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 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." [5]
[3]In the sense that the functional units constituting the model are widely used in various other models.

Then there is mathematical abstraction; Russell defines abstraction as follows:

> This principle [of abstraction] asserts that, whenever a relation, of which there are instances, has the two properties of being symmetrical and transitive, then the relation in question is not primitive, but is analyzable into sameness of relation to some other term; and that this common relation is such that there is only one term at most to which a given term can be so related, though many terms may be so related to a given term. [12]

Intriguing as these notions of abstraction may be, they are distinctly different from the notion of abstraction in computer science; in particular with respect to mathematical abstraction (bolding ours):

> ...the primary product of mathematics is *inference structures*, while the primary product of computer science is *interaction patterns*. This is a crucial difference, and it shapes their use of formalism and the kind of abstraction used in the two disciplines.
>
> ...computer science is distinguished from mathematics in the use of a kind of abstraction that computer scientists call *information hiding*. The complexity of behaviour of modern computing devices makes the task of programming them impossible without abstraction tools that hide, but do not neglect, **details that are essential in a lower-level processing context but inessential in a [particular] software design and programming context**. [13]

This understanding of abstraction is widely agreed upon; notably Abelson, Sussman, and Sussman in their much revered *Structure and Interpretation of Programs*:

> We are not at that moment concerned with how the procedure computes its result, only with the fact that it computes the square. The details of how the square is computed can be suppressed, to be considered at a later time. Indeed, as far as the `good-enough?` procedure is concerned, `square` is not quite a procedure but rather an abstraction of a procedure, a so-called *procedural abstraction*. At this level of abstraction, any procedure that computes the square is equally good. [14]

Thus, abstraction is the modulation of concern for details in accordance with the needs of the user and levels of abstractions are graded by the degree of the elision (bolding ours):

> To specify nontrivial computational processes in machine language is a practical impossibility for humans, and so programming languages with higher levels of abstraction are necessary.
>
> ...At a higher level of [abstraction], a *subroutine*, *function*, or *procedure* is an abstraction of a segment of memory that hides the details of how the segment represents a piece of a program that is passed certain parameter values and returns a value as a result.
>
> ...A *garbage collector* is an abstraction of a special process that collects garbage and makes it once again available to the original program, hiding from that program the details of how this is done.
>
> ...**This use of code libraries is an example of *procedural abstraction***, or the ability to execute code through the calling of named procedures that accept explicitly described parameters and return certain guaranteed results. It is an example of abstraction because the details of how the procedure performs its computation are hidden from the procedure's caller; since the caller only makes use of the procedure for its results, there is no need for it to know the internals. [13]

Taking *information and concern encapsulation* as our operational definition of abstraction, in what sense shall we measure the costs of the abstractions employed by DL frameworks? An immediate candidate measure of cost is the asymptotic time (or space) complexity of various operations and data structures that comprise the abstraction. We claim that, with rare exception[4], asymptotic complexity is a poorly suited measure of the complexity or cost of abstractions in the sense that we here deal with. If the abstraction is truly abstract then it bears no resemblance to the realization (recall Locke's definition of abstraction) and if the abstraction is reified then the analysis becomes completely impractical (owing to the numerous components and operations). Even if such analysis were practicable the result would most likely be uninteresting and inconsequential for actual DL frameworks and their users. It is well known that the constant factors in the complexity and peculiarities of those systems themselves more closely govern performance than the order terms. For example, Quicksort, an $O(n^2)$ sorting routine, outperforms even many $\Theta(n \log n)$ sorting routines because it is more cache efficient [16].

Another way to reason about the cost of abstractions is according to the "zero-overhead" principle as articulated by Bjarne Stroustrup:

> In general, C++ implementations obey the zero-overhead principle: What you don't use, you don't pay for. And further: What you do use, you couldn't hand code any better. [17]

Therefore we make the expedient and practical assumption that what is more interesting and valuable to the DL community than asymptotics is, in fact, an empirical study of the resource efficiency of the abstractions; namely execution time, memory usage, and GPU utilization.

2.2 GPUs

We briefly review NVIDIA GPUs[5] in order that the performance criteria we measure in section (3) 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[6]. Each compute core in

[4] One result does come to mind: Pippenger [15] produces a program that runs in $O(n)$ on an impure LISP but which runs in $\Theta(n \log n)$ a pure LISP.

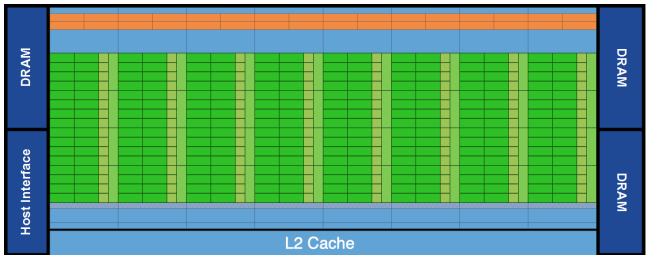
[5] A more comprehensive introduction to GPUs themselves and CUDA programming is available in [18].

[6] For example, individual NVIDIA GTX-1080 Ti cores run at ~1500MHz.

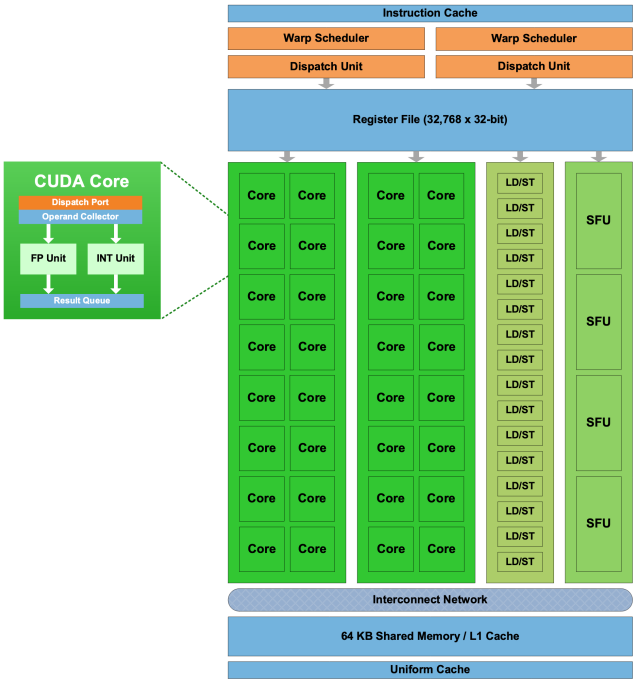
an SM can execute one floating-point or integer operation per clock cycle. See fig. (1) for a diagram of NVIDIA's Fermi architecture, where each SM consists of 32 cores, 16 load/store (LD/ST) units, four special-function units (SFUs) which compute transcendental functions (such as sin, cos, exp), a relatively large register file[7], 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[8].

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[9]. Compiled kernels are executed by many *threads* in parallel, with each thread starting at the same instruction; NVIDIA describes this addition to Flynn's taxonomy [19] as Single Instruction Multiple Thread (SIMT)[10]. The large register file enables very fast thread context switching (~25 microseconds on the Fermi architecture [21]), 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 fig. (1b)) and starting with the Fermi architecture two warps can be executed concurrently on the same SM in order to increase utilization[11].

We present an example CUDA program in fig. (2) 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 six 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 fig. (2a)). Within the context of a kernel, each thread is uniquely identified by its multi-index in the thread hierarchy (`threadIdx` and `blockIdx`). Hence, to carry out the sum, the kernel maps this multi-index to the physical



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



(b) An individual Fermi SM

Figure 1: NVIDIA Fermi Architecture [22]

address of the data[12]. 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 and Tensors

DL frameworks primarily function as graph compilers and tensor abstractions[13]; They typically also include some "quality of life" utilities useful for the training of DL models (e.g. optimizers and data loaders). PyTorch's `Tensor` abstraction is responsible for a

[7] For example, Intel's Haswell architecture supports 168 integer and 168 floating-point registers.
[8] 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).
[9] 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 is compilers all the way down.
[10] 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 [20].
[11] That is, one warp can occupy the compute cores while the other occupies the SFUs or Load/Store units.
[12] 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).
[13] 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.

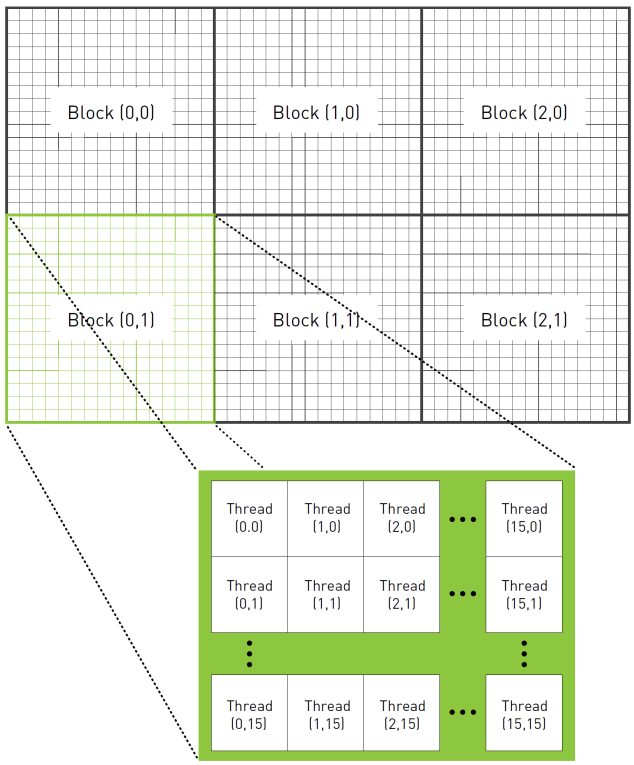
```cuda
__global__ void matrix_sum(
    float *A,
    float *B,
    float *C,
    int rows,
    int cols
) {
    // blockDim is short for block dimension
    // blockDim.x == blockDim.y == 16 threads
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    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 [18].

Figure 2: Canonical CUDA "hello world" kernel (matrix addition).

great deal of the complexity and implementation overhead of the framework. Due to the framework's broad support for hardware and data types, dynamic dispatch[14] is employed to resolve methods on `Tensors` (see fig. (3)). This dynamic dispatch produces deep call stacks for every single operation on a `Tensor` (see fig. (4)); it remains to be seen whether the context switching[15] between function contexts incurs any appreciable execution time penalty.

DL graph compilers are distinct from other dataflow compilers (such as VHDL and Verilog[16]); 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 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*)[17]. 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[18] or maximizes the objective.

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[19] the intricacy of the dataflow graph but, obversely, can be leveraged to improve performance and scalability [24, 25]. 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 section (3)).

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 bytecode representations for dataflow graphs that arbitrary compiler "frontends" can target. The Multi-Level Intermediate Representation (MLIR) [26]

[14] Kernels live in shared-object libraries (e.g. `libcaffe2.so`, `libcaffe2_gpu.so`) and therefore call sites of virtual functions (indirection) are resolved at runtime.
[15] Every function call corresponds to a stack frame allocation and register allocations. In addition indirection to far away call sites leads to poor instruction cache efficiency [23]
[16] 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.
[17] 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.
[18] In which case, it is, in fact, the negatives of the gradients that are back-propagated.
[19] For example, branches and loops are cumbersome to specify statically.

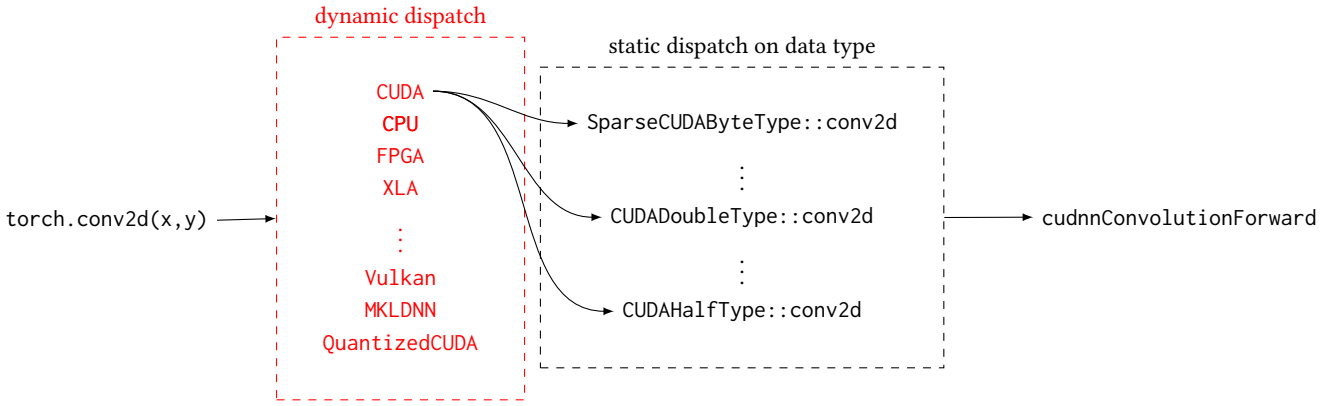


Figure 3: How the `torch.conv2d` operation on tensors `x, y` is implemented in PyTorch.

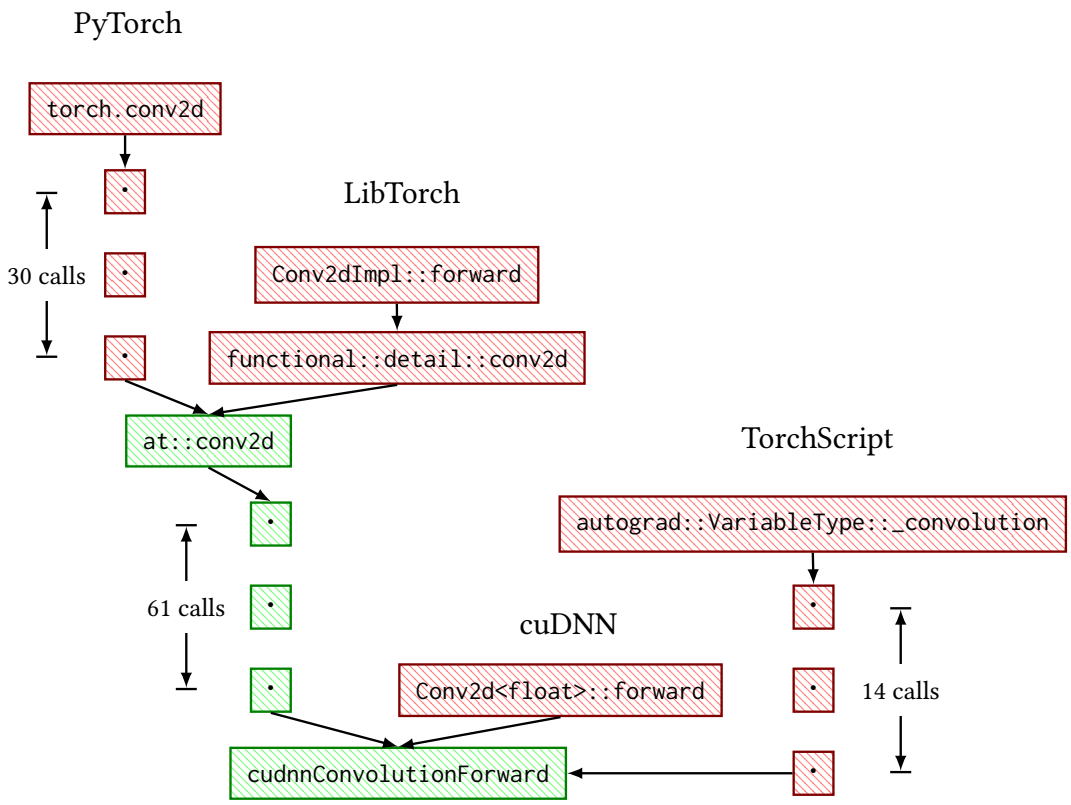


Figure 4: Call graphs representing the number of calls between `Conv2d.forward` at the level of abstraction and the ultimate execution of the convolution `cudnnConvolutionForward` on the GPU. ▭ represents calls where the implementations diverge and ▨ represents calls where two or more implementations coincide. Note that program setup calls are omitted. These were produced by building each implementation with debug symbols and using `gdb` to set a breakpoint at `cudnnConvolutionForward`. Complete stacktraces are available on GitHub at main/tex/stack_traces.

project has goals that include supporting dataflow graphs, optimization passes on those graphs and hardware specific optimizations[20].

Stripe [27] is a polyhedral compiler[21] that aims to support general machine learning kernels, which are distinguished by their

[20]Interestingly enough, the project is headed by Chris Lattner who, in developing LLVM, pioneered the same ideas in general purpose programming languages.

[21]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 [28].

high parallelism with limited mutual dependence between iterations. Tensor Comprehensions [29] is an intermediate specification language (rather than intermediate bytecode 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) [30] 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.

3 METHODS

As discussed in the preceding, the translation from high-level neural network specification to hardware native involves a diverse set of choices at design time and translation time. Any such choice made implicitly by the DL framework abstracts away intermediary details at the level of abstraction at which the choice is made. For example, in PyTorch, by default, convolutions include not only learnable filters but also a learnable bias. Naturally this increases the number of parameters for which gradients need to be kept account of and updated for. At the next level of abstraction (translation from Python specification to C++ objects) another implicit choice is made in choosing tensor dimension ordering[22]. Finally, at the level of abstraction just prior to compilation into CUDA kernels a convolution strategy is chosen[23] according to heuristics. Each of these choices potentially incurs a runtime execution and memory cost, depending on whether the heuristics according to which the choice was made apply to the user's DL model.

With this as subtext, we describe the intent and design of our experiments. We implement the popular object detection deep neural network ResNet-50 [31] at four levels of abstraction (PyTorch, TorchScript[24], LibTorch, and cuDNN) in order to investigate the differences amongst them. We measure accuracy, execution time, GPU utilization, and memory efficiency of each implementation on four image datasets (MNIST, CIFAR10, STL10, PASCAL). The source for the implementations is available on GitHub[25]. The datasets were chosen because they span the spectrum of image complexity (from small single-channel images to large multi-channel images). The reasons for choosing ResNet-50 (see fig. (5)) are two fold. Firstly, it serves as a benchmark architecture in the research community. Secondly, it includes functional units included in many other network architectures (residual units, convolutions of various sizes, batch normalizations, ReLU activations, and pooling layers) and is therefore representative of typical neural network compute workloads. The reason for staying within the same ecosystem is that, in theory, we fix as many of the dimensions of functionality orthogonal to our concerns as possible.

We employ two test platforms (see table (1)); training is performed for 100 epochs with batch size 128 (except for on PASCAL where we use batch size 16) on the "Training platform". Distinct models were trained in parallel in order to expedite the overall training process but neither data parallelism nor intra-model parallelism were employed. In addition we perform scaling experiments (in resolution and batch size) on the PASCAL dataset; for `batch_size`, `resolution` $= 8, 16, \ldots, 1024$. For this purpose we employ the "Resolutions platform", which has GPU RAM enough to accommodate large batch sizes and large image resolutions. For both sets of experiments, each run is repeated 10 times and results are averaged to reduce variance in the measurements. Precise execution time measurements are collected using the CUDA `cudaEventCreate`, `cudaEventRecord`, `cudaEventElapsedTime` APIs. Precise memory and GPU utilization measurements are collected using the NVIDIA Management Library C API. Both sets of APIs were appropriately wrapped for use in Python.

stage	output	ResNet-50
conv1	112×112	7×7, 64, stride 2
conv2	56×56	3×3 max pool, stride 2 [1×1, 64 3×3, 64 1×1, 256] ×3
conv3	28×28	[1×1, 128 3×3, 128 1×1, 512] ×4
conv4	14×14	[1×1, 256 3×3, 256 1×1, 1024] ×6
conv5	7×7	[1×1, 512 3×3, 512 1×1, 2048] ×3
	1×1	global average pool 1000-d fc, softmax
# params.		$\mathbf{25.5}\times10^6$

Figure 5: ResNet-50 network architecture [31]. Note that each convolution is followed by a batch normalization unit and each "stage" is followed by a ReLU (residual connections omitted).

4 RESULTS

The PyTorch implementation compares favorably with both LibTorch and the cuDNN implementations (see fig. (6)) in terms of accuracy. On MNIST and CIFAR10 all three implementations perform reasonably well; LibTorch and PyTorch attain maximum accuracy at around the same time while cuDNN lags behind. On the more complex STL10 and PASCAL datasets (see fig. (14) in the appendix) the cuDNN implementation dramatically underperformed PyTorch and LibTorch. The cause of the difference between the cuDNN implementation and the others is unclear.

In attempting to resolve the poor performance of the cuDNN implementation it was discovered that PyTorch (and LibTorch as well) initialize weights in convolutional, linear, and batch normalization layers. This is not documented and not configurable. For convolutional and linear layers Kaiming uniform initialization [32]

[22] PyTorch uses some heuristics to order tensor dimensions as either NCHW or NHWC.
[23] Winograd convolution, general matrix multiply (GEMM), or FFT convolution.
[24] TorchScript models are serializations of PyTorch models but can run in inference mode in C++, i.e. sans Python runtime.
[25] https://github.com/makslevental/pytorch_abstraction_comparison

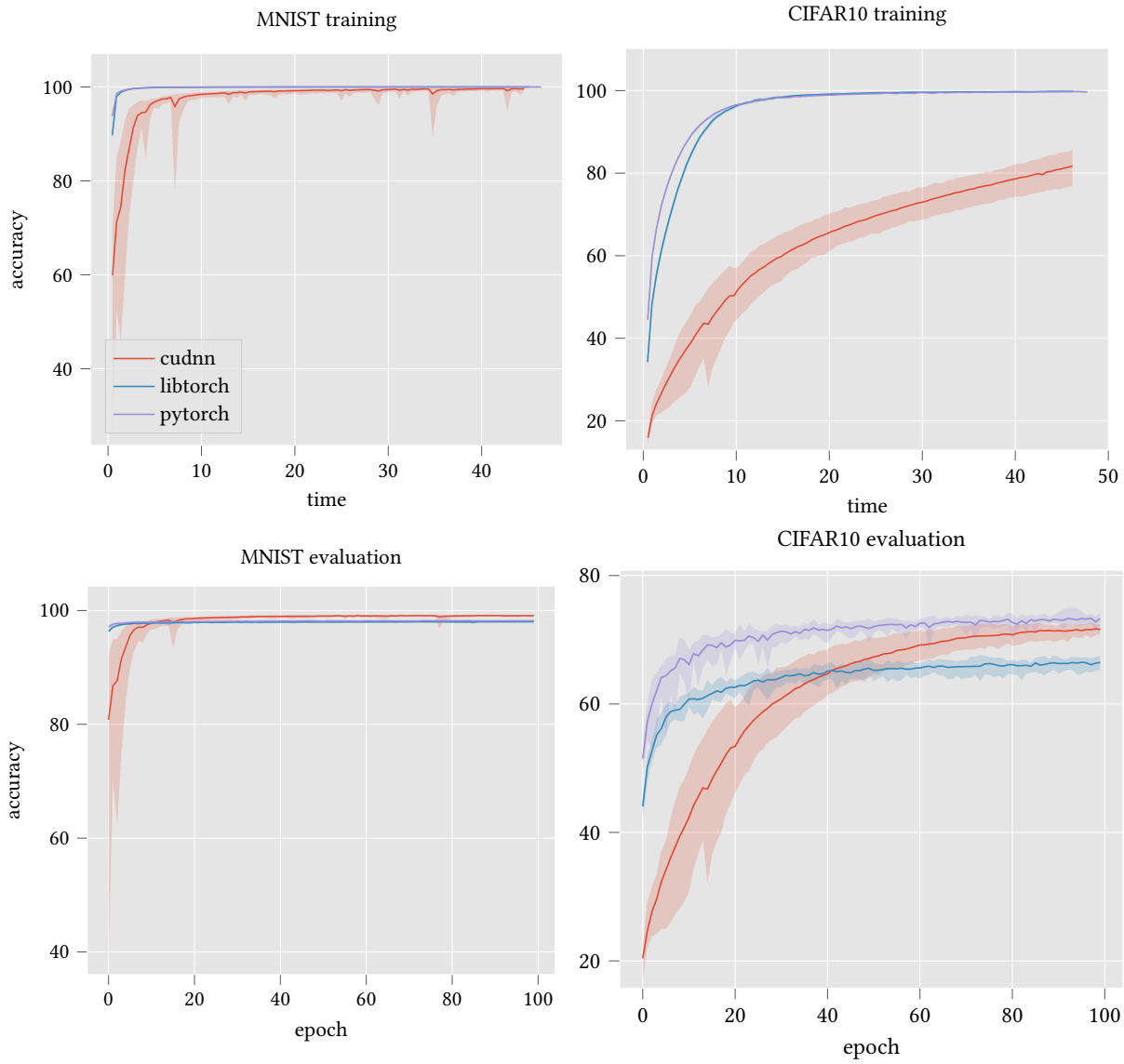


Figure 6: Comparison of accuracy for PyTorch, LibTorch, and cuDNN implementations during training and evaluation. Solid line corresponds to mean while shaded regions correspond to min and max. Time is measured in units of *epoch* × *average epoch time*.

is used and for batch normalization layers ($\gamma = 1, \beta = 0$) initialization is used. This presents a serious problem for us because it is known that **ResNets with Kaiming initializations lead to exploding gradients** [33]. Nonetheless, we implemented Kaiming initialization for the cuDNN implementation but it did not resolve the under-performance issues. Indeed, the network vacillated between vanishing gradients and exploding gradients depending on various settings of the hyper-parameters in the Kaiming initialization. Note that TorchScript training and evaluation accuracy is not measured/reported because TorchScript implementations cannot (as of yet) be trained, only evaluated.

The undocumented initialization leads us to believe that most likely there are several other heuristic optimizations implemented by Pytorch (and LibTorch). While such optimizations generally do improve performance (to wit: here on STL10 and PASCAL) this prompts the question of whether or not this is a "moral" cost of abstraction (since the optimizations might hurt performance for other models [33]).

In terms of execution time and memory usage PyTorch compares unfavorably with each of the other implementations. We measure execution time, memory usage, and GPU utilization during evaluation on PASCAL for various batch sizes and resolution. For example,

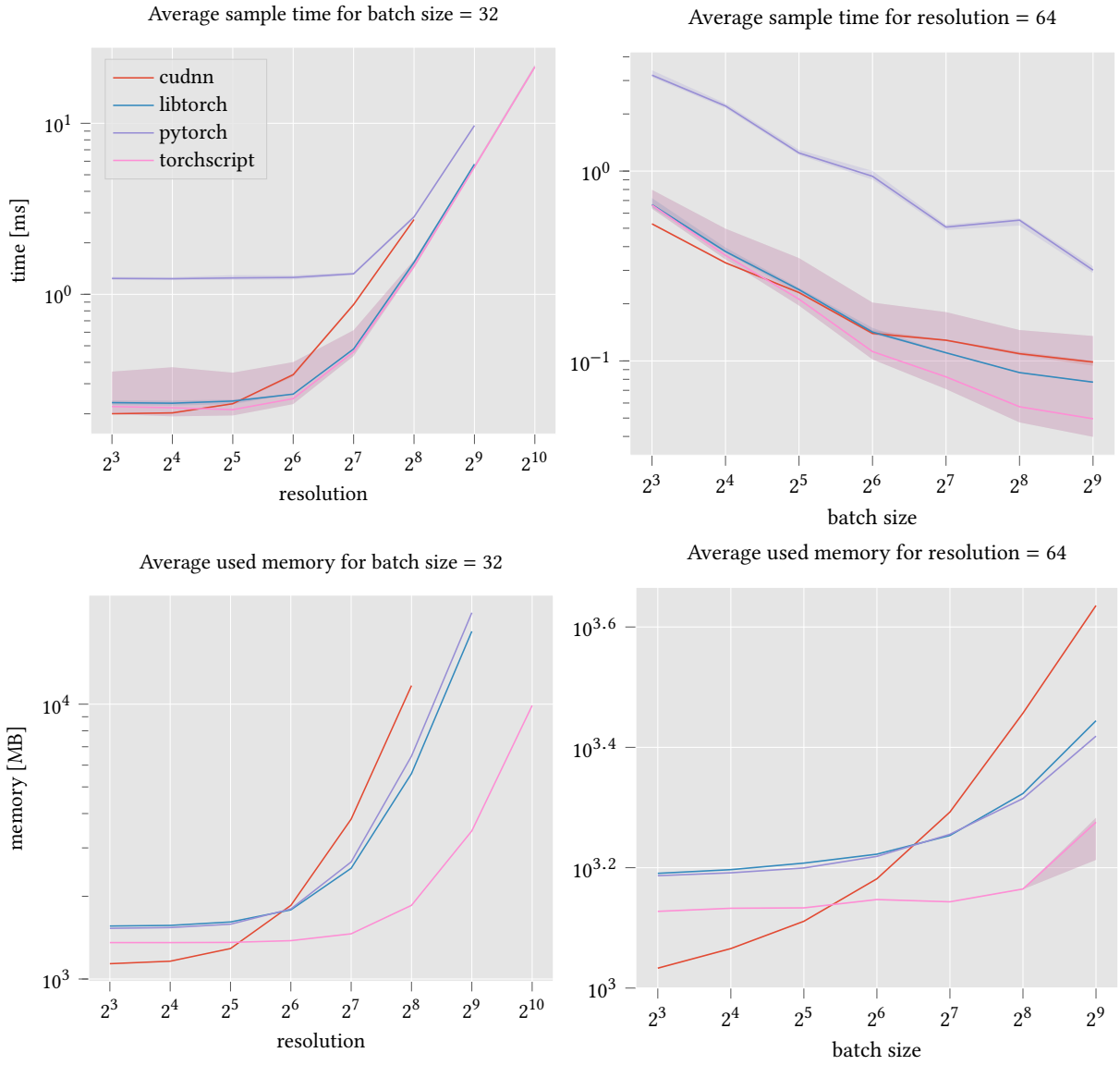


Figure 7: Comparison of execution time and memory efficiency for PyTorch, LibTorch, and cuDNN implementations on PASCAL for fixed batch size (32) and various resolutions and fixed resolution (64) and various batch sizes.

for fixed batch size and various resolutions and for fixed resolution and various batch sizes (see fig. (7)), we see that PyTorch is almost an order of magnitude slower than all other implementations. This execution time difference persists across resolutions and batch sizes but narrows as either increases (see figs. (8), (9), (10), (11), (12) and (13) in the appendix). With respect to memory usage PyTorch and LibTorch are approximately the same across resolutions and batch sizes, while cuDNN and TorchScript are more memory efficient, especially below resolution 2^6 and batch size 2^7.

We use NVIDIA's Visual profiler[26] to investigate fixed `batch_size = 32` further. One critical way in which the PyTorch implementation differs from the others is in host-to-device per batch copy size: PyTorch copies 25.166MB while the other implementations copy 12.583MB. Consequently PyTorch requires ~15.17ms to copy the batch while all other implementations require ~7.55ms[27]. Another significant discrepancy is the choice in block size and grid size (regarding threaded distribution across GPU SMs). For the ReLU kernel, which is the second most often executed kernel (~12% of total execution time), the PyTorch implementation allocates a grid

[26] https://developer.nvidia.com/nvidia-visual-profiler
[27] In fact, pinning memory (copying from memory that is not paged) halves copy time again.

Table 1: Test platforms

CPU	AMD Ryzen 2970WX 24-Core @ 4.2 GHz
GPU	GeForce GTX 1080Ti
HD	Crucial MX500 2TB 3D NAND SATA
RAM	64GB
Software	PyTorch-1.7.0, CUDA-11.1, NVIDIA-455.23.05, g++10.1.0

Training platform

CPU	Intel Xeon Gold 6230 CPU @ 2.10GHz
GPU	Tesla V100-PCIE-32GB
HD	HPE 800GB SAS 12G Mixed Use SFF
RAM	384GB
Software	PyTorch-1.7.0, CUDA-11.1, NVIDIA-450.80.02

Resolutions platform

of size (1024, 2, 1) while the LibTorch and cuDNN implementations allocate grids of size (512, 2, 1). Consequently, on average, each PyTorch ReLU invocation consumes ~690.6 microseconds while each invocation consumes ~372.6 microseconds. it is unclear exactly why the larger grid leads to a slowdown but one hypothesis is that distributing work across more SMs leads to more stalls on cache misses[28].

5 DISCUSSION

Overall the central challenges for this work revolved around the cuDNN implementation. Chief among them involved familiarizing ourselves with the SIMT compute model and the CUDA/cuDNN APIs. it is important to contextualize these challenges appropriately: how much of the challenge is akin to the initial learning curve associated with any new software toolset (e.g. PyTorch) and how much is enduring. Certainly memory leaks, and debugging them, given that C++ is not a memory managed language, will persist, but the difficulties associated with the idiosyncrasies of the APIs most likely will not. Assuming that the majority of challenge decreases with time, are these lower levels of abstraction worth the investment of effort and time?

The lackluster accuracy results of cuDNN seemingly do not bode well for the hypothesis that one can, in a straightforward way, trade performance for implementation effort. The cuDNN performance indicates there are serious bugs with the implementation. Alternatively the accuracy results suggest that there are optimizations present in the PyTorch and LibTorch implementations that are obscured from the user (such as the Kaiming initialization mentioned in section (4)). The former case, when juxtaposed with the execution time and memory usage results, suggests that the cuDNN implementation could be as accurate the PyTorch and LibTorch implementations, with much lower execution time and memory usage (assuming the bugs can be rectified — a reasonable assumption). In the latter case, we face a sort of existential crisis: how many results in DL research, attributed to architectural innovations, in fact, hinge on the implementation details of the frameworks those architectures are themselves implemented against?

[28]SM level statistics are not presented in the NVIDIA profiler.

It bears repetition: even broadly useful heuristics are a cost of abstraction if they cannot be adjusted. Case in point, **Kaiming initialization is not always net positive with respect to performance**:

> Standard initialization methods (Glorot & Bengio, 2010; He et al., 2015; Xiao et al., 2018) attempt to set the initial parameters of the network such that the activations neither vanish nor explode. Unfortunately, it has been observed that without normalization techniques such as BatchNorm they do not account properly for the effect of residual connections and this causes exploding gradients. [33]

In addition batch normalization layers being initialized to ($\gamma = 1, \beta = 0$) also does not uniformly improve performance:

> For BN layers, the learnable scaling coefficient γ is initialized to be 1, *except for each residual block's last BN where γ is initialized to be 0*. Setting $\gamma = 0$ in the last BN of each residual block causes the forward/backward signal initially to propagate through the identity shortcut of ResNets, which we found to ease optimization at the start of training. [34]

In theory, a sufficiently flexible DL compiler could rescue us from the purgatory we find ourselves in; a sufficiently powerful compiler would implement the necessary DL abstractions in a robust way but also have enough flexibility to enable users to implement custom extensions of those abstractions. One promising project that has as its goal such a high-level compiler is the "Extensible Programming" [35] project. Besard et al. expose interfaces to alter the compilation process for Julia-lang[29]. The project instruments the Julia compiler itself and enables users to build high-level hardware abstractions in the source language itself[30]. They've had initial success writing high-level GPU code that performs comparably with CUDA C[31].

6 CONCLUSION AND FUTURE WORK

In this work we have implemented ResNet-50 in PyTorch, LibTorch, TorchScript, and cuDNN. We then trained[32] and evaluated each implementation on the MNIST, CIFAR10, STL10, and PASCAL VOC datasets. Despite difficulties with the cuDNN implementation, we show that PyTorch underperforms lower level abstractions along various batch sizes and resolutions (see figs. (8), (9), (10), (11), (12) and (13) in the appendix). The ultimate causes for these differences in performance are hypothesized to be the larger buffers and larger grid allocations used by PyTorch and consequently longer host-to-device copy times.

Future work will focus on further narrowing down the causes of the memory and sample time inefficiencies of PyTorch; in particular we hope to more closely investigate the execution paths of the PyTorch and LibTorch implementations in order to discover what additional heuristic choices are made (relative to the cuDNN

[29]https://julialang.org/

[30]This is possible because Julia is JITed using LLVM and is homo-iconic i.e. it supports a macro system that can manipulate the JITed LLVM bytecode.

[31]https://github.com/JuliaGPU/CUDAnative.jl

[32]With the exception of TorchScript since currently it cannot be trained.

implementation). A long term research goal is to design a DL framework that uses code generation to statically generate C++ code corresponding to neural network graphs. Such a framework would obviate the need for dynamic dispatch at the object level.

7 SPECULATION

We speculate about DL systems along three dimensions: hardware, software, and use cases/techniques. Firstly, we project that with the end of Dennard scaling [36] general purpose processors will give way to Application Specific Integrated Circuit (ASIC). Architectures like Cerebras' CS-1, SambdaNova's Cardinal SN10, Google's TPU, and even Apple's Neural Engine (packaged with their M1) demonstrate that chip designers recognize the need for ML/DL purpose built hardware. These purpose built chips are better suited for the concurrency and memory access patterns unique to ML/DL workloads. With the impending cambrian explosion of different architectures, there will be a great need for compilers that act as intermediaries between high-level neural network representations and their hardware implementations. Compiler infrastructures such as MLIR, PlaidML, and Intel's oneAPI [37] will become critical for software developers. As ML/DL ASICs become more ubiquitous and more performant, ML/DL powered software will become commensurately more ubiquitous; already mobile phones employ ML/DL for text autocorrection [38], image compression/super-resolution [39], and facial fingerprinting [20]. We project that most user-interaction driven tasks (e.g. setting alarms, coordinating appointments/meetings, planning daily routines) will have ML/DL solutions in the coming years. Further, more platforms that have been up until today analog or simple computers (e.g. kitchen appliances, light power tools) will become "smart". This distribution of lower power edge devices will necessitate more effective (and private) federated learning methods [40].

8 ACKNOWLEDGEMENTS

We would like to thank Rick Stevens and Ian Foster for their constructive criticism and feedback on the project and paper itself.

REFERENCES

[1] Paszke, A. et al, Pytorch: An imperative style, high-performance deep learning library (2019).
[2] Abadi, M. et al, Tensorflow: Large-scale machine learning on heterogeneous distributed systems (2016).
[3] Chen, T. et al, Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems (2015).
[4] Seide, F. and Agarwal, A., *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16 (Association for Computing Machinery, New York, NY, USA, 2016), p. 2135.
[5] Knuth, D.E., *Commun. ACM* **17**, 667–673 (1974).
[6] Abdelhamed, A. et al, Ntire 2020 challenge on real image denoising: Dataset, methods and results (2020).
[7] Hall, D. et al, *IEEE Winter Conference on Applications of Computer Vision (WACV)* (2020).
[8] Russakovsky, O. et al, *International Journal of Computer Vision (IJCV)* **115**, 211 (2015).
[9] Shi, W. et al, *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016), pp. 1874–1883.
[10] Brown, T.B. et al, Language models are few-shot learners (2020).
[11] Locke, J., *An Essay Concerning Human Understanding* (Oxford University Press, 1689).
[12] Russell, B., *Principles of Mathematics* (Routledge, 1937).
[13] Colburn, T. and Shute, G., *Minds and Machines* **17**, 169 (2007).
[14] Abelson, H., Sussman, G. and Sussman, J., *Structure and Interpretation of Computer Programs*, Electrical engineering and computer science series (MIT Press, 1996).
[15] Pippenger, N., *ACM Trans. Program. Lang. Syst.* **19**, 223–238 (1997).
[16] Skiena, S.S., *The Algorithm Design Manual* (Springer Publishing Company, Incorporated, 2008), second edn.
[17] Stroustrup, B., *Proceedings of the 21st European Conference on Programming Languages and Systems*, ESOP'12 (Springer-Verlag, Berlin, Heidelberg, 2012), p. 1–25.
[18] Sanders, J. and Kandrot, E., *CUDA by Example: An Introduction to General-Purpose GPU Programming* (Addison-Wesley Professional, 2010), first edn.
[19] Flynn, M.J., *IEEE Transactions on Computers* **C-21**, 948 (1972).
[20] NVIDIA, Cuda toolkit documentation, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#control-flow-instructions (2020). [Online; accessed 3-December-2020].
[21] Glaskowsky, P. (2009).
[22] Wittenbrink, C.M., Kilgariff, E. and Prabhu, A., *IEEE Micro* **31**, 50 (2011).
[23] Panchenko, M. et al, *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019 (IEEE Press, 2019), p. 2–14.
[24] Le, T.D. et al, Tflms: Large model support in tensorflow by graph rewriting (2019).
[25] Pradelle, B. et al, *ESPT/VPA@SC* (2017).
[26] Lattner, C. et al, Mlir: A compiler infrastructure for the end of moore's law (2020).
[27] Zerrell, T. and Bruestle, J., Stripe: Tensor compilation via the nested polyhedral model (2019).
[28] Griebl, M., Lengauer, C. and Wetzel, S., *In IEEE PACT* (IEEE Computer Society Press, 1998), pp. 106–111.
[29] Vasilache, N. et al, Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions (2018).
[30] Chen, T. et al, *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18 (USENIX Association, USA, 2018), p. 579–594.
[31] He, K. et al, Deep residual learning for image recognition (2015).
[32] He, K. et al, Delving deep into rectifiers: Surpassing human-level performance on imagenet classification (2015).
[33] Zhang, H., Dauphin, Y.N. and Ma, T., Fixup initialization: Residual learning without normalization (2019).
[34] Goyal, P. et al, Accurate, large minibatch sgd: Training imagenet in 1 hour (2018).
[35] Besard, T., Foket, C. and De Sutter, B., *IEEE Transactions on Parallel and Distributed Systems* **30**, 827–841 (2019).
[36] ã o M.P. Cardoso, J., é Gabriel F. Coutinho, J. and Diniz, P.C., *Embedded Computing for High Performance*, ã o M.P. Cardoso, J., é Gabriel F. Coutinho, J. and Diniz, P.C., eds. (Morgan Kaufmann, Boston, 2017), pp. 17 – 56.
[37] Intel, Driving a new era of accelerated computing, https://software.intel.com/content/www/us/en/develop/tools/oneapi.html (2020). [Online; accessed 3-December-2020].
[38] Ghosh, S. and Kristensson, P.O., *CoRR* **abs/1709.06429** (2017).
[39] Romano, Y., Isidoro, J. and Milanfar, P., *CoRR* **abs/1606.01299** (2016).
[40] Augenstein, S. et al (2019).

Appendices

A EXTRA PLOTS

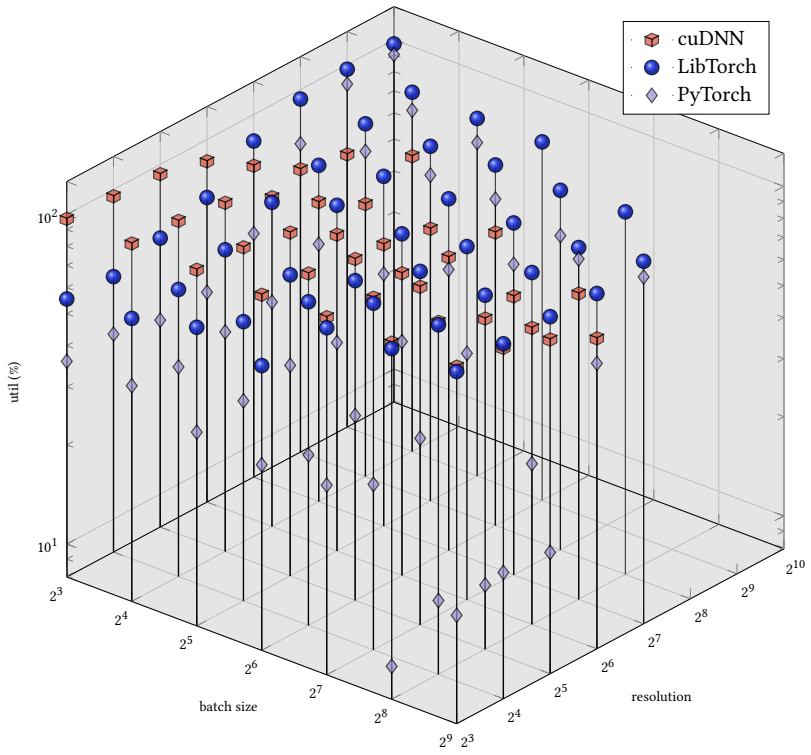


Figure 8: Average GPU utilization during in training

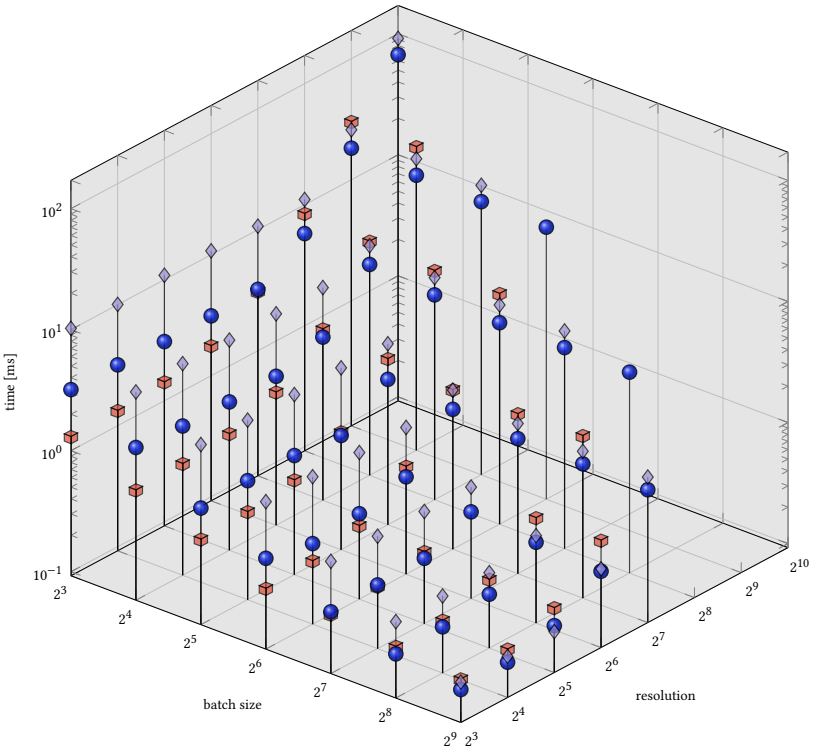


Figure 9: Average sample time in training on PASCAL

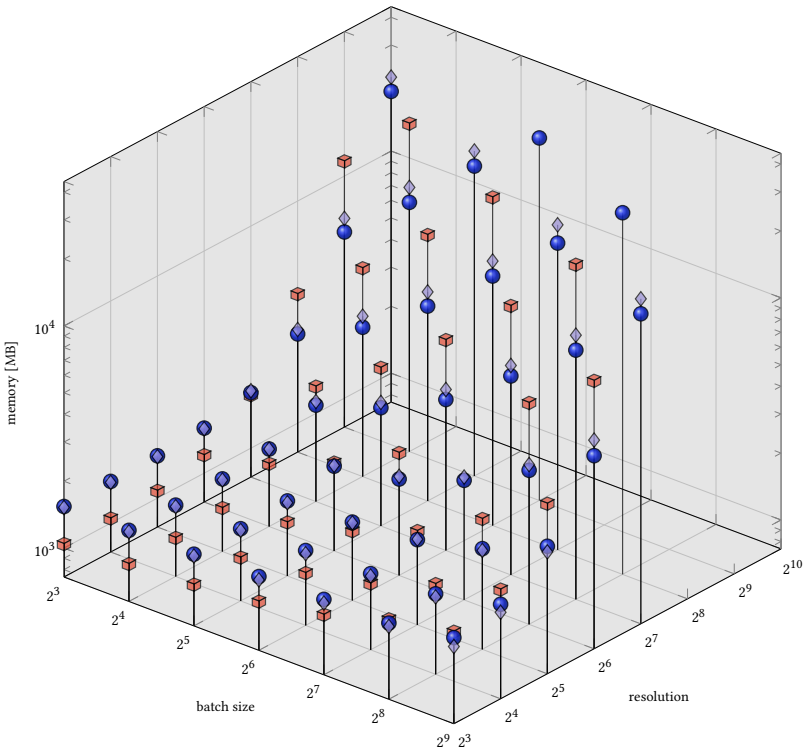


Figure 10: Average memory used in training on PASCAL

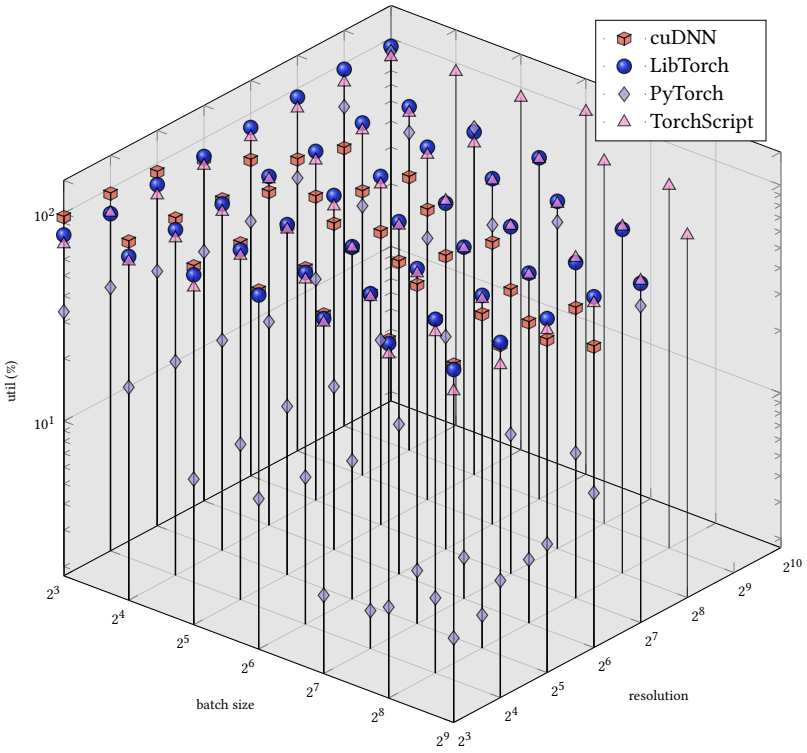


Figure 11: Average GPU utilization during in evaluation on PASCAL

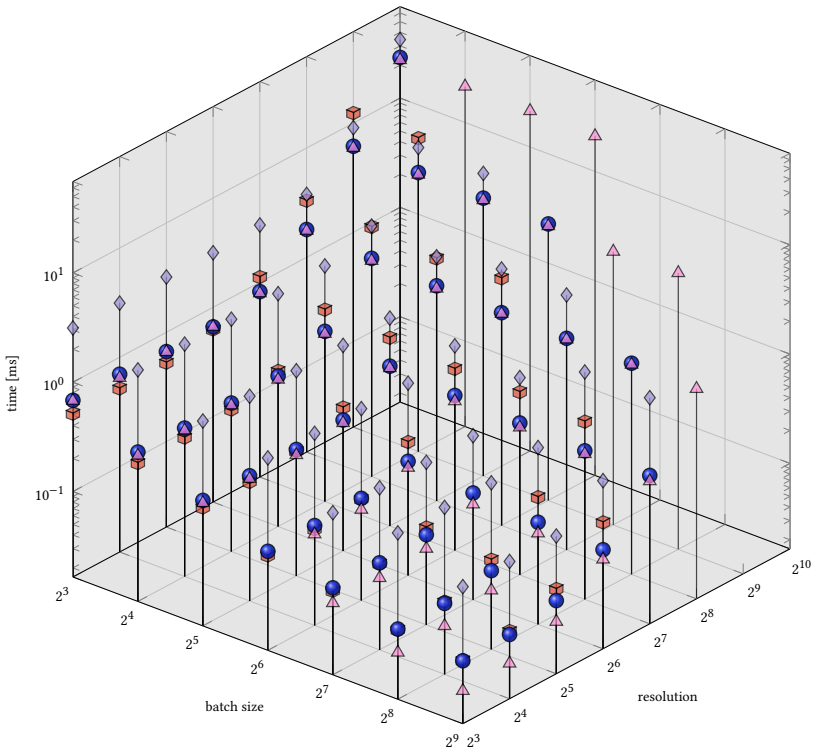


Figure 12: Average sample time in evaluation on PASCAL

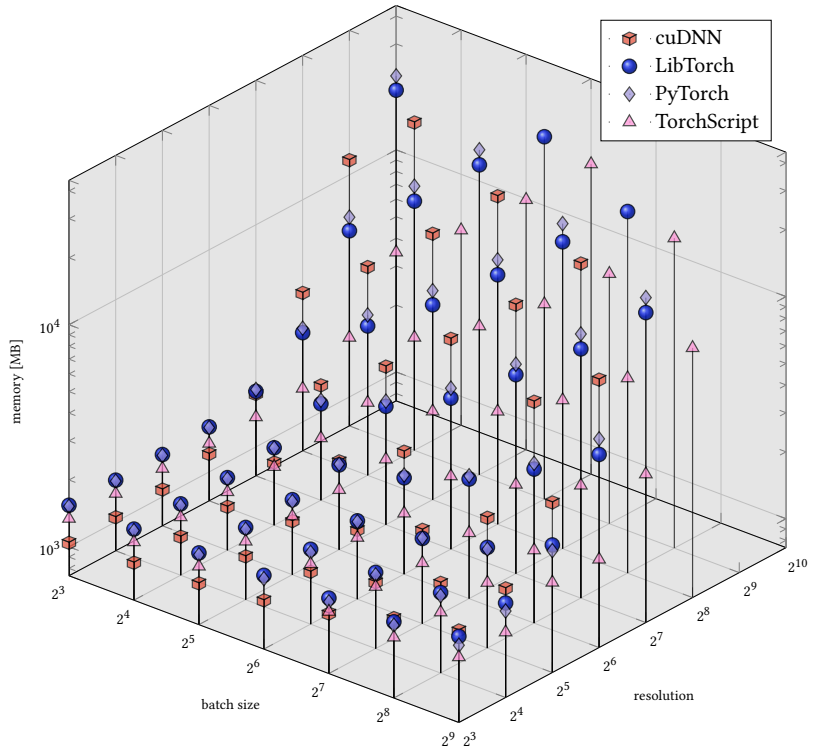


Figure 13: Average memory used in evaluation on PASCAL

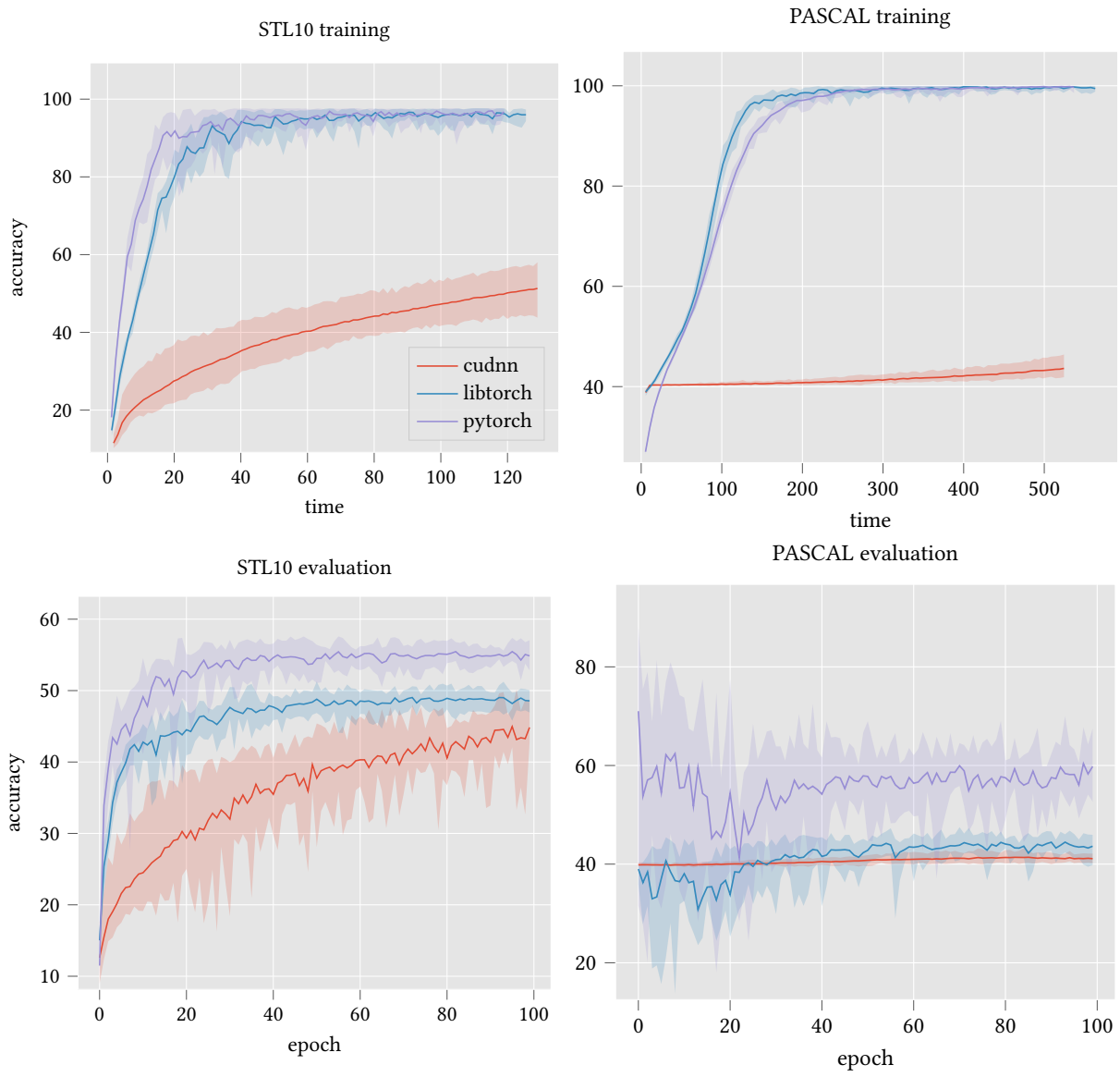


Figure 14: Comparison of accuracy for PyTorch, LibTorch, and cuDNN implementations during training and evaluation. Solid line corresponds to mean while shaded regions correspond to min and max. Time is measured in units of *epoch* × *average epoch time*.