Matrix Multiplication C/Assembly Code Evaluation

James Qian

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:

	Matrix C = A x B, where C (MxP) A(MxN) B(NxP)				
M	1024	2560	5120	7168	9216
N	1024	2560	5120	7168	9216
P	1024	2560	5120	7168	9216
	function runing time (ms)				
mat_mult_d	352.23	7833.50	65337.36	175609.89	379676.92
mat_mult5	325.76	8188.98	69604.27	187521.46	404964.70
mat_mult4	362.20	8464.07	69082.46	184975.34	402535.09
mat_mult3	303.93	7452.36	62669.37	169365.90	364724.27
mat_mult2	250.60	6167.44	54476.47	144922.92	314897.56
mat_mult	531.00	22315.28	310896.34	957007.58	2021554.81
matmul_f	177.48	4144.81	35635.76	96761.41	204005.95
mat_mult5_f	159.46	4208.69	37629.49	103011.44	218042.54
mat_mult4_f	292.42	6072.98	51805.98	141040.96	298925.25
mat_mult3_f	160.86	3966.17	34256.88	92973.29	197050.12
mat_mult2_f	105.27	2849.08	27182.71	72621.25	165021.28
mat_mult_f	166.22	8544.33	140176.49	444282.51	964945.18

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

	Matrix C = A x B, where C (MxP) A(MxN) B(NxP)							
M	1024	1024	1024	1024	7168	7168	7168	7168
N	1024	1024	1024	1024	7168	7168	7168	7168
P	1024	1024	1024	1024	7168	7168	7168	7168
function	mat_mul	mul_mul2	mat_mul_f	mul_mul2_f	mat_mul	mul_mul2	mat_mul_f	mul_mul2_f
	perf results							
L1-dcache-load-misses	425,760,943	136,969,840	208,679,571	67,711,760	247,964,230,365	92,495,966,757	116,176,292,474	46,331,114,853
L1-dcache-miss-percentage	72.29%	23.22%	63.60%	22.13%	132.62%	49.52%	122.85%	49.03%
L1-dcache-loads	588,975,781	589,891,081	328,122,330	305,995,524	186,980,454,733	186,781,131,226	94,570,800,758	94,496,871,835
LLC-load-misses	25,933,483	2,511,783	31,977	32,942	131,557,886,126	13,935,053,719	62,100,411,984	6,775,425,357
LLC-load-miss-percentage	5.99%	8.26%	0.01%	0.18%	69.06%	46.05%	73.76%	47.93%
LLC-loads	432,971,741	30,395,983	218,679,504	17,870,745	190,499,887,299	30,262,967,339	84,189,430,947	14,135,362,419
branch-misses	375,269	1,164,686	220,768	1,203,459	28,818,436	57,478,832	12,088,547	55,878,643
branch-miss-percentage	0.12%	0.39%	0.14%	0.69%	0.03%	0.06%	0.03%	0.12%
branches	301,716,738	302,081,577	161,560,882	175,049,066	95,044,482,247	94,128,724,399	47,976,509,771	47,963,197,626

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