Game Plan

Max Ruth

September 30, 2020

1 Description

This document summarizes my thoughts on how to optimize the code. The general work flow should probably be the same as what David recommended, namely:

- Optimize small matrix-matrix multiplication.
- Optimize the "blocking procedure", and deal with all of the problems that bigger problems introduce.

2 block_dgemm

My impression is that we should have some sort of script for optimizing a function named `block_dgemm`. This function would be

```
void block_dgemm(const int M, const int N, const int K,
                 const double* restrict A, const double* restrict B, double* restrict C)
```

where `M`, `N`, and `K` are the lengths `i`, `j`, and `k` loop over respectively. I imagine that it would be ideal to have this defined for fixed sizes M_B, N_B, and K_B where it is assumed that $A \in \mathbb{R}^{M_B \times K_B}$, $B \in \mathbb{R}^{K_B \times N_B}$, and $C \in \mathbb{R}^{M_B \times N_B}$. The reason I think this is ideal is because it would naturally lend itself to aligned memory, where A and B would be stored in $M_B \cdot K_B$ and $K_B \cdot N_B$ chunks respectively. As of right now, it is unclear to me if we would prefer C to be stored in an aligned format and copied over at the end, or if it is easier to just edit it in place.

Here is a fullish list of design decisions

- Version of C. Currently it is being compiled via `-std=gnu99`, which is probably for the best.
- Compiler. David used `gcc-10` for the kernel example. This is likely a good decision, but it is unclear to me what the differences are.
- `-O2` or `-O3`. It seems pretty clear to me that `-O3` is faster, but I haven't experimented with it.
- `-march=native`. From my brief tests, this allows for more vectorization, so we should have faster code! Specifically, it switched the vectorization from 16 to 32 bytes. This one seems pretty straightforward.
- `-ffast-math`. It is not always clear if this tag helps or hurts from David's advice.
- `restrict` keyword. Adding this keyword switches us from slow C to Fortran speed, so I think we should include it.
- Block sizes M_B, N_B, and K_B.
- Whether or not to use alignment. I have not been able to get a speed up out of alignment yet, but I think I have a sense of how it works now! If we do have alignment, we have more decisions:
  - `alignas` or `posix_memalign`? We have good working examples from David using the `alignas` framework with in `kernel2.c`.

  - Size of aligned chunks of memory. Is it faster to have 32 or 64 byte chunks?
  - Should we align C as well?
  - If $M_B \neq N_B$, should A and B be aligned differently?

- `pragma omp simd reduction(+:x)`. Honestly, I am not exactly sure what is up with this command, but it made things faster in `centroid.c`. I don't see why the same wouldn't be true for us!
- Loop order. It is generally not easy to figure out how loops should be ordered.
- Row major or column major for A and B. That is, do we choose the indexing of `A[i*KB + k]` or `A[k*MB + i]`? We have the opportunity to play with this if we are copying over memory. I am assuming we have fixed the order of `C` in this example.

I'm sure that I have missed something in this list, but I think that is a good start. As for testing it, I think we should test a bunch of parameters for these things. The performance should be measured in GFlops/second, as the speeds for the full `dgemm` is. For the purposes of the block test, I don't think that we should include the copy time for alignment. Memory copying is is an $O(n^2)$ operation, which shouldn't make a big difference on the larger scale, but it might be bad for the performance at smaller scales.

Here is a list of the things I think are reasonable to test:

- Block sizes - maybe choose from 4, 8, 12, and 16. There are $4^3 = 64$ options. **64 options**
- Alignment size 8, 32, and 64, where 8 is unaligned. Assuming A, B, and C are all aligned the same, there are **3 options**
- Loop order. **6 options**
- Row/column major for A and B. **4 options**
- `-ffast-math`. **2 options**

Multiplying this all out, there are **9216 options**. If we take a second per test, this will take 2.56 hours.

Hypothesis: I think the best looping will be over 16 by 16, aligned in 32 byte chunks, with the loop order `j(k(i()))`.

3 square_dgemm

In this part, memory management will be our problem. Here are some things that we have to decide

- Block ordering. There are so many options here. I don't know the right way to think about it.
- Copy optimization. When and how to copy the memory from the input A and B to the aligned packages isn't entirely obvious.
- Buffering. Are we okay with the buffering from tiling the matrix due to memory alignment?
- etc.