Exploring Hardware Parallelism's Influence on Programming Through Convolution in CUDA and PyTorch

A CS242 Project Report

Sam Redmond
Stanford School of Engineering
Stanford, California
sredmond@stanford.edu

Christopher Sauer
Stanford School of Engineering
Stanford, California
cpsauer@stanford.edu

ACM Reference Format:
Sam Redmond and Christopher Sauer. 2017. Exploring Hardware Parallelism's Influence on Programming Through Convolution in CUDA and PyTorch: A CS242 Project Report. In *Proceedings of CS242 Autumn 2017*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.100.10

1 SUMMARY

In our final project, we explored programming paradigms for general parallel computing on GPUs. In a world in which performance gains increasingly come from parallel hardware, our goals were (1) to understand how sequential programming paradigms compare to those used on the massively parallel hardware of a GPU and (2) to understand the benefits and drawbacks of the parallel paradigms that seek to replace sequential ones. We began by exploring the historical context of CUDA and GPGPU and by getting an understanding of how parallel hardware architectures influence which programming constructs can be executed efficiently. We then implemented several solutions to a classic parallel problem, image convolution, directly in CUDA and benchmarked against implementations we wrote using PyTorch, cuDNN, and sequential and parallel C++. We found CUDA to be surprisingly natural to use for sequential programmers, even compared to the wrappers that intend to abstract it away. We also found that it offered significantly more freedom in programming patterns by allowing intra-kernel compositionality. We concluded our project by exploring the benefits of that compositionality on programmability and performance by generating the following edge saturation effect.

CS242 Autumn 2017, December 2017, Palo Alto, California USA

ACM ISBN 123-4567-89…$1,000,000.00
https://doi.org/10.100.10

2 BACKGROUND

2.1 Project Context

Both of us have relied upon GPUs as co-processors to accelerate deep learning workloads; however we have always been abstracted far away from the hardware. With this project, we aimed to understand how languages and programming paradigms change beneath that abstraction boundary to adapt to the highly parallel hardware of a GPU.

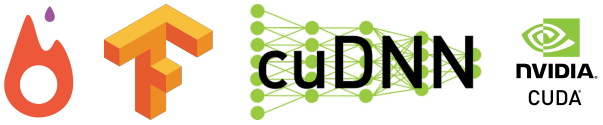


Figure 1: PyTorch, TensorFlow, cuDNN and CUDA.

In particular, libraries like PyTorch and TensorFlow wrap highly optimized, parallel operations on GPUs into matrix operations and auto-differentiation. As long as the operations you wish to execute can be easily expressed in terms of their library primitives, these libraries allow the parallel operations to fit naturally into easy-to-write, single-threaded, imperative Python programs, usually without requiring knowledge of how the underlying hardware works. The fact that TensorFlow and PyTorch can impose such a clean abstraction boundary with so few leaks is certainly worthy of study. However, in this project we were interested in exploring beneath the abstractions provided by those libraries. In particular, we wanted to better understand the following:

(1) How do programming languages change beneath the boundary to accommodate far greater hardware parallelism than on a CPU?
(2) What performance penalties do the abstractions provided by those libraries incur?
(3) What additional flexibility does programming in CUDA provide?

We were not disappointed by CUDA's graceful adaptation of sequential constructs over to massively parallel ones, by the performance gains from directly writing CUDA code, or by the additional freedom of composability offered by CUDA.

2.2 Why are GPUs so Important?: Moore's Law and the Rise of Parallelism

We are interested in understanding how languages change to better support GPU parallelism not only because we are reliant upon

their output but also because we are moving toward a world with greater and greater hardware parallelism. As improvements in sequential execution plateau, future gains must come from increased parallelism.

Consider the following chart, shown in class:

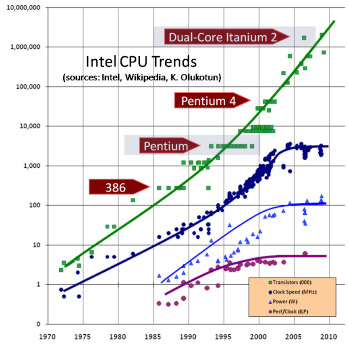


Figure 2: Dramatic plateau in single core speeds in 2004. Graph shown in class on Nov. 8, 2017 [9].

Figure 2 should cause panic. As computer scientists, we have relied on the exponential march of improved processor performance to continually unlock new opportunities without having to considerably complicate our programs.

Programming language development is certainly no exception. One example is our reliance on continued relaxation of performance constraints to allow for better and better abstractions in order to increase programmer productivity. Since with current paradigms, even moderate parallelism tends to introduce significant complexity for the programmer, which is why most programs are still based around a single thread, a plateau in single-core speed would be disastrous.

Thankfully, this trend did not continue quite as badly as it seemed in the mid 2000s.

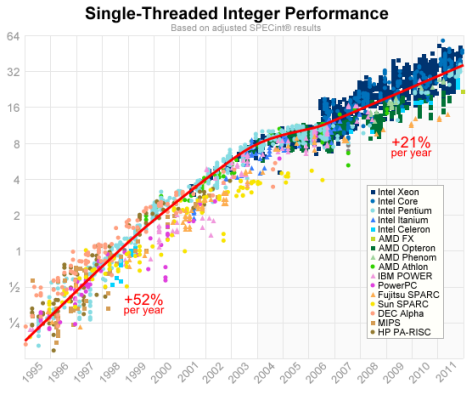


Figure 3: SPEC benchmarks for most processors released between 1995 and 2012. Note the plateau around the single-core crisis in 2004, and the subsequent return to increases at a lower rate thereafter. Graph from [1].

Figure 3 presents perhaps a fairer view of how single core performance has evolved. After the brief plateau from 2004-05, performance continued to increase at 21%.

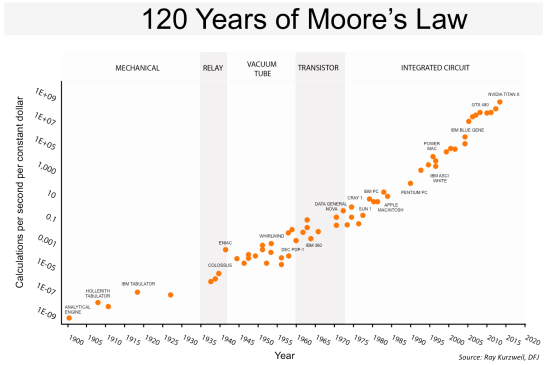


Figure 4: Moore's law continues apace in terms of compute per dollar, with its slope accelerating [17]. Note that all devices in the upper right are GPUs.

Still, given the log scale, the difference is quite significant—52% growth would quickly compound 21% growth into irrelevance. And that is exactly what is happening. Moore's law is happily continuing apace—just look at Figure 4—except most of the gains are going into massively parallel processors, like GPUs.

We are living in a world where it appears gains in massively parallel processors will quickly compound single or few-cored processors into computational irrelevance. This makes studying languages that have succeeded in easing parallelism of the utmost importance. This is especially true given that our traditional imperative and object-oriented programming paradigms largely fail us in parallel environments by being built around a linear thread of execution. Again, recall that most programs use only one or at most a few threads.

Since CUDA's simplification and structuring of highly parallel programming kicked off a revolution in using GPUs for accelerating massively parallel workloads, it seems reasonable to learn from CUDA how more commonly used languages might adapt to parallelism in the future.

2.3 Project Phase 1: History of GPUs, the Evolution of GPU Programming, and the Rise of General Purpose Computation on GPUs

We are young enough to not remember the historical context in which general purpose computing on GPU evolved. To understand the space of possible languages, it is important to learn the history of programming languages for these devices. Therefore, the first undertaking for our project was to compile such a history.

Since it has all happened so recently and is changing so fast, there is a distinct lack of comprehensive accounts available online. Here is what we compiled, first as a history and second as a timeline.

Modern graphics processing units (GPUs) were originally created in the 1990s to accelerate realtime, scanline graphics, which is an inherently parallel operation on triangles.

Customizations to the image were provided via shaders, written in assembly languages like ARB or the DirectX Shader Assembly Language (see Figure 5). Each shader was run in independently in parallel, over pixels, vertices, etc. These languages were more hardware configuration languages than general purpose computing languages: They had fixed total program length and a very limited instruction set.

```
!!ARBvp1.0
TEMP vertexClip;
DP4 vertexClip.x, state.matrix.mvp.row[0], vertex.position;
DP4 vertexClip.y, state.matrix.mvp.row[1], vertex.position;
DP4 vertexClip.z, state.matrix.mvp.row[2], vertex.position;
DP4 vertexClip.w, state.matrix.mvp.row[3], vertex.position;
MOV result.position, vertexClip;
MOV result.color, vertex.color;
MOV result.texcoord[0], vertex.texcoord;
END
```

Figure 5: Example of ARB Shader Assembly Language from [5]. Total execution length and the instruction set were quite limited, because the code was configuring a step in a hardware rendering pipeline.

By 2002, pressures for more shader flexibility had driven graphics hardware to become sufficiently general to support higher level shading languages like Direct3D's HLSL and OpenGL's GLSL. In subsequent versions, fixed-size hardware limitations disappeared to allow these shaders to become become more like software than configured hardware (see Figure 6). As an example, the number of instructions quickly scaled from 12 to unlimited, as we would expect for a software program.

```
layout (std140) uniform Matrices {
    mat4 projModelViewMatrix;
    mat3 normalMatrix;
};

in vec3 position;
in vec3 normal;
in vec2 texCoord;

out VertexData {
    vec2 texCoord;
    vec3 normal;
} VertexOut;

void main()
{
    VertexOut.texCoord = texCoord;
    VertexOut.normal = normalize(normalMatrix * normal);
    gl_Position = projModelViewMatrix * vec4(position, 1.0);
}
```

Figure 6: Example of GLSL from [14]. Note the resemblance to C to help convey the new flexibility. Input variables sent to all instances running in parallel are termed "uniforms." Per-thread inputs and outputs are labeled with the keywords "in" and "out." Each shader is typically in its own file, not composed.

With single core speeds beginning to tap out in 2004, Intel and AMD scrambled to release multicore chips in 2005, bringing hardware parallelism into the mainstream. NVIDIA, noticing that GPU hardware was now sufficiently general for non-graphics tasks, released CUDA in 2007, which allowed direct programming of non-graphics workloads on GPUs. The language has easy interoperability with C++, prioritizing ease of converting and sharing existing CPU code. See analysis in Figure 7.

```
__global__
void add(int n, float *x, float *y)
{
  int index = threadIdx.x;
  int stride = blockDim.x;
  for (int i = index; i < n; i += stride)
      y[i] = x[i] + y[i];
}
```

Figure 7: Example of CUDA from [13]. Input variables sent to all instances running are now expressed as the input variables to the kernel, while values sent into each thread are sent in as the threadIdx and blockDim "global" variables. The kernel would run as a valid C++ function if the CUDA references were stripped and can be intermixed with C++ code in the same file. Different kernels can easily draw on the same helper functions, declared with "__device__." Code that runs on both GPU and CPU can be declared with "__host__ __device__".

Since then, there has been significant wrapping of those APIs, especially for matrix operations and machine learning workloads. Pytorch, Tensorflow, and others are all dependent on NVIDIA's cuBLAS and cuDNN implementations, while OpenCL support remains sparse. We will describe the tradeoffs between these interfaces more in later sections.

Separately, OpenACC was released for easily parallelizing existing programs explicitly based upon OpenMP, which we touched on in class. Like OpenMP, OpenACC is based on compiler directives (see Figure 8), but the fine-grained control around the different task and memory models is somewhat problematic, and there is no working implementation for most platforms.

```
// OpenMP
#pragma omp parallel for collapse(2) schedule(guided)
```

```
// OpenACC
#pragma acc kernels {
#pragma acc loop independent collapse(2)
```

Figure 8: OpenMP code for guided parallelization of a double for loop and the equivalent OpenAcc code. OpenMP code is from one of Christopher's programs. OpenAcc is from [19].

(Sources [6, 7, 10, 12, 14, 15, 19–23, 26])

2.4 GPU Language Evolution Timeline

- **1979** — Basic Linear Algebra Subprograms (BLAS) Released for Fortran
- **1992** — OpenGL released by Silicon Graphics

- **1996** — Direct3D released by Microsoft
- **1997** — OpenMP released
- **2002** — HLSL released by Microsoft to Replace DirectX Shader Assembly Language (!), enabled by increased flexible GPU rendering pipelines.
- **2004** — GLSL released by the OpenGL Architecture Review Board (ARB), replacing the ARB assembly language.
- **2004** — Single-core speeds begin to tap out
- **2005** — Intel releases first dual-core processor intended for normal use, beating AMD's long-announced processor announcement by a few weeks. See [24].
- **2007** — CUDA released by NVIDIA, giving direct access to GPU instructions rather than as a shader. Enables GPUs as general-purpose parallel accelerators. Originally stands for Compute Unified Device Architecture
- **2009** — OpenCL follows CUDA as non-proprietary variant.
- **2013** — OpenACC Official Release
- **2014** — NVIDIA Releases cuDNN for deep learning. (Date from [11].)
- **2015** — TensorFlow Public Release
- **2016** — PyTorch Initial Release
- **Currently** — CUDA is dominant over OpenCL. NVIDIA has 80% market share [3] because AMD cards are behind the last generation NVIDIA cards and have higher power consumption [4]. OpenCL performance is still famously bad on NVIDIA cards [25], and most users are heavily reliant on NVIDIA libraries like cuDNN. See practically no OpenCL support for deep learning software in reference [8].

(Note, many dates are not easily found on the web because people are mostly too busy releasing software to chronical its history. Those dates were found from press releases.)

2.5 Key Architectural Differences on GPU

Despite CUDA's similarity to C++ and the ability of modern GPUs to execute general purpose routines, GPUs have several architectural differences that are essential to keep in mind while programming. GPU languages need to expose these constructs to allow the user to write efficient code. There is a tension here in programming language design: Making GPU programming as similar as possible to CPU programming makes it easer for sequential programmers to onboard, but you need to expose new constructs in order to make those programs fast. And performance is, after all, the whole point of using a GPU.

Here are those key architectural differences:

GPUs have hierarchically structured parallelism. Threads are arranged into cohorts or warps, which execute together in lockstep on a single streaming multiprocessor (SM). Workgroups for a specific task are together called a kernel. Individual GPU threads execute quite slowly; their speed comes from massive parallelism.

This mode of parallelism is termed SIMT (Single Instruction, Multiple Thread) and is different and more independently parallel than SIMD (Single Instruction Multiple Data) vector instructions on CPU. However, it is disguising as "threads," operations on a streaming multiprocessor processor that are not fully independent.

Unlike on CPU, switching threads is very low cost. Execution state for multiple cohorts is held at the same time per core, rather than being swapped into a thread control block like on CPU. Cohorts swap out while waiting on data.

For discrete GPUs that communicate with the CPU over PCIe, memory bandwidth is a major constraint. GPUs do not share memory with the CPU and passing data across the PCIe bus is a significant bottleneck. Note that this is not true in a SOC where the CPU and GPU share the same die. As we'll see in the results section, even within the GPU, memory bandwidth often becomes a bottleneck because the GPU has so much processing power.

Branching kills performance. For SIMT processors, if one thread takes a branch, all threads must execute those instructions and throw away the results.

Modern GPUs have caches but they're typically only a few bytes per thread. The caches are mostly to share data between threads in a warp, not for longer term temporal locality like on a CPU. Coalescing addresses from a single streaming multiprocessor also helps reduce bandwidth usage to GPU memory. Before these caches, there was more reliance on manual memory management, wherein one explicitly pinned to shared memory for temporal locality.

(Sources [2, 16, 18])

3 APPROACH

All the learning in the previous sections set us up to understand the space of programming languages for GPUs and the underlying architecture principles that drove them. That knowledge was good in theory—it was time to put it into practice.

To best learn how CUDA's parallelism models can support the abstractions we have relied upon, we started by implementing several versions of image convolution. Image convolution is useful for convolutional neural networks, of course, but also for a wide variety of other tasks, like blurring, anti-aliasing, and edge detection.

3.1 Visual Explanation of Approach

From an implementation perspective, we successfully implemented image convolutions using four different approaches:

(1) C++ code, running on CPU, at different optimization levels and with or without multithreading.
(2) Custom CUDA code, running on GPU.
(3) CUDA code that calls into cuDNN (NVIDIA's Deep Neural Network library), running on GPU.
(4) PyTorch code, running on GPU. We also explored the slowdown from running PyTorch on CPU.

Inherently visual operations are much better explained with pictures, so here goes:

Throughout this project, both in exploration and benchmarking, we used the low-poly render of a deer (shown in Figure 9) as our reference image, either in color or in grayscale.

We chose this image because it has visually interesting edges to detect and amplify. It turned out that it also has what look like interesting JPEG artifacts in the background, which our algorithms find and visualize. The image is high resolution enough to comfortably saturate our GPU's streaming multiprocessors ($1918x1078 = 2,067,604$ kernel applications across 22 streaming multiprocessors

or 2816 CUDA cores). Using the profiler, we can see that the GPU does indeed achieve full processor load while working on the image.

Figure 9: Our reference 1920x1080 image of a low-polygon deer.

Again, many image effects can be modeled as the application of a convolution kernel. By applying a specific 3x3 convolution kernel over an image, the resulting matrix (when viewed as an image) computes the edges of the original image, as shown in Figure 10. We simulate this common realtime image processing or computer vision task. Each of our four implementation strategies can successfully compute this image convolution and dump the resulting data. Our post-processing Python script grabs this data and renders it as an image in the case of C++, CUDA, and cuDNN (all of which operate on matrices, not images).

Figure 10: Edges in the original image, converted to grayscale.

Lastly, we wanted to build a more interesting visualization to strain the generality of each of our implementations. The goal was to have a task that is not a natural matrix operation in order to to understand the benefits of using CUDA's additional flexibility over libraries like PyTorch. We construct a composite image consisting of the original image, where each pixel's brightness is interpolated between its original value and a dimmed version, weighted by the aggregate pixel distance between itself and its neighbors, measured according to Euclidean distance in $\mathbb{R}^3$ on the RGB colorspace. Essentially, if a pixel is in an edge, its brightness is maintained, while non-edge pixels are dimmed. We built both PyTorch and CUDA implementations to handle this more complex workload, resulting in the pretty Figure 11.

Figure 11: A complicated per-pixel function results in this edge-saturated deer, which we think looks rather neat. The odd apparent block pattern in the background is the result of JPEG artifacts in the original image.

3.2 Implementation as Qualitative Results

Since our subjective experience writing these implementations is part of the results of our experiment, we have included the details and decisions we made in our implementation experience as the first part of the results section. For ease of reading, the description of our benchmarking is included with the results themselves.

4 RESULTS

Overall, we were extremely successful in achieving our goals. We originally proposed two criteria for evaluating this project's technical success: (1) analyzing the subjective experience of implementing convolutions in four different frameworks and (2) benchmarking the performance of these implementations on CPU and on GPU.

4.1 Qualitative

Subjectively, we noticed several differences in the experience of writing normal C++ code to be run on the CPU, writing custom CUDA coda for the GPU, calling into cuDNN, and writing a PyTorch script.

Both of us have prior experience with PyTorch, so we began our implementation journey there, expecting that the high-level abstractions of the library would make writing a simple convolution kernel easy and painless. PyTorch, in contrast to the low-level operations afforded (and required) by C++, CUDA, and cuDNN, provides three abstraction layers for the programmer: matrix operations, variables and auto-differentiation, and neural networks. Indeed, it explicitly bills itself as both a GPU replacement for numpy and a competitor of TensorFlow. It includes some libraries on the side for common tasks, like multi-threaded image loading, but the primary abstraction model is that from matrices to neural networks. As a result, we found that the built-in convolution operation (`torch.nn.functional.conv2d`) did not accept plain torch Tensors and required us to wrap the image data in a batch of size 1 and convert to a Variable, only to unwrap the batch and the Variable immediately afterwards. We noticed that we were immediately paying for the abstractions provided by PyTorch. However, one of these abstractions was incredibly useful. In PyTorch, moving objects to GPU in order to switch execution to use CUDA behind

the scenes contains just a call to `.cuda()` on the underlying Tensor or Variable. As a result, PyTorch would allow a developer to easily switch between CPU and GPU computations when moving between development environments and hardware.

Writing single-threaded C++ for naive image convolution, on the other hand, felt entirely familiar and straightforward. Although C++ forces the programmer to make decisions about stack-allocated vs. heap-allocated arrays, manage memory, and explicitly keep track of array bounds, with C++ we felt as though nothing mysterious was happening. It was rather frustrating that C++ does not allow heap-allocation of dynamically-sized 2D arrays, so we built a workaround using multidimensional indexing into a 1D array. One of the primary tradeoffs of C++ is that it gives you more control and but requires more responsibility than higher-level languages like Python.

Translating our existing C++ to a form usable by CUDA was perhaps the most natural portion of this project. In CUDA, C++ functions are annotated with `__host__`, `__global__`, or `__device__`, depending on which of the CPU and GPU can call and execute the function. GPU functions have the strange mechanic in which the parameters represent "global" data that all threads will need access to, and there are "global" values `threadIdx.x` and others that provide information about the currently executing thread and its block. In this way, the CUDA approach to programming feels reversed from normal serial programming: Each thread receives global data as parameters and has its own parameters (which determine, for instance, which data element this thread will process) stored in a global, non-obvious location.

Navigating and implementing convolutions in cuDNN, however, was hugely painful. cuDNN provides opaque C-object-like API calls with dozens of mysterious parameters, and it was only by following an online tutorial very precisely that we were able to run convolutional passes.

Reflecting on the experience of implementing convolutions through these disparate approaches, we came to some broad conclusions. The abstractions provided by PyTorch can be a powerful tool, but is only an obvious win when the problem structure naturally fits into the problem type assumed by PyTorch. That is, PyTorch is a powerful hammer, and with it you have to be careful when hitting non-nail objects. In fact, while implementing Task B, we found that it was faster to adapt our CUDA convolution kernel function to the complex nonlinear operation compared to adapting our PyTorch code to solve a fundamentally non-matrix computation using matrices. In both development time and runtime, we paid heavily in PyTorch when solving Task B. The cuDNN API felt too opaque to be immediately useful, and didn't provide a meaningful runtime speedup over custom CUDA kernels (in fact, cuDNN lost by over 10x!). CUDA feels more natural, provides more control, and allows for higher developer velocity compared to cuDNN and PyTorch, in large part because cuDNN and PyTorch seem to force you to solve your task using their way of thinking. Furthermore, CUDA required very little effort to transition from C++, and seems to provide nice high-level abstractions that not only allow for GPU-backed performance enhancements, but also stay out of the way enough to allow the programmer to solve a broad class of problems with standard C++ idioms and paradigms (save the odd parameter/-global information swap described above). Once the CUDA style of problem-solving is internalized, it's extremely straightforward to write parallel code.

Surprisingly, the most challenging portion of this project was not learning new languages and frameworks, but rather setting up dealing with the odd quirks of setting up the development environment, installing CUDA/cuDNN and managing packages in C++. A few of many examples: Installing the proper versions of `nvcc` (the NVIDIA compiler) and `nvprof` (the NVIDIA profiler) was a real pain. CUDA does not work with the latest macOS versions of llvm clang++, so you have to download old versions of Xcode and switch the system compilation over to them. This is made much more frustrating by the error not telling you which older versions of Xcode would work. cuDNN did not naturally play nicely with macOS headers or frameworks, spewing thousands of compiler errors because it had doubly defined macros in CoreServices. Most editors don't understand .cu file endings, even if they handle C++ just fine, so they will not syntax highlight or autocomplete. Installing OpenCV2 and linking against the libraries from `nvcc` turned into such an ordeal that we rewrote image reading and writing utilities ourselves, which indicates a design failure of the C++ package management and build process. Finally, if the machine has been running too long before you need to allocate GPU memory, the screen may have allocated all the memory on the GPU, so all allocations will start failing. We found that power cycling the screen caused these buffers to be freed. We found ourselves longing for Rust's excellent attention to install scripts, package management, and build systems.

4.2 Quantitative

In order to quantitatively assess our implementations, we benchmarked our code on two tasks. Task A consisted of convolving a 3x3 convolution kernel once over a 1920x1080 grayscale image of a deer. Each image pixel and each kernel element was represented as a float, and the convolution kernel was executed without padding (that is, the result image did not contain the husk of the original image). In Task B, we computed a complex nonlinear function for each pixel in order to explore the weakness of using PyTorch's higher-level matrix operations compared to the broader class of functions usable as CUDA kernels. In particular, for each pixel, we computed the sum of the pythagorean distance of each of 8 neighboring pixels across all channels, and used this value to interpolate between a dimmed version of the original color and the original color, with scaling constants controlled by tweakable parameters.

Our goal with Task A was to simply measure the cost of performing a matrix convolution on various hardware with various implementations. With Task B, we sought to challenge the PyTorch abstraction layers by posing a complex function that could be easily implemented as a CUDA kernel but hard to write as PyTorch operations.

In all cases (unless otherwise noted), each benchmark was run for 5,000 iterations, and the average runtime is reported. We ensured that the profiling only captured the time to perform the convolution and not the startup cost of image I/O and miscellaneous setup.

These benchmarks were performed on the following hardware provided by Christopher: (CPU) Intel Core i7-6700k (4 physical cores, 8 virtual at 4GHz); (GPU) GeForce GTX 980 Ti, with 2816

CUDA cores overclocked to 1.2GHz. Our CUDA is updated to the latest version, and cuDNN is on 7.04.

Table 1: Performance (ms) on Task A: Convolution of 3x3 kernel on an 1920x1080 image.

Processor Type	Build Configuration		Runtime (ms)
CPU	OpenMP	Opt. Level	
	Off	-O3	1.0878
	Off	-O2	1.1220
	Off	-O1	52.4910
	Off	-O0	88.5548
	On	-O3	6.5904
	On	-O2	6.7702
	On	-O1	16.8424
	On	-O0	30.2072
GPU/CUDA	Threads Per Block		
	32		0.11264
	64		0.076126
	128		0.076037
	256		0.076540
	512		0.077442
	1024		0.081313
	1 GPU thread, 1 block		470.12
GPU/cuDNN			
			0.97269
PyTorch	Running On		
	GPU		1.03
	CPU		24.3

Table 2: CPU Performance (ms) on Task A

Build Support	-O0	-O1	-O2	-O3
CPU Without OpenMP	88.5548	52.4910	1.1220	1.0878
CPU With OpenMP	30.2072	16.8424	6.7702	6.5904

Table 3: Best Performance (ms) on Task A

Build Configuration	Runtime (ms)
GPU (CUDA)	0.0760
GPU (cuDNN)	0.9727
PyTorch (CUDA on GPU)	1.0300
CPU (C++, -O3, no OpenMP)	1.0878

4.3 Analysis

From a performance perspective, many of our results were consistent with our hypotheses, but a few benchmark numbers surprised us. We will explore performance relationships within each implementation strategy and across implementation strategies.

Table 4: CUDA v. PyTorch Performance on Task B: Nonlinear Artistic Kernel

System	Runtime (ms)
CUDA-Backed C++ on GPU	3.2706
CUDA-Backed PyTorch on GPU	24.0806

Firstly, within the CPU-only implementation of C++ convolutions (see Table 2, the performance varied based on whether OpenMP was enabled and on the optimization level selected. Recall that OpenMP is the compiler directive that instructs the CPU to spawn threads for each invocation of a for loop, implicitly asserting that each execution of the loop is independent. With OpenMP disabled, the performance monotonically increases as higher optimization flags were used. Notably, there is an enormous jump from -O1 (about 50ms) to -O2 (about 1ms). This massive improvement is due in part to the fact that -O2 enables `-slp-vectorize` which allows AVX vector instructions, special instructions that allow for superword-level parallelism by operating on 256 bits at a time. When OpenMP is enabled, threads are spawned at the start of the operation to parallelize the task. At -O0 and -O1, OpenMP beat OpenMP, but at higher optimization levels, the non-OpenMP version won. We hypothesize that this is due to the higher fixed startup costs of thread management on a CPU, and that OpenMP would outperform non-OpenMP even at high optimization levels given sufficiently large inputs.

Our custom CUDA kernels, running on the GPU, blew away the competition. When launching a CUDA kernel, the programmer specifies the number of threads to a block (usually a multiple of 32), and the number of blocks to launch. In general, we instruct CUDA to launch one thread per data element, which in this case is per-pixel. However, the memory coalescing behavior of CUDA's access to unified memory within a block means that memory efficiency can depend on the size of the groups. We found a local optimum at 128 threads per group, as shown in the second section of Table 1. Threads were allocated in row-major order across the image - our attempts to cleverly have each thread compute an memory-optimal pixel location (each thread group processes a square, rather than a row) resulted in about a 50x slowdown. We also ran the convolution using just one GPU thread responsible for all of the computations, which was over 6,000x slower than the parallel computation, and slower than the single-threaded CPU approaches as well (due to the overhead of memory and the smaller cache size on a GPU core). Furthermore, it is worth noting that `nvcc` defaults to -O3, so these results are comparable with the final column of Table 2.

It is also interesting to compare this performance to the theoretical maximum performance. The memory bandwidth of the bus on our GPU processor is 336GB/s, so we can compute the minimum possible time to read every pixel once and write every pixel once, in a 1920x1080 image. With 2 I/O ops of 4 bytes per pixel (the size of a float), we see a minimum time of $\frac{2 \cdot 4 \text{ bytes} \cdot (1920 \cdot 1080)}{336 \cdot 2^{30} \frac{\text{bytes}}{s}} \cdot \frac{1000ms}{1s} =$ 0.04ms, which is only 2x faster than our CUDA implementation's performance. How is this possible? The GPU threads can be running almost always, due to the extremely low thread-switching cost, while waiting for data to come in.

The cuDNN implementation leverages NVIDIA's deep neural library but pays for this power in the form of awkward abstractions. We had hypothesized that cuDNN would seriously beat any implementation we could concoct, due to the fact that NVIDIA has highly optimized their library. In fact, the 64 thread-per-block hand-rolled CUDA code outperformed cuDNN by more than a factor of 10x. This indicates to us that the abstractions they impose, such as requiring that all input matrices have descriptors set up, can slow the overall runtime of the pipeline. Although the descriptors form a setup abstraction, we imagine that there are similarly slow confounding abstractions in the convolution layer.

PyTorch allows the programmer to move matrices and matrix operations to and from CUDA at will, so we compared a PyTorch convolution on GPU and CPU. After navigating the abstraction boundaries between Tensors and batch Variables with some difficulty, the PyTorch GPU-backed 2D convolution beat the CPU-backed 2D convolution by a factor of almost 24x. This is not too surprising, given the strengths of GPUs.

Lastly, we observed drastically different performances on the Task A benchmark across our four different implementation strategies, as demonstrated in Table 3. Our basic CUDA implementation, when saturating the GPU, outperformed cuDNN, PyTorch, and single-threaded C++ on CPU by between 12x and 15x. This is partially due to the direct memory management and data access performed by each thread. With plain C++ annotated to run on the GPU, we utilize almost no implementation abstractions (save for unified memory). As a result, the code feels messy but pays for none of the abstraction layers imposed by cuDNN and PyTorch (both of which are also running on the GPU). The fact that GPU-backed C++ code trounces CPU-backed C++ code is a further indication of the incredible computing power of GPUs.

4.3.1 Task B. On Task B (layered convolutions on an RGB image to produce an artistic result), we were tasked with computing a nonlinear operation for each pixel, one that is not easily transferable to standard matrix convolutions. Therefore, using CUDA we were able to compute all operations at once, invoking one thread per data element (and, admittedly, asking that thread to do more work than a simple computation). However, PyTorch operates on matrices with built-in operations. You cannot write a Python function and have the GPU access it, because the Python code lives at a higher level than the executing CUDA code. Thus, the PyTorch implementation, while shorter, consists of several matrix operations, each of which invokes a pass using CUDA. Thus, we see that the hand-rolled CUDA code beats the PyTorch layer by almost a factor of 8, as shown in Table 4.

We wanted to think more about these strange interface boundary constraints. What's really driving the limitation? When calling into CUDA from Python, you can't call back from CUDA into custom Python functions, so any functional composition requires the application of distinct kernels in multiple CUDA passes. On the other hand, when everything is CUDA, the GPU can call into device functions written by you. This allows the programmer to intelligently handle memory sharing and function fusing in a way that is impossible in PyTorch. The fact that the CUDA code beat the PyTorch implementation confirms the cost of adopting a language barrier between CUDA and Python.

REFERENCES

[1] 2012. A Look Back at Single-Threaded CPU Performance. (2012). http://preshing.com/20120208/a-look-back-at-single-threaded-cpu-performance/
[2] 2013. University of Washington CSE471: GPU Architectures: A CPU Perspective. (2013). https://courses.cs.washington.edu/courses/cse471/13sp/lectures/GPUsStudents.pdf
[3] 2017. 1 Reason NVIDIA Investors Need to Worry. (2017). https://www.fool.com/investing/2016/12/09/1-reason-nvidia-investors-need-to-worry.aspx
[4] 2017. AMD Vega reviews, news, performance, and availability. (2017). https://www.pcgamesn.com/amd/amd-vega-gpu-specifications
[5] 2017. ARB Assembly Language. (2017). https://en.wikipedia.org/wiki/ARB_assembly_language
[6] 2017. BLAS History. (2017). https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms
[7] 2017. BLAS Spec. (2017). http://www.netlib.org/blas/
[8] 2017. Comparison of Deep Learning Software. (2017). https://en.wikipedia.org/wiki/Comparison_of_deep_learning_software
[9] 2017. CS242 Lecture 7.2: Parallelism. (2017). http://cs242.stanford.edu/assets/slides/07.2-parallelism.pdf
[10] 2017. CUDA History. (2017). https://en.wikipedia.org/wiki/CUDA
[11] 2017. cudNN Released. (2017). https://www.infoq.com/news/2014/09/cudnn
[12] 2017. DirectX. (2017). https://en.wikipedia.org/wiki/DirectX
[13] 2017. An Even Easier Introduction to CUDA. (2017). https://devblogs.nvidia.com/parallelforall/even-easier-introduction-cuda/
[14] 2017. GLSL Tutorial – Vertex Shader. (2017). http://www.lighthouse3d.com/tutorials/glsl-tutorial/vertex-shader/
[15] 2017. High Level Shading Language. (2017). https://en.wikipedia.org/wiki/High-Level_Shading_Language
[16] 2017. Inside Volta. (2017). https://devblogs.nvidia.com/parallelforall/inside-volta/
[17] 2017. Moore's Law. (2017). https://en.wikipedia.org/wiki/Moore%27s_law#/media/File:Moore%27s_Law_over_120_Years.png
[18] 2017. NVIDIA CUDA Programming Guide. (2017). http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
[19] 2017. OpenACC. (2017). https://www.openacc.org/
[20] 2017. OpenCL History. (2017). https://en.wikipedia.org/wiki/OpenCL
[21] 2017. OpenGL. (2017). https://en.wikipedia.org/wiki/OpenGL
[22] 2017. OpenMP. (2017). http://www.openmp.org
[23] 2017. OpenMP History. (2017). https://en.wikipedia.org/wiki/OpenMP
[24] 2017. Pentium D History. (2017). https://en.wikipedia.org/wiki/Pentium_D
[25] 2017. Using OpenCL on GTX gives slower computation compared to CPU. (2017). https://forums.khronos.org/showthread.php/13406-Using-OpenCL-on-GTX-give-slower-computation-compared-to-CPU-why
[26] Mark Silberstein. 2014. GPUs: High-performance Accelerators for Parallel Applications: The Multicore Transformation (Ubiquity Symposium). *Ubiquity* 2014, August, Article 1 (Aug. 2014), 13 pages. https://doi.org/10.1145/2618401

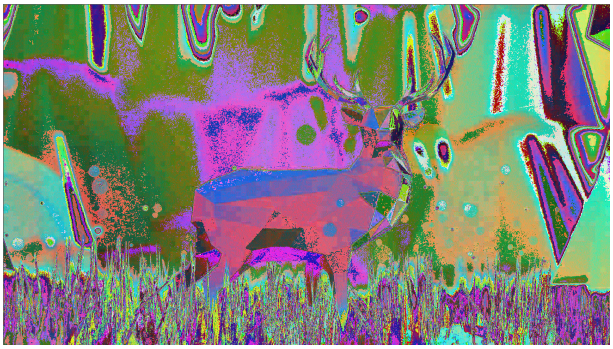
A WORK PERFORMED

Given that our goal was to learn as much about possible about GPU architectures, CUDA, cuDNN, and PyTorch in the context of evolving hardware constraints and programming language abstractions and decisions, we approached almost every task together. Equal work was performed by each project member.

B ART GALLERY

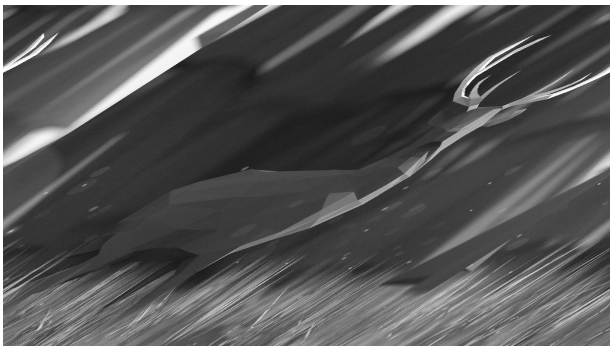
While working on this project, we generated some *interesting* pieces of artwork when encountering bugs in our implementation. We've decided to share some of the highlights (lowlights?) with you in Figure 12.

(a) Convolution of grayscale deer with random kernel.



(b) Convolution of color deer with random kernel.

(c) Everything is inverted! Now with more polka dots.



(d) Error reshaping C++ output, dimensions off-by-one.

Figure 12: Art Gallery