Project 1 Report

MM Basic

One of the first hypotheses I had was that a bigger cache will result in lower AAT because, obviously, accessing cache is much faster than accessing memory. This hypothesis seemed to be verified with the various `C` values I tried. Thus, I set `C` to the max value possible while still fulfilling the constraints of not exceeding 48 KiB. Therefore, `C = 15`.

From there, I experimented with various `B` and `S` values to get a general sense of how AAT changes as those values change. I first kept `S` statically set to `0` to find the optimal block size for each program. Here are some results for `mm_basic`:

With `C = 15`, we try

- `B = 4` : AAT=3.86618402
- `B = 5` : AAT=3.85600573
- `B = 6` : AAT=3.85099778
- `B = 7` : AAT=3.84929528
- `B = 8` : AAT=3.85023644
- `B = 9` : AAT=3.85351368
- `B = 10` : AAT=3.83925552
- `B = 11` : AAT=3.37367568
- `B = 12` : AAT=2.68686259
- `B = 13` : AAT=2.35283862
- `B = 14` : AAT=2.18124271
- `B = 15` : AAT=2.09545129

At this point, I concluded that `2^7` bytes per block was the inflection point for optimal size. However, upon further experimentation, I noticed that `2^15` bytes had a lower AAT. Even though the L1 miss rate was going up after `B=7`, the victim cache miss rate was going down by almost an order of magnitude each time. This makes sense since the matrix multiplication uses the same vector of numbers repeatedly, so as long as the block can store the entire vector, keeping it in l1 or victim cache is advantageous.

Then I started experimenting with various set sizes and ultimately determined that a direct mapped cache is the best configuration for cache. This is probably because the programmers of the matrix multiplication optimized their memory usage, so giving them as many index bits as possible to control memory placement is advantageous. Furthermore, matrix multiplication has poor temporal locality, so increasing index bits also reduces evictions.

Finally, the victim cache is of course helpful because it essentially extends the cache and hads no overhead since it is checked in parallel with the l1 cache.

Thus, the optimal configuration for the cache was: `C=15, B=15, S=0`.

MM Tiled

Similar to the basic version, I found `C=15` to be the optimal size of the cache since a bigger cache allows for fewer evictions.

I again experimented with various B values and found:

- `B = 4` : AAT=2.26175522
- `B = 5` : AAT=2.25577504
- `B = 6` : AAT=2.18713142
- `B = 7` : AAT=2.15347469
- `B = 8` : AAT=2.14067270
- `B = 9` : AAT=2.13748365
- `B = 10` : AAT=2.14533345
- `B = 11` : AAT=2.10289608
- `B = 12` : AAT=2.04227413
- `B = 13` : AAT=2.02881359
- `B = 14` : AAT=2.02436909
- `B = 15` : AAT=2.12851991

Again, like the basic version, `2^9` bytes seemed to be the inflection point for block size, but because of the nature of matrix multiplication, the victim cache miss rate reduced dramatially with larger block sizes. Thus, the optimal block size ended up being `2^14` bytes.

Also, as noted with basic, having index bits be as large as possible is advantageous because the program is probably optimized for memory management and because matrix multiplication has poor temporal locality. However, the speed gained from victim cache miss rate decreasing overrode the speed gained from a large index size.

Thus, the optimal configuration for the cache was: `C=15, B=14, S=0`.