Lecture 21: GPU Computing

CS4787 — Principles of Large-Scale Machine Learning Systems

Over the past three lectures, we've been talking about the architecture of the CPU and how it affects performance of machine learning models. However, the CPU is not the only type of hardware that machine learning models are trained or run on. In fact, most modern DNN training happens not on CPUs but on GPUs. In this lecture, we'll look at why GPUs became dominant for machine learning training, and we'll explore what makes their architecture uniquely well-suited to large-scale numerical computation.

A brief history of GPU computing. GPUs were originally designed to support the 3D graphics pipeline, much of which was driven for demand for videogames with ever-increasing graphical fidelity. Important properties of 3d graphics rendering:

- Lots of opportunities for parallelism—rendering different pixels/objects in the scene can be done simultaneously.
- Lots of numerical computations—3d rendering is based on geometry.

The first era of GPUs ran a fixed-function graphics pipeline. They weren't programmed, but instead just configured to use a set of fixed functions designed for specific graphics tasks: mostly drawing and shading polygons in 3d space. In the early 2000s, there was a shift towards programmable GPUs. These programmable GPUs allowed for people to customize certain stages in the graphics pipeline by writing small programs called *shaders* which let developers process the vertices and pixels of the polygons they wanted to render in custom ways. These shaders were capable of very high-throughput parallel processing, since they would need to process and render very large numbers of polygons in a single frame of a 3d animation.

The GPU supported parallel programs that were *more parallel* than those of the CPU. But unlike multi-threaded CPUs, which supported computing *different functions* at the same time, the GPU focused on computing *the same function* simultaneously on multiple elements of data. That is, the GPU could run the same function on a bunch of triangles in parallel, but couldn't easily compute a different function for each triangle. This illustrates a distinction between two types of parallelism:

- **Data parallelism** involves the same operations being computed in parallel on many different data elements.
- **Task parallelism** involves different operations being computed in parallel (on either the same data or different data).

How are the different types of parallelism we've discussed categorized according to this distinction?

- **SIMD/Vector parallelism?**
- **Multi-core/multi-thread parallelism?**
- **Distributed computing?**

The general-purpose GPU. Eventually, people started to use GPUs for tasks other than graphics rendering. However, working within the structure of the graphics pipeline of the GPU placed limits on this. To better support general-purpose GPU programming, in 2007 NVIDIA released *CUDA*, a parallel programming language/computing platform for general-purpose computation on the GPU. (Other companies such as Intel and AMD have competing products as well.) Now, programmers no longer needed to use the 3d graphics API to access the parallel computing capabilities of the GPU. This led to a revolution in GPU computing, with several major applications, including:

- Deep neural networks (particularly training)
- Cryptocurrencies

A function executed on the GPU in a CUDA program is called a *kernel*. An illustration of this from the CUDA C programming guide:

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

This syntax launches N threads, each of which performs a single addition. Importantly, spinning up many threads like this could be reasonably fast on a GPU, while on a CPU this would be *way* too slow due to the overhead of creating new threads. Additionally, GPUs support many many more parallel threads running than a CPU.

- Typical thread count for a GPU: tens of thousands.
- Typical thread count for a CPU: at most a few dozen.

Important downside of GPU threads: they're data-parallel only. You can't direct each of the individual threads to "do its own thing" or run its own independent computation (although GPUs do support multi-task parallelism to a limited extent).

GPUs in machine learning. Because of their large amount of data parallelism, GPUs provide an ideal substrate for large-scale numerical computation. In particular, GPUs can perform matrix multiplies very fast. Just like BLAS on the CPU, there's an optimized library from NVIDIA "cuBLAS" that does matrix multiples efficiently on their GPUs. There's even a specialized library of primitives designed for deep learning: cuDNN. Machine learning frameworks, such as TensorFlow, are designed to support computation on GPUs. And training a deep net on a GPU can decrease training time by an order of magnitude. **How do machine learning frameworks support GPU computing?**