Matrix Multiplication C/Assembly Code Evaluation

James Qian

[jnqian@sfu.ca](mailto:jnqian@sfu.ca)

**Objective**

To compose and evaluate a few pieces of code that do matrix multiplication C = A x B. I am interested in matrix multiplication because it is one of the core operations in ANN (artificial neural network) and Deep Learning. Specifically, I am interested in multiplication of matrix of larger size (at least over one thousand in width and height) since smaller sized ones can usually be done in milliseconds and does not really need much attention.

**Code Description**

I will use g++ -Ivectorclass -Wall -Wpedantic -std=c++17 -march=haswell -funsafe-math-optimizations -O3 matmul2.c matmul.S as my compilation command. The test platform is CSIL desktop with Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz running Linux. The -funsafe-math-optimizations option allows the compiler freedom to arrange the math operations in arbitrary order within small errors to enable the usage of accelerations such as SIMD instructions.

The \_\_restrict key word for the type of the matrix is used to indicate that there is no overlap in memory address for all the matrix to allow the compiler to compile the code more efficiently in order to improve performance.

1. Copy and use C code from Rosetta which does double typed matrix multiplication, I call this function mat\_mult. Also adapt the same function to float data type matrix, called mat\_mult\_f. All the other codes are based on this basic algorithm.

2. To improve data locality, I would do the transpose of matrix B first, then use B\_t (transpose of B) to get A x B. It is because during the process of multiplication, the elements of matrix A are fetched in a by-row fashion while the elements of matrix B are fetched in a by-column fashion. This by-column fashion data access greatly decreases data locality, especially when the size of the matrix is large and all the elements can not be fit into cache. So, pre-calculating the transpose of matrix B allows the by-row access of matrix B and should theoretically drastically decrease cache misses and increase performance for multiplication of matrix of larger sizes. I would call the respective implementation mat\_mult2 (for double data type) and mat\_mult2\_f (for float data type).

3. I would use Agner Fog's vectorclass library to vectorize the inner most loop of matrix multiplication into SIMD instructions. I call the respective function mat\_mult3 (for double data type) and mat\_mult3\_f (for float data type). This is based on the usage of transpose of matrix B method mentioned in part 2.

4. I would use a manual unfolding of the inner-most loop by adding an additional array which accumulates the total for each of the unfolding. No vectorclass is used but it still is based on the transpose of matrix B method. The respective functions are called mat\_mult4 (for double data type) and mat\_mult4\_f (for float data type).

5. A combination of method 4 and 5 are used and the respective functions are called mat\_mult5 (for double data type) and mat\_mult5\_f (for float data type).

6. Assembly code is written using SIMD double and float instruction with 256 bits ymm registers. The functions are called matmul\_d (for double data type) and matmul\_f (for float data type)

**Data & Analysis**

Running speeds are measured with clock\_gettime() within the c code. Also I used perf to collect branch-misses, branches, L1-dcache-load-misses, L1-dcache-loads, LLC-load-misses, LLC-loads data to evaluate the performance of the functions as shown in the tables below:



Table 1. Running speed (ms) for different types of implementations for matrix multiplication.



Table 2. Output from perf utility for branch-misses, branches, L1-dcache-load-misses, L1-dcache-loads, LLC-load-misses, LLC-loads parameter for smaller (M=1024, N=1024, P=1024) versus larger (M=7168, N=7168, P=7168) matrix multiplication.

From Table 1. I can see that the original c code from Rosetta (mat\_mul & mat\_mul\_f) is the slowest, while the c implementation (mat\_mul2 & mat\_mul2\_f) with the transposed matrix B without any other explicit vector or SIMD instructions are the fastest. All the other implementations are still much faster than the original Rosetta implementation. For smaller (M,N,P <= 2560) matrix, the speed up over the Rosetta code is around 1.5-3 fold. For larger (M,N,P>=5120) matrix, the speed up over the Rosetta code is around 6-7 fold. I also check the running speed for M,N,P = 7168 for mat\_mul2 without the -funsafe-math-optimizations option and the running time is 355977.91 (ms) which is ~2.4x slower than the running time with that option. I suspect that less cache loading miss due to better data locality is the main reason for the speed up over Rosetta code, so I used the perf tool to confirm that.

From Table 2. I can see that there are ~2.5-3x less cache miss for the mat\_mul2 and mat\_mul2\_f versus their respective Rosetta implementations. Also there are far less LLC miss for mat\_mul2 and mat\_mul2\_f versus their respective Rosetta implementations when the matrix sizes are large (M,N,P = 7168). This confirms the hypothesis that less cache miss is the reason for the overall faster computation. Also, I can see that there are more branch mispredictions for mat\_mul2 and mat\_mul2\_f over the Rosetta codes. However, this effect is countered by the effect due to less cache miss, so the overall speed is still much faster for mat\_mul2 and mat\_mul2\_f than the Rosetta code.

**Conclusion**

By testing all the implementations, I can see that the fastest implementation is the idiomatic C code with consideration for improved data locality and the g++ -funsafe-math-optimizations options as well as the \_\_restrict keyword to enable SIMD parallelism. This suggests that I should write clean readable C code as much as possible and leave the difficult assembly code writing to the compiler. Also, it makes sense to think about leaky abstractions because data locality (and possibly branch misprediction and other hardware related issues) does make a huge difference on performance due to hardware setup including the limited size of different levels of storage devices such as registers, cache and memory.

**Reference**

1. *SFU CMPT295 Lecture Slides*, Greg Baker

2. *Agner Fog's vectorclass library*, <https://github.com/vectorclass/version2>

3. *Rosetta C code for matrix multiplication*, <https://rosettacode.org/wiki/Matrix_multiplication#C>