Matrix Multiplication on CPU (/matrix-multiplication-on-cpu.html)

Importance of matrix multiplication

Matrix multiplication is a fundamental operation in scientific computing. Here are just a few of countless uses of this fundamental linear algebra operation:

- compactly notating (https://en.wikipedia.org/wiki/Matrix_multiplication#System_of_linear_equations) systems of linear equations
- solving least squares problems, e.g. linear regression (both the analytical (https://towardsdatascience.com/analytical-solution-of-linear-regression-a0e870b038d5) and the iterative (https://towardsdatascience.com/step-by-step-tutorial-on-linear-regression-with-stochastic-gradient-descent-1d35b088a843) solution)
- characteristic equations in differential equations are based on eigenvalue decomposition
- dimensionality reduction (PCA (https://en.wikipedia.org/wiki/Principal_component_analysis))
- decompositions such as SVD (https://en.wikipedia.org/wiki/Singular_value_decomposition) (since $M = U\Sigma V^*$), which can be used e.g. in topic modeling (LSA (https://en.wikipedia.org/wiki/Latent_semantic_analysis))
- developing search result ranking algorithms (e.g. PageRank (https://en.wikipedia.org/wiki/PageRank#Python))
- finding the transitive closure (https://en.wikipedia.org/wiki/Transitive_closure) in graphs
- solving the all-pairs shortest path (http://www.cs.tau.ac.il/~zwick/Adv-Alg-2015/Matrix-Graph-Algorithms.pdf) (APSP) problem

Since this blog is about machine learning and mostly deep learning, the primary application of interest to the reader will most likely be the implementation of fully-connected layers (https://towardsdatascience.com/under-the-hood-of-neural-networks-part-1-fully-connected-5223b7f78528) in neural networks. At its core, the fully-connected layer is based on a generalized matrix multiplication (GEMM (https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms#Level_3)).

Convolutions can also be cast as matrix multiplications. For example, a classic approach for this is to preprocess the input using im2col (https://www.mathworks.com/help/images/ref/im2col.html), do a matrix multiplication, and then apply col2im (https://www.mathworks.com/help/images/ref/col2im.html). This is very inefficient, because the im2col matrix needs to be generated, and takes up a lot of extra memory and time. However, this approach can be made efficient using implicit im2col, resulting in an implicit GEMM convolution, which is implemented in some of the cuDNN (https://developer.nvidia.com/cudnn) algorithms for NVIDIA GPUs. Also, a $1x1$ convolution can be cast (https://datascience.stackexchange.com/questions/12830/how-are-1x1-convolutions-the-same-as-a-fully-connected-layer) as a matrix multiplication even without im2col.

Review of the mathematical operation

Before we focus on the code, let's review the basics. If we take a matrix A_{MxK} with M rows and K columns, we can multiply it by matrix B_{KxN} with K rows and N columns, obtaining matrix C_{MxN} with M rows and N columns. To obtain each value C_{ij}, we calculate a dot product between the ith row of A and the jth column of B.

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$$

$$B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix}$$

$$C = AB = \begin{bmatrix} a_{11} * b_{11} + a_{12} * b_{21} + a_{13} * b_{31} & a_{11} * b_{12} + a_{12} * b_{22} + a_{13} * b_{32} \\ a_{21} * b_{11} + a_{22} * b_{21} + a_{23} * b_{31} & a_{21} * b_{12} + a_{22} * b_{22} + a_{23} * b_{32} \end{bmatrix}$$

Machine Config

For any benchmark, we need to know what hardware and software configuration we're dealing with. The performance reported below was based on the following hardware and software:

- CPU: Intel® Core™ i7-7800X @ 3.50 GHz
- RAM: 32 GB DDR4 @ 2.4 GHz (G-Skill Ripjaws F4-2400C15-16GVR)
- Motherboard: ASUS PRIME X299-A motherboard
- OS: Ubuntu 16.04.6 LTS (Xenial Xerus)
- C++ Compiler: GCC 5.4.0 (also others for comparison, if explicitly stated)
- Compiler flags: `-O3 -march=native -funroll-loops` (unless specified otherwise)

Matrix dimensions for our experiments

Let's assume that M = N = K = 1024 for simplicity. This will let us easily reason about Big O (https://en.wikipedia.org/wiki/Big_O_notation), or rather, in this case, Big Θ (https://en.wikipedia.org/wiki/Big_O_notation#Family_of_Bachmann%E2%80%93Landau_notations) (tight bound, rather than the upper bound). For example, we know that the default multiplication algorithm takes $\Theta(MNK)$ time, which we can just write as $\Theta(N^3)$ since we assumed all dimensions to be the same.

The timings reported are averaged over 10 runs, since 1 run may not be representative due to background process (system services) load variability, possible interrupt handling by the CPU, etc.

Naive Iterative Algorithm

Let's try a simple 3-loop iterative algorithm in C++:

```
void naiveIterativeMatmul(
        float* const A,
        float* const B,
        float* const C,
        const int M,
        const int N,
        const int K) {

    for (int m = 0; m < M; m++) {
        for (int n = 0; n < N; n++) {
            for (int k = 0; k < K; k++) {
                C[m * M  + n] += A[m * M + k] * B[k * K + n];
            }
        }
    }
}
```

Since there's a `+=` operator, the C matrix needs to be first `memset` to zero. Note that I assume the memset is done in all reported performance results. It also turns out that the memset doesn't really matter for performance, since it's $\Theta($ N^2) time (the size of the C matrix), compared to $\Theta(N^3)$ time (compute time for the matrix multiplication). Empirical measurements match the complexity intuition, plus the `memset` component applies to all the algorithms except one, so it won't affect our comparisons.

Can we see any problems with the code above? The code runs extremely slowly. With the good compiler flags mentioned above, this implementation took 1,601 ms. With lower optimization levels, performance was even worse. For example, just using `-O3` without `-march=native`, I got 1,739 ms; `-O1` was 2,440 ms; and no flags took 4,496 ms. By comparison, a good implementation should take less than 40 ms on this hardware for the 1,024x1,024 matrices.

A few words about compiler optimization flags

As you saw above, high optimization levels result in a very significant performance improvement. Surely we wouldn't want to compare algorithms if there were plenty of scope for the compiler to further optimize code. On the other hand, `-march=native` is a bit risky.

Compiling for a given machine may be a bad assumption - after all, we wish the code to be reasonably portable (e.g. for machine code to work on both Skylake (https://en.wikipedia.org/wiki/Skylake_(microarchitecture)) and Haswell (https://en.wikipedia.org/wiki/Haswell_(microarchitecture)) chips). In practice, it's not as bad. For instance, we can build libraries with implementations for different architectures, and load the library compiled for the closest architecture once we know which CPU is used at runtime, e.g. via the use of the CPUID (https://en.wikipedia.org/wiki/CPUID) instruction. CPUID lets us identify the manufacturer, supported instruction sets (e.g. AVX (https://en.wikipedia.org/wiki/Advanced_Vector_Extensions)), the CPU architecture, etc. For an Intel-specific example, see here (https://software.intel.com/en-us/articles/intel-architecture-and-processor-identification-with-cpuid-model-and-family-numbers). To make libraries highly optimized yet portable, we can generate several shared library files, one for each architecture, and then `dlopen` the one that matches the runtime CPUID information. Also, for repetitive, long-running code, another option is to just-in-time compile (JIT (https://en.wikipedia.org/wiki/Just-in-time_compilation)) such code. There are added benefits that can be exploited when JITting, such as known dimensions which can be provided to the compiler as literals, rather than runtime values. Compile-time literals can aid the compiler with further optimizations, such as better loop unrolling (https://en.wikipedia.org/wiki/Loop_unrolling). For problems such as deep learning inference, once we know the tensor dimensions of a pre-trained model, JITting is a good way to squeeze the last bits of performance.

For now, unless stated otherwise, let's assume that we're using `-march=native` to have the compiler optimize as much as possible. This will allow us to focus purely on algorithmic differences.

Optimizing the naive iterative algorithm

Let's try to improve performance with 2 lines of code, while also addressing the issue of not having to `memset` matrix C. Note that the `+=` is a problem - we read from RAM and then update. If a matrix is big enough, we'll surely pollute the cache that could be used for reading matrices A and B instead. Since we only need to update one float at a time, we could create a single float variable, which the compiler would likely put in a register. This way we should go from 3 reads (A, B and C) and 1 write (C) from RAM to 2 reads and 1 write. Since the register access time is insignificant compared to RAM, we'd expect about 25% of the execution time to go away.

```
void iterativeMatmulRegisterSum(
        float* const A,
        float* const B,
        float* const C,
        const int M,
        const int N,
        const int K) {

    for (int m = 0; m < M; m++) {
        for (int n = 0; n < N; n++) {
            float sum = 0.0f;
            for (int k = 0; k < K; k++) {
                sum += A[m * M + k] * B[k * K + n];
            }
            C[m * M  + n] = sum;
        }
    }
}
```

This implementation took 1,070 ms to execute, which was 67% of the time it took to execute the original. We actually gained more time than we expected!

You might ask the question - why didn't the compiler perform this optimization itself? It turns out that it's an illegal optimization to make, considering that the compiler can't understand the semantics of the algorithm. All it knows is that it had to preserve the program semantics from the perspective of the memory model. We as developers know that we're multiplying matrices, so we are at liberty to do the accumulation locally and only push the final result to RAM. However, when the compiler generates the object file, it has no idea of who the users of this function would be. The data pointers are provided as arguments to this function, not allocated by the function itself. Therefore, it's conceivable that other threads might want to access the memory from the buffer represented by matrix C pointer's at the same time. While this doesn't make sense for this particular algorithm, there are other algorithms which multi-thread data reads and writes, and in such cases the lack of RAM visibility of the intermediate result would be a problem. Therefore, the code as originally written isn't semantically equivalent to the rewritten one from a memory model semantics point of view, even though as a matrix multiplication algorithm, it's exactly the same. The compiler only cares about memory semantics, so it can't optimize.

Of course, it's also possible that the CPU might have a write-through (https://en.wikipedia.org/wiki/Cache_(computing)#Writing_policies) cache, so that writes are captured in the cache. However, that may only buy us time for the next read - the write still requires the execution to wait for both the cache and RAM write to complete. If the cache is write-back

(https://en.wikipedia.org/wiki/Cache_(computing)#Writing_policies), we could proceed right after the cache is updated, we wouldn't have to wait for the RAM write. Still, even then, a cache write would be slower than a register write, so why not use a local variable, which would likely end up in a register?

Note that in a row-major programming language like C/C++, having k as the innermost loop is a problem. That's because while for matrix A, k represents columns, for matrix B, it represents rows. If these are big matrices, we're likely to incur cache misses on every read from matrix B! What can we do about this? As we discussed, k represents columns of matrix A. On the other hand, n indexes into the columns of matrix B. This means that interchanging the middle and inner loops so that we change the n index the most frequently (columns of B), followed by k (columns of A, but rows of B), followed by m (rows of A) will ensure the most cache hits for this basic implementation.

```
void iterativeMatmulLoopReorder(
        float* const A,
        float* const B,
        float* const C,
        const int M,
        const int N,
        const int K) {

    for (int m = 0; m < M; m++) {
        for (int k = 0; k < K; k++) {
            for (int n = 0; n < N; n++) {
                C[m * M  + n] += A[m * M + k] * B[k * K + n];
            }
        }
    }
}
```

The above code took only 63 ms! Not surprisingly, hitting the caches matters. This technique is known as loop reordering or loop interchange (https://en.wikipedia.org/wiki/Loop_interchange).

Static dimensions

Let's try one more thing. Let's assume that we have the ability to just-in-time compile a function optimal for given matrix dimensions. Say for instance that this is some hidden LSTM (https://en.wikipedia.org/wiki/Long_short-term_memory) state which we expect to use over and over for training or inference, so we can statically determnine the dimensions. Let's see what the impact of statically known dimensions is on performance. To do this, let's template the function:

```
template <int M, int N, int K>
void iterativeMatmulLoopReorderTempl(
        float* const A,
        float* const B,
        float* const C) {

    for (int m = 0; m < M; m++) {
        for (int k = 0; k < K; k++) {
            for (int n = 0; n < N; n++) {
                C[m * M  + n] += A[m * M + k] * B[k * K + n];
            }
        }
    }
}
```

Since we set M, N and K to equal 1,024, we instantiate the templated function with those values. The function ran for 53 ms.

While we're at it, let's also check NumPy (from Anaconda (https://www.anaconda.com), version 1.16.4):

```
>>> import numpy as np
>>> from time import time
>>> x = np.random.randn(1024, 1024).astype(np.float32)
>>> y = np.random.randn(1024, 1024).astype(np.float32)
>>> start = time(); np.dot(x, y); print((time() - start) * 1000)
```

NumPy ran for 23 ms. Note that we had to cast x and y to `np.float32` (single precision), because NumPy by default uses `np.float64` (double precision).

Unfortunately, our implementation is still 2.3x slower than the NumPy one. However, optimizing linear algebra used by NumPy, especially Intel's Math Kernel Library (MKL (https://software.intel.com/en-us/mkl)) in case of the Anaconda build, is the fruit of hundreds of developer-years, manual assembly coding, etc. I didn't expect to beat libraries that took so much effort to develop.

Note that it's also unreasonable to compare a generic-dimension matrix multiplication algorithm from libraries used by NumPy, PyTorch, etc., against a dimension-templated function, which will only work for this particular dimension. Still, even before templating, we went down from 1,600 ms to 63 ms, without any assembly hand-coding.

Cache behavior depending on matrix sizes

Let's assume M bytes of cache and B bytes per cache line, and consequently $\frac{M}{B}$ cache lines. For matrix dimensions, let's assume Since M=N=K, let's call them all N. Let's also note the precision of the data as P. Here, we don't care about precision directly, but about how much data instances of a given precision take. For example, double precision (https://en.wikipedia.org/wiki/Double-precision_floating-point_format) means 8 bytes (64 bits), single precision (https://en.wikipedia.org/wiki/Single-precision_floating-point_format) means 4 bytes (32 bits), and so on.

Case 1: $N > \frac{M}{BP}$

When we keep going down the rows of matrix B for a given column (because originally, the inner loop was indexed by k), then we may incur a cache miss on every iteration over B. This becomes a problem because even if A is cached, all computation will stall while fetching data for B. Since we have $\Theta(N^3)$ multiplications and additions while multiplying the matrix, if we incur a cache miss on every value of B, our cache misses with be $Q = \Theta(N^3)$.

Case 2: $\frac{\sqrt{M}}{P} < N <$c$\frac{M}{BP}$ for $1 < c < 0$

In this case, we'll only have cache misses for the matrix being indexed down the rows every BP bytes (the matrix dimension is NxN, but we're using $\frac{M}{BP}$ bytes per dimension, and $B << M$ (tall cache assumption). We will still have cache misses, but only every BP bytes. This means that our cache misses will go down to $Q = \Theta(\frac{N^3}{BP})$

Case 3: $\frac{\sqrt{M}}{P} < N$

In this case, everything fits in cache, since $N * N$ would be less than M. Of course, to have both A and B matrices in cache, we would have to assume that each of the matrices actually occupies at most half the cache. This was shown to be optimal by Hong and Kung (1981) (http://www.eecs.harvard.edu/~htk/publication/1981-stoc-hong-kung.pdf).

More on hardware caches

We briefly discussed how the naive algorithm had issues with caching, because of the loop order. We also covered the 3 cases of matrix dimensions relative to the cache size. Unfortunately, even loop reordering isn't ideal. For one, even if data is prefetched, we're not breaking up the problem to fit the cache well. A much better way to break up the processing in a way that would fit the cache is via tiling.

Before we get into tiling, let's review hardware caches.

First of all, hardware caches are supposed to be fast. These are small chunks of memory that sit on the same chip as the CPU, and use static RAM (SRAM) instead of dynamic RAM (DRAM) like the main memory that we're used to. SRAM uses flip-flops and is much faster than DRAM, which uses capacitors. Unfortunately, SRAM takes up a huge amount of chip real estate, and uses a lot of power, which is why main RAM is dynamic.

This is typical of any memory hierarchy in a computer, where we have a clear speed-capacity trade-off - registers are fast but tiny, L1 cache is slightly bigger but also a bit slower, DRAM is much larger but slower still, and SSDs have the highest capacity but are slower still. Here are some statistics (https://www.prowesscorp.com/computer-latency-at-a-human-scale/) in CPU clock cycles that show the memory access time depending on the device being accessed.

Of course, these numbers may vary, they are just here to give you a general idea of the orders of magnitude.

Device	Access Time (ns)	CPU cycles
register	0.4	1
L1 cache	0.9	2.25
L2 cache	2.8	7
L3 cache	28	70
DRAM	100	250
NVMe SSD	25,000	62,500
SATA SSD	100,000	250,000

Clearly caching matters! If a CPU is sitting idle waiting thousands of cycles for memory access, it's not spending those cycles doing any computation. Needless to say, optimizing cache use will be key to improving performance of our matrix multiplication algorithm.

Let's cover some basics of computer caches. We won't go into too much detail, so feel to review more at your own leisure here (https://en.wikipedia.org/wiki/Cache_(computing)).

Caches aren't exactly ideal in practice. They tend to use heuristics for eviction policies, they aren't fully associative (https://en.wikipedia.org/wiki/Cache_placement_policies), etc. However, it turns out that practical caches can be assumed to be ideal for practical code analysis to simplify things. An ideal cache has several characteristics:

- It is fully associative. This means that there exists a single cache set with multiple cache lines, and a given memory block can occupy any of them. If we take a cache line to be of size 1 of some units, this would mean that the cache can be viewed as a matrix with m rows and 1 column.
- The cache has an omniscient residency/eviction policy. This assumes a perfect policy not just accounting for the past, but also the future, anticipating what may be needed next.

It turns out that while such assumptions are unreasonable, they can approximate a practical cache relatively well. The least-recently used assumption can replace the assumption of omniscience. Sleator and Tarjan (1985) demonstrated (https://www.cs.cmu.edu/~sleator/papers/amortized-efficiency.pdf) that given an omniscient cache with M bytes and Q cache misses, it can be replaced with an LRU cache with 2M bytes and 2Q cache misses. Therefore, the omniscience assumption can be approximated by a real, practical cache up to a small, known constant factor. The full associativity assumption is also true up to a point, e.g. L1 caches in recent Intel Core i7 CPUs are 8-way set associative, etc.

We also need to make a "tall cache" assumption, which means that the number of bytes per cache line is much less than the number of cache lines. Formally, if a cache line has B bytes and the cache has M bytes total (giving us $\frac{M}{B}$ cache lines), we assume that $B^2 < c * M$ for $c \leq 1$. This is also a reasonable assumption for practical caches. For example, on a Core i7-7800K, there's an L1 cache of 32 KB per core, and each cache line is 64 bytes. In this case, $B^2 = 4,096$, which is 4KB, which is much less than 32 KB.

Why do we need to make a tall cache assumption? Essentially, this is because we need to be able to have multiple smaller items that are able to be stored in the cache, rather than fetching one gigantic item when we store. For instance, let's imagine that we had a single cache line of 32 KB, and we wanted to read a matrix down a row. A single read that resulted in a cache miss would result in storing contiguous 32 KB of memory in the cache, but we only needed one element per row, so the other elements for that row would be polluting the cache. Ideally, we'd want the cache line to be small, since then we could store small chunks from various locations in memory. However, the cache line has to be big enough to make contiguous reads efficient, by filling up the memory bus when the read is done. We will see more of that in the case of GPUs.

Given all the above, let's assume either an omniscient or LRU cache, as convenient, that is fully or nearly fully associative, with a tall cache assumption.

Iterative Algorithm with Tiling

Let's break our large matrix into sub-matrices or tiles. If these tiles are small enough to fit in the cache (say L1), then we shouldn't have to care whether the loops are reordered or not. Of course, since each x, y coordinate in the output matrix C has to be the result of multiplying an entire row vector i from matrix A by the entire column vector j of matrix B, taking small square tiles out of the matrix from both matrices will only give us partial results for each output coordinate, so we'll have to sum up the products coming out of multiplying individual tiles. For example, to calculate the blue tile in matrix C in the figure below, we'll need to take two partial results from tile multiplications.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

$$+ \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

$$= \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix}$$

Remember that we'll need to tune the implementation to our actual CPU. I have an Intel® Core™ i7-7800X. We can Google around and find (http://www.cpu-world.com/CPUs/Core_i7/Intel-Core%20i7%20i7-7800X.html) the following details about that CPU:

- L1 cache: 32 KB / core
- L2 cache: 1 MB / core
- L3 cache: 8.25 MB shared cache

Let's check out the following code:

```
template <int T>
void naiveIterativeMatmulTiled(
        float* const A,
        float* const B,
        float* const C,
        const int M,
        const int N,
        const int K) {

    for (int m = 0; m < M/T; m += T) {
        for (int n = 0; n < N/T; n += T) {
            for (int k = 0; k < K/T; k += T) {
                for (int mt = m; mt < m + T && mt < M; mt++) {
                    for (int nt = n; nt < n + T && nt < N; nt++) {
                        for (int kt = k; kt < k + T && kt < K; kt++) {
                            C[mt * M + nt] += A[mt * M + kt] * B[kt * K + nt];
                        }
                    }
                }
            }
        }
    }
}
```

Note that now, we don't need to reorder the loops, since cache hits will be ensured by the small size of the tiles in the inner loops.

The above kernel combines the flexibility of runtime dimensions for M, N and K, while compiling for a fixed tile size. Thus, the compiler can make some static optimizations for a fixed T, while not requiring recompilation as matrix dimensions change. This way, one can look up the best tile size for a given CPU and run that particular kernel.

The above code leaves room for improvement, since we keep recomputing the inner loop boundaries every time. Let's refactor the boundary conditions as follows:

```
template <int T>
void naiveIterativeMatmulTiled(
        float* const A,
        float* const B,
        float* const C,
        const int M,
        const int N,
        const int K) {

    for (int m = 0; m < M/T; m += T) {
        for (int n = 0; n < N/T; n += T) {
            for (int k = 0; k < K/T; k += T) {

                const int minMt = std::min(m + T, M);
                const int minNt = std::min(n + T, N);
                const int minKt = std::min(k + T, K);

                for (int mt = m; mt < minMt; mt++) {
                    for (int nt = n; nt < minNt; nt++) {
                        for (int kt = k; kt < minKt; kt++) {
                            C[mt * M + nt] += A[mt * M + kt] * B[kt * K + nt];
                        }
                    }
                }
            }
        }
    }
}
```

The tile size indicated by the previous cache discussion, for optimal hits of L1, is $\sqrt{32 * 1024} \approx 181$. Let's round it down to 180, and divide by 4, since we're using single precision floating point, which is 4 bytes per scalar. This would give us a tile size of 45. Since we have 3 matrices we're dealing with, we should divide that by 3, giving 15. The closest power of 2 to 15 is 16, which we will use. Unfortunately, the performance of this function with the "optimal" cache size was rather poor: execution took 526 ms. Other tile sizes were mcuh worse. It looks like our tiled solution isn't nearly as good as conventional wisdom (texbooks, papers) suggest.

Problems tiling on the CPU

Tiling is not cache-agnostic. This means that if we pick a tile size that's optimal for a CPU with a given cache size and then end up running the same code on a CPU with a smaller cache size (assuming the same instruction set), the performance on the CPU with the smaller cache might end up abysmal. Note that we might essentially end up with the case of the $\Theta(N^3)$ cache misses. The other headache is that the number of unique implementations would need to be dependent on all cache hierarchies. The tiling discussion above is based on one level of caching, but modern CPUs have 3 levels (L1, L2, L3). This means we might need to add extra loops to cover tiles for L2, not just L1, and do the same for L3. Too many loops end up having control flow costs, which may outweigh the benefits of 3 levels of cache-aware tiling.

Another major concern with cache size-tuned algorithms is that they make the assumption that nothing else can run on the CPU at any given time. This might be an appropriate assumption to make if we were using a microcontroller, but on any platform with a running operating system, it's unrealistic. Even if the CPU does not need to run any heavy

processes in the background, the kernel itself needs to context switch between OS housekeeping tasks (say interrupt servicing) and running the application. The context switch between the kernel and the application can result in purging caches to make room for the task being done by the kernel. The problem is obviously worse if the kernel needs to context switch not just between itself and our matrix multiplying application, but other userspace (https://en.wikipedia.org/wiki/User_space) applications. This for instance is a serious concern if we try to load up the cache with a lot of data at once, since it may be evicted due to a context switch.

Overall, the fact that tiling isn't all that great, because it's not cache-agnostic, it assumes that there are no context switches, and the performance seems to reflect those shortcomings.

Compiler considerations

I had a gut feeling that perhaps the compiler wasn't doing a good job with tiled code, so I decided to try newer versions of GCC. Unfortunately, they ended up being worse, particularly GCC 8. This is likely due to the compiler patches introduced to combat Spectre (https://en.wikipedia.org/wiki/Spectre_(security_vulnerability))-based attacks. However, I decided to check other compilers available as default packages for my OS (Ubuntu 16.04.6 LTS). Of course, Clang is an alternative to GCC.

Let's compare the results:

GCC 5.4.0

- Loop reordering (non-templated MNK): 63 ms
- Loop reordering (templated MNK): 53 ms
- Tiling (best tile size=16): 526 ms

Clang 3.8.0

- Loop reordering (non-templated MNK): 72 ms
- Loop reordering (templated MNK): 69 ms
- Tiling (best tile size=16): 79 ms

These results are stunning! It's amazing how drastically different the performance of tiled code is when we switch compilers. Clang clearly wins by a huge margin in the tiling case, but GCC was better for the loop reordering case.

In general, the performance gain of simple loop reordering is very stable, across both different compilers and CPUs, while the gains from tiling have extremely large variation. Even in the clang case where tiling is doing pretty well, changing the tile size resulted in a performance cliff. For example, changing tile size from 16 to 8 resulted in performance degradation from 79 ms to 336 ms, and a tile size of 32 took 629 ms.

The instability of tiling performance across both tile sizes and compilers is concerning since we may have different CPUs to take care of, and we would need to know optimal sizes for each. Even then, loop reordering is still a bit better. The lesson here is: don't believe textbooks, measure performance for everything you do. Or, as the Cold War saying goes: "trust, but verify (https://en.wikipedia.org/wiki/Trust,_but_verify)." We'll see in another blog post that tiling is key to GPU matrix multiplication performance, but for CPUs, we have many other and possibly better options.

Can we do any better than loop reordering? So far, we haven't even discussed optimizing matrix multiplication for multi-core CPUs.

Performance summary

Let's summarize what we achieved so far for the 1,024x1,024 matrix multiplication. In addition to listing time, let's list floating-point operations per second (FLOPs/s).

The total number of FLOPs for 1,024x1,024 matrix multiplication is $2MNK$, or $2*1024^3$, i.e. $2*2^{30}$, i.e. 2 GibiFLOPs (https://en.wikipedia.org/wiki/Binary_prefix), ~2.14 GigaFLOPs (GFLOPs). We can get FLOPs/s by dividing the total number of FLOPs by the number in seconds the implementation takes. Let's check out the summary now.

Algorithm	time (ms)	GFLOPs/s	Speedup rel. to naive
naive	1,601	1.34	1.00
register accum	1,070	2.00	1.50
loop reordering	63	34.09	25.41
reordering + templ.	53	40.52	30.21

Note that the Core i7-7800X CPU has 6 cores, each of which has 2 AVX units, which can do 16 floating point operations per clock. Since AVX supports fused multiply-add instructions, for matrix multiplication, we can count 32 FLOPs/cycle. There are 2 AVX units per core. If we multiply it all out (3.5*10^9 Hz * 32 FLOPs/Hz/AVX unit * 2 AVX units/core * 6 cores), we'll see that the CPU is capable of max arithmetic throughput of 1,344.2 GFLOPs, while we got at most 40. Note though that there are several factors that prevent all this compute power from ever getting used:

- limited data bandwidth between CPU and RAM
- during cache misses, we have to deal with the RAM latency
- CPU/OS/other process context switches (and possible cache evictions)
- control flow is required to check boundary conditions, using up cycles that could have gone to the computation
- problem size (1,024x1,024 matrix multiplication is a small problem for the CPU to warrant multi-threading across cores)

Still, let's imagine that we could at least get the top single-threaded performance on 1 core, which would have given us 224 GFLOPs. Even that ends up being unrealistic for a small problem, and we end up getting about a 2x speed-up over our simply-optimized best case. Still, if we assume that we're multiplying much bigger matrices, the gap between manually optimized routines and ones provided by Intel as part of their Math Kernel Library (MKL (https://software.intel.com/en-us/mkl)) can only grow. The bottom line is: this is an post about how to think about optimizations, I don't recommend that you hand-tune linear algebra routines. That's why optimized libraries such as Intel's MKL, ARM's Performance Libraries (https://developer.arm.com/tools-and-software/server-and-hpc/arm-architecture-tools/arm-performance-libraries) and other vendor-specific libraries exist.

Divide-and-conquer algorithm

The divide-and-conquer algorithm is cache-agnostic, in that it doesn't depend on the tile size. Since it's based on the fork-join (https://en.wikipedia.org/wiki/Fork%E2%80%93join_model) paradigm, it can also be implemented in a way that exploits multi-core processing.

Here's a mathematical representation of this concept. Let's divide both matrices into 4 submatrices (https://en.wikipedia.org/wiki/Block_matrix), i.e. partitioning vertically and horizontally.

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

$$B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$C = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

We can of course break up these matrices recursively and stop at the base case of having to multiply two elements together. This in theory would ensure the cache agnostic property, since work would not depend on any tile size. Of course, the performance of an algorithm with a trivial base case would be horrendous, since the overhead of function calls would overshadow any data fetches and compute.

Clearly, bigger base cases would result in better performance. However, if you think about it, a larger base case essentially ends up behaving like a tile, so the promise of being cache-agnostic is false if we have real performance considerations. That said, divide and conquer lets us at least use multiple cores, so we can combine that with a tile-based or loop-reordered base case and multi-core processing to further improve performance. For now, let's write down the algorithm for partitioning each matrix in half both horizontally and vertically.

1. Partition the matrices
   - partition A into A_{11}, A_{12}, A_{21}, A_{22}
   - partition B into B_{11}, B_{12}, B_{21}, B_{22}
   - partition C into C_{11}, C_{12}, C_{21}, C_{22}
2. Fork (https://en.wikipedia.org/wiki/Fork%E2%80%93join_model) the work:
   - fork task 1: C_{11} += A_{11}*B_{11}
   - fork task 2: C_{12} += A_{11}*B_{12}
   - fork task 3: C_{21} += A_{21}*B_{11}
   - fork task 4: C_{21} += A_{21}*B_{12}
3. Wait for (join (https://en.wikipedia.org/wiki/Fork%E2%80%93join_model)) dependent tasks and launch new ones:
   - wait for task 1; fork task 5: C_{11} += A_{12}*B_{21}
   - wait for task 2; fork task 6: C_{12} += A_{12}*B_{22}
   - wait for task 3; fork task 7: C_{21} += A_{22}*B_{21}
   - wait for task 4; fork task 8: C_{22} += A_{22}*B_{22}
4. Wait for (join (https://en.wikipedia.org/wiki/Fork%E2%80%93join_model)) tasks 5-8 to complete.

Note that there's some serialization here - tasks 5-8 need to wait for tasks 1-4 to complete, e.g. task 5 needs to wait for task 1 to complete. We chose this because it ensures that the fork-join approach doesn't use any extra memory. If instead we used scratch space for tasks 5-8 so there would be no write-write conflicts, we could have launched all 8 tasks at the same time, and then we could do elementwise addition of the output of task 5 with the output of task 1 and so on. Since elementwise addition is $\Theta(N^2)$ time while matrix multiplication is $\Theta(N^3)$ time, we would have gained further acceleration, at the cost of using extra memory. We could also use no additional memory, at the cost of making all memory writes atomic, which doesn't buy us much relative to the first scenario of 2 batches of tasks, since it serializes memory access.

Let's see how this might look in practice, along with multi-threading. Note that I'm just considering the 4-way partitioning, but the system could have more cores, say 8, 16 or more. In practice, we need to make this algorithm adapt to the core count of the system. For now, however, the hard-coded 4-way split should illustrate the point.

First, let's modify our loop-reordered algorithm so that it can take pointer offsets and work on independent sub-ranges of the flat memory, i.e. on the partitions of the matrix. The arguments such as `loM`, `hiM` and so on will guard the ranges that we're interested in for a particular subtask. Note that with these low/high values, we need both the maximum M and K, but not N, since we don't need it for dimension striding. We still need the low and high values of K though to guard the processing of the submatrix.

```
void iterativeMatmulLoopReorderSubMatrix(
        float* const A,
        float* const B,
        float* const C,
        const int M,
        const int K,
        const int loM,
        const int hiM,
        const int loN,
        const int hiN,
        const int loK,
        const int hiK) {

    for (int m = loM; m < hiM; m++) {
        for (int k = loK; k < hiK; k++) {
            for (int n = loN; n < hiN; n++) {
                C[m * M + n] += A[m * M + k] * B[k * K + n];
            }
        }
    }
}
```

Now, let's schedule the work on different threads asynchronously, using C++11's `std::async`. Let's schedule work immediately using `std::launch::async`, and then only wait for the future to return a value if we have subsequent work to do on those particular memory addresses of the C matrix. For example, in the partitioning, we see that $C_{11} = A_{11} * B_{11} + A_{12} * B_{21}$. This means that in order to avoid allocating extra memory, we need to run one of the two inputs to the sum first, then only start calculating the second once the first one finishes. Here's an example of how the above sub-matrix computation could be scheduled in this fork-join (https://en.wikipedia.org/wiki/Fork%E2%80%93join_model) scenario.

```
void mtMatmul(float* const A,
              float* const B,
              float* const C,
              const int M,
              const int N,
              const int K) {

    const int m2 = M/2;
    const int n2 = N/2;
    const int k2 = K/2;

    // First part of C11: A_11*B_11
    auto c11P1 = std::async(
            std::launch::async,
            iterativeMatmulLoopReorderSubMatrix,
            A, B, C, M, K,
            0, m2, 0, n2, 0, k2
            );

    // First part of C12: A_11*B_12
    auto c12P1 = std::async(
            std::launch::async,
            iterativeMatmulLoopReorderSubMatrix,
            A, B, C, M, K,
            0, m2, n2 + 1, N, 0, k2
    );

    // First part of C21: A_21*B_11
    auto c21P1 = std::async(
            std::launch::async,
            iterativeMatmulLoopReorderSubMatrix,
            A, B, C, M, K,
            // loM, hiM, loN, hiN, loK, hiK
            m2 + 1, M, 0, n2, 0, k2
    );

    // First part of C22: A_21*B_12
    auto c22P1 = std::async(
            std::launch::async,
            iterativeMatmulLoopReorderSubMatrix,
            A, B, C, M, K,
            m2 + 1, M, n2 + 1, N, 0, k2
    );

    // Second part of C11: A_12*B_21
    c11P1.wait();
    auto c11P2 = std::async(
            std::launch::async,
            iterativeMatmulLoopReorderSubMatrix,
            A, B, C, M, K,
```

```
            // loM, hiM, loN, hiN, loK, hiK
            0, m2, 0, n2, k2 + 1, K
    );

    // Second part of C12: A_12*B_22
    c12P1.wait();
    auto c12P2 = std::async(
            std::launch::async,
            iterativeMatmulLoopReorderSubMatrix,
            A, B, C, M, K,
            // loM, hiM, loN, hiN, loK, hiK
            0, m2, n2 + 1, N, k2 + 1, K
    );

    // Second part of C21: A_22*B_21
    c21P1.wait();
    auto c21P2 = std::async(
            std::launch::async,
            &iterativeMatmulLoopReorderSubMatrix,
            A, B, C, M, K,
            m2 + 1, M, 0, n2, k2 + 1, K
    );

    // Second part of C21: A_22*B_22
    c22P1.wait();
    auto c22P2 = std::async(
            std::launch::async,
            &iterativeMatmulLoopReorderSubMatrix,
            A, B, C, M, K,
            m2 + 1, M, n2 + 1, N, k2 + 1, K
    );

    // sync threads before returning
    c11P2.wait();
    c12P2.wait();
    c21P2.wait();
    c22P2.wait();

}
```

The above setup looks a bit daunting, so let's see what benefits we get.

Note that since we needed to add threading support, we needed to link pthreads (`-pthread`). After all, even though C++11 has a platform-independent threading API, it's backed by the system-native threading support - in case of Linux and other POSIX (https://en.wikipedia.org/wiki/POSIX) systems, it's backed by POSIX threads (pthreads (https://en.wikipedia.org/wiki/POSIX_Threads)).

Just to clarify, the CPU I mentioned above has 6 cores, but given the 4-way parallelism of the code, we'll only be using 4 out of 6 cores. For 1024x1024 matrices, the gain wasn't large - we went from 70 to 60 ms. This is likely because thread scheduling isn't cheap, so for small problems, the cost of the initial overhead wasn't amortized. Switching to

larger (4096x4096) matrices resulted in an observable gain - execution time went from 2058 ms to 1060 ms - that's a speed-up of about 2x! The more work we have to do per thread, the more amortized the threading overhead should become, and the closer to a linear speed-up we would get.

What we didn't cover

We clearly didn't cover vector processing on the CPU in detail, using instruction sets such as AVX (https://en.wikipedia.org/wiki/Advanced_Vector_Extensions) on x86 and NEON (https://en.wikipedia.org/wiki/ARM_architecture#Advanced_SIMD_(NEON)) on ARM. Fortunately, in practice, using `-O3 -march=native -funroll-loops` tends to benefit from some vectorization generated by the compiler, although usually it's not nearly good enough unless we know all dimensions statically. Hand-written assembly code would tend to do better.

Also, there exist algorithms that do less work than $\Theta(MNK)$, such as Strassen's algorithm (https://en.wikipedia.org/wiki/Strassen_algorithm), but this post is already getting long, so let's discuss it some other time. Strassen's algorithm reduces time complexity from $\Theta(N^3)$ to $\Theta(N^{2.8074})$. Strassen's algorithm can have worse numerical accuracy than the usual "brute force" algorithm. There are other algorithms which has even better complexity than Strassen's algorithm, such as Coppersmith-Winograd (https://en.wikipedia.org/wiki/Coppersmith%E2%80%93Winograd_algorithm) with complexity $\Theta(N^{2.375477})$, however these algorithms have gigantic constant factors, that are big enough that the better big-O performance only pays off for big enough problems that they don't fit on modern computers (that is, they are galactic algorithms (https://en.wikipedia.org/wiki/Galactic_algorithm)).

We also didn't cover distributed, communication-avoiding matrix multiplication algorithms, such as Cannon's algorithm (https://en.wikipedia.org/wiki/Cannon%27s_algorithm). That would be a topic with enough content for yet another long blog post.

Published

Aug 8, 2019

Category

linear algebra (/categories.html#linear-algebra-ref)

Tags

- computational linear algebra 1 (/tags#computational-linear-algebra-ref)
- GEMM 1 (/tags#gemm-ref)
- linear algebra 1 (/tags#linear-algebra-ref)
- matmul 1 (/tags#matmul-ref)
- performance 1 (/tags#performance-ref)
- scientific computing 1 (/tags#scientific-computing-ref)
- systems 1 (/tags#systems-ref)

Contact