Lecture 19: Parallelism in Machine Learning

CS4787 — Principles of Large-Scale Machine Learning Systems

Recall from last time: four types of parallelism common on CPUs.

- Instruction level parallelism (ILP): run multiple instructions simultaneously.
- Single-instruction multiple-data parallelism (SIMD): a single special instruction operates on a vector of numbers.
- Multi-thread parallelism: multiple independent worker threads collaborate over a shared-memory abstraction.
- Distributed computing: multiple independent worker machines collaborate over a network.

...and programs need to be written to take advantage of these parallel features of the hardware.

What does this mean for machine learning?

In order to optimize the ML pipeline, we need to reason about how we can best use parallelism at each stage. Since we've been talking about training for most of the class, let's look at how we can use these types of parallelism to accelerate training an ML model.

Use #1: Using parallelism to make linear algebra fast. We can get a major boost in performance by building linear algebra kernels (e.g. matrix multiplies, vector adds, et cetera) that use parallelism to run fast. Since matrix multiplies represent most of the computation of training a deep neual network, this can result in a major end-to-end speedup of the training pipeline. This mostly involves ILP and SIMD parallelism, and (for larger matrices) it can also use multi-threading.

How do ML frameworks use this?

- Use BLAS and other highly optimized matrix/tensor libraries for arithmetic.
- Write algorithms in terms of matrix operations rather than vector operations as much as possible.
- Use broadcast operations rather than loops.[1]
- Contain explicitly parallelized implementations of some special common operations (such as convolutions for CNNs).

Benefits of using BLAS. You already saw this in the demo last time, but being able to take advantage of a highly-optimized mathematical kernel library gives a huge performance boost: potentially an order of magnitude above even `-O3` optimized C.

[1] In languages like Python where the compiler won't automatically vectorize. In languages like C, the compiler will (try to) automatically vectorize loops so broadcast operations are neither necessary nor supported.

What types of computations does BLAS support? Here's a non-exhaustive list:

- **axpy**: $y = ax + y$ for vectors x and y and scalar a.
- **dot**: $x^T y$
- **norm**: $\|x\|^2 = x^T x$
- **matrix-vector multiply-add**: $y = \beta y + \alpha A x$
- **matrix-matrix multiply-add**: $C = \beta C + \alpha A B$

...there are more operations supported too, such as specialized ops for symmetric and structured matrices. More fast linear algebra operations appear in more advanced libraries such as LAPACK.

Writing algorithms in terms of matrix operations. One important thing to know about parallel programs is that they need some sort of parallelism in the underlying algorithm to be able to work. Usually, this parallelism comes in the form of a *parallel loop*, where multiple iterations of a loop can be run (at least partially) in parallel. For example, look at this non-parallel `C` implementation of a matrix-vector multiply-add $y = \beta y + \alpha A x$ for $A \in \mathbb{R}^{m \times n}$, $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^m$.

```c
void mv_multiply_add(int m, int n, float* y, float* A, float* x) {
        for (int i = 0; i < m; i++) {
                float acc = 0.0; // product accumulator
                for (int j = 0; j < n; j++) {
                        acc += A[i*n+j] * x[j];
                }
                y[i] = beta * y[i] + alpha * acc;
        }
}
```

What operations within these loops could be run in parallel? We can think about this as determining how much parallelism is "available" for a program to use.

Simple heuristic: The more parallelism is available for use in an algorithm, the more an implementation can utilize the parallel resources of the hardware, and the faster it will run. For example, compare the following matrix-matrix multiply-add $Y = \beta Y + \alpha A X$ for $A \in \mathbb{R}^{m \times n}$, $X \in \mathbb{R}^{n \times p}$ and $Y \in \mathbb{R}^{m \times p}$.

```c
void mm_multiply_add(int m, int n, int p, float* Y, float* A, float* X) {
        for (int k = 0; k < p; k++) {
                for (int i = 0; i < m; i++) {
                        float acc = 0.0; // product accumulator
                        for (int j = 0; j < n; j++) {
                                acc += A[i*n+j] * x[j*p+k];
                        }
                        y[i*p+k] = beta * y[i*p+k] + alpha * acc;
                }
        }
}
```

The addition of an extra outer loop (over $k \in \{1, \ldots, p\}$) adds more operations to the algorithm that could be parallelized by an efficient implementation. **Do we expect this to benefit more or less from parallelism when compared to the matrix-vector multiply-add?**

Takeaway: if we can write our operations in terms of matrix-matrix computations (or linear algebra computations with more data in general) we can get an increased benefit from parallelism and by using hand-tuned kernels. You've probably observed this effect while working on the programming assignments in this course, and parallelism is part of the reason why this happens.

Parallelism from broadcast operations. You're all familiar with the broadcasting capabilities of numpy. But did you know that one way that these broadcasting operations can be faster...is because of parallelism? Compare the python code to implement the elementwise product of two vectors $x \odot y$:

```python
def elementwise_product(x, y):
        assert(len(x) == len(y))
        z = numpy.zeros(len(x))
        for i in range(len(x)):
                z[i] = x[i] * y[i]
        return z
```

with the python code to do this using broadcasting

```python
def broadcast_elementwise_product(x, y):
        return x * y
```

The latter is certainly simpler. But is it faster?

- Most of the speedup here is coming from getting rid of the python interpreter overhead...but some of it also comes from parallelism. Although numpy does not use multithreading for broadcast operations by default, it is benefitting from vectorization and ILP.
- Nevertheless you should always broadcast whenever possible.

Using specialized implementations for things like convolution layers. Most ML frameworks involve BLAS-like hand-tuned parallel implementations of operations that are common in deep learning.

- For example, the linear convolution layer of a CNN.

These implementations should be used whenever possible, and can mean big speedups for commonly used DNN architectures.

Of course, parallelism isn't just for making linear algebra fast, as we'll see...

Use #2: Using parallelism within an iteration of SGD, but not just for linear algebra. What can we parallelize within a single iteration of SGD?

Use #3: Using parallelism across iterations of SGD. Is the outer loop of SGD a parallel loop? **No,** there are inherent sequential dependencies. One idea: just let a bunch of worker threads run it in parallel as if it were a parallel loop, and use synchronization constructs like locks to prevent race conditions caused by the threads getting in each others' way. This can work, but is also slow because of the overhead of the synchronization.

An even crazier idea (called *asynchronous SGD* or HOGWILD!): just **have multiple worker threads run SGD in parallel without any synchronization**. Now race conditions could occur! But it turns out that in many cases this is fine, and the algorithm converges just as well (sometimes we can even prove this). **Intuition:** the noise/error from the race conditions just adds a small amount to the noise/error already present in the algorithm because of the random gradient sampling.

Use #4: Using parallelism in hyperparameter optimization. What opportunities are there to use parallelism to speed up hyperparameter optimization? What types of parallelism in the hardware are possible to use for these opportunities?

Use #5: Using parallelism elsewhere in the ML pipeline. Where else in the ML pipeline can we use parallelism to improve performance?

Takeaway: there's lots of opportunities to help machine learning scale by using parallelism. We can reason about parallelism by reasoning about the sources of parallelism in the hardware, the availability of parallelism in the algorithm, and the use of parallel resources in the implementation. Existing ML frameworks do a lot of this automatically for common tasks like deep learning training.