When they go high we go low

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 enormous translation and indirection overheads. We study at which points exactly in the engineering life-cycle of a DL model are the highest costs 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.

ACM Reference Format:
Maksim Levental and Elena Orlova. 2020. When they go high we go low. 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 [2, 5, 10, 12] 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 onto 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 though, if one is in industry and taking for granted a research direction bears fruit, 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 [3, 7, 11]. For example, in the area of super-resolution a deliberate goal is to be able to "super-resolve" in real-time [13]. Similarly, in natural language processing, where enormous language models are becoming the norm [4], 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 aim here is to investigate the costs of some of the abstractions employed by framework developers. In particular we focus on the PyTorch framework and ecosystem (chosen for its popularity amongst academic researchers) on the software side and Graphics Processing Units (GPUs) on the hardware side. 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 that in the forthcoming that these four implementations do in fact span considerable breadth in the abstraction spectrum. Furthermore we train, test, 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, 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) proposes future work, and section (7) speculates wildly about the future of DL systems more generally.

2 BACKGROUND

2.1 GPUs

We briefly review NVIDIA GPUs[4] in order that the performance criteria we measure in section (3) are intelligible.

A GPU consists of many simple processors, called streaming multiprocessors (SMs), colloquially referred to as cores, that run at

*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." [9]
[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 [1].

relatively low clock speeds[5] and execute one floating-point 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[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) this 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 shared 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 comprised by many *threads* that execute the same instruction(s) in parallel; NVIDIA describes this addition to Flynn's taxonomy [6] as Single Instruction Multiple Thread (SIMT). 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 fig. (1b)) and starting with the Fermi architecture two warps can be executed concurrently on the same SM in order to increase utilization.

2.2 Graph compilers

DL frameworks function as graph compilers. What are the design choices made by DL framework architects and what are their costs? That is to say, what are the costs of the abstractions

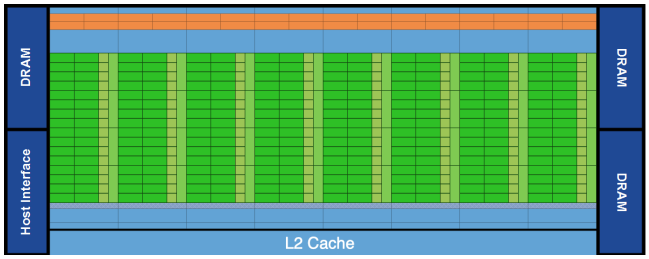
These frameworks encapsulate and "abstract away" many of the implementation details of DL, such as

- building the dataflow graph between units/layers (and the corresponding gradient-flow graph)
- tensor manipulation and memory layout
- hardware specific optimizations

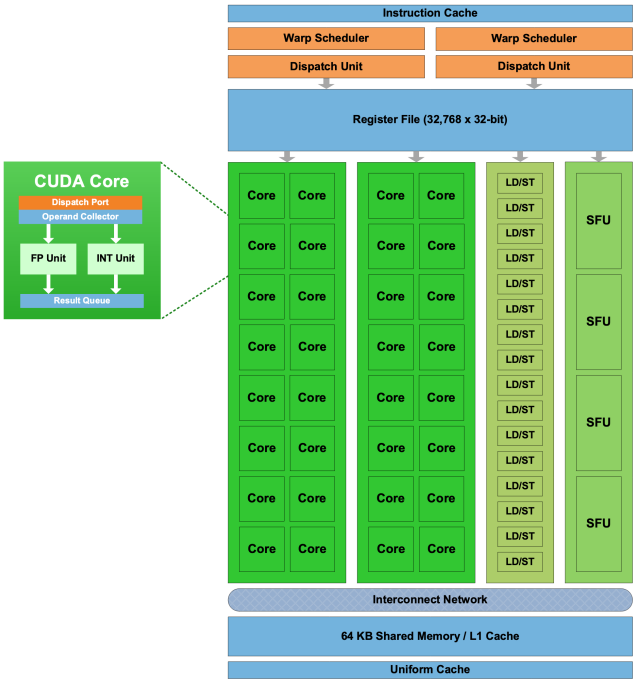
2.2.1 *Static.*

2.2.2 *Dynamic.*

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



(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 [14]

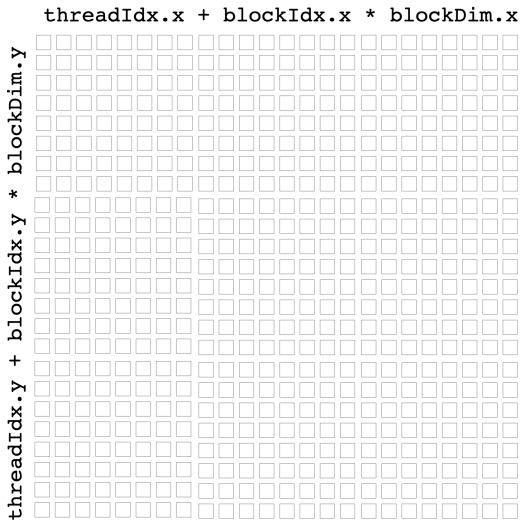
3 METHODOLOGY

We implement ResNet-50 at four levels of abstraction: PyTorch, TorchScript, LibTorch, and cuDNN. The reason for staying within the same ecosystem (PyTorch)is, in theory, we keep as many of the pieces of functionality orthogonal to our concerns as possible. We'll see that that reasoning doesn't quite bear out (see5).

```
__global__ void matrix_sum(
    float *A,
    float *B,
    float *C,
    int m,
    int n
) {
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    if (x < m && y < n) {
        int ij = x + y*m; // column-major order
        C[ij] = A[ij] + B[ij];
    }
}

int main() {
    int no_of_blocks = 10, threads_per_block = 16;
    int m = 100, n = 50;
    float A[m][n], B[m][n], C[m][n];
    matrix_sum<<m,n>>(A, B, C, m, n);
}
```

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



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

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

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 3: ResNet-50 network architecture[8].

3.1 Implementations

3.2 Profiling

4 RESULTS

4.1 Training and evaluation

4.2 Memory and utilization

5 DISCUSSION

6 FUTURE WORK

7 SPECULATION

REFERENCES

[1] 2014. *Professional CUDA C Programming* (1st ed.). Wrox Press Ltd., GBR.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. arXiv:cs.DC/1603.04467

[3] Abdelrahman Abdelhamed, Mahmoud Afifi, Radu Timofte, Michael S. Brown, Yue Cao, Zhilu Zhang, Wangmeng Zuo, Xiaoling Zhang, Jiye Liu, Wendong Chen, Changyuan Wen, Meng Liu, Shuailin Lv, Yunchao Zhang, Zhihong Pan, Baopu Li, Teng Xi, Yanwen Fan, Xiyu Yu, Gang Zhang, Jingtuo Liu, Junyu Han, Errui Ding, Songhyun Yu, Bumjun Park, Jechang Jeong, Shuai Liu, Ziyao Zong, Nan Nan, Chenghua Li, Zengli Yang, Long Bao, Shuangquan Wang, Dongwoon Bai, Jungwon Lee, Youngjung Kim, Kyeongha Rho, Changyeop Shin, Sungho Kim, Pengliang Tang, Yiyun Zhao, Yuqian Zhou, Yuchen Fan, Thomas Huang, Zhihao Li, Nisarg A. Shah, Wei Liu, Qiong Yan, Yuzhi Zhao, Marcin Możejko, Tomasz

Latkowski, Lukasz Treszczotko, Michał Szafraniuk, Krzysztof Trojanowski, Yanhong Wu, Pablo Navarrete Michelini, Fengshuo Hu, Yunhua Lu, Sujin Kim, Wonjin Kim, Jaayeon Lee, Jang-Hwan Choi, Magauiya Zhussip, Azamat Khassenov, Jong Hyun Kim, Hwechul Cho, Priya Kansal, Sabari Nathan, Zhangyu Ye, Xiwen Lu, Yaqi Wu, Jiangxin Yang, Yanlong Cao, Siliang Tang, Yanpeng Cao, Matteo Maggioni, Ioannis Marras, Thomas Tanay, Gregory Slabaugh, Youliang Yan, Myungjoo Kang, Han-Soo Choi, Kyungmin Song, Shusong Xu, Xiaomu Lu, Tingniao Wang, Chunxia Lei, Bin Liu, Rajat Gupta, and Vineet Kumar. 2020. NTIRE 2020 Challenge on Real Image Denoising: Dataset, Methods and Results. arXiv:cs.CV/2005.04117

[4] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:cs.CL/2005.14165

[5] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. arXiv:cs.DC/1512.01274

[6] M. J. Flynn. 1972. Some Computer Organizations and Their Effectiveness. *IEEE Trans. Comput.* C-21, 9 (1972), 948–960. https://doi.org/10.1109/TC.1972.5009071

[7] David Hall, Feras Dayoub, John Skinner, Haoyang Zhang, Dimity Miller, Peter Corke, Gustavo Carneiro, Anelia Angelova, and Niko Sünderhauf. 2020. Probabilistic Object Detection: Definition and Evaluation. In *IEEE Winter Conference on Applications of Computer Vision (WACV)*.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. arXiv:cs.CV/1512.03385

[9] Donald E. Knuth. 1974. Computer Programming as an Art. *Commun. ACM* 17, 12 (Dec. 1974), 667–673. https://doi.org/10.1145/361604.361612

[10] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. arXiv:cs.LG/1912.01703

[11] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)* 115, 3 (2015), 211–252. https://doi.org/10.1007/s11263-015-0816-y

[12] Frank Seide and Amit Agarwal. 2016. CNTK: Microsoft's Open-Source Deep-Learning Toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. Association for Computing Machinery, New York, NY, USA, 2135. https://doi.org/10.1145/2939672.2945397

[13] W. Shi, J. Caballero, F. Huszár, J. Totz, A. P. Aitken, R. Bishop, D. Rueckert, and Z. Wang. 2016. Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 1874–1883. https://doi.org/10.1109/CVPR.2016.207

[14] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu. 2011. Fermi GF100 GPU Architecture. *IEEE Micro* 31, 2 (2011), 50–59. https://doi.org/10.1109/MM.2011.24

Appendices

A APPENDIX