Assignment 2

David Warren
University of Alabama
dlwarren2@crimson.ua.edu
CWID: 11703777

I. Introduction

This report reflects the observations and analyses of *Matrix Multiplication Triple Loop Algorithms* via three methods: Single Register Reuse, Cache blocking, a combined Cache Blocking and Register Blocking method. We start off with a general performance analysis of the triple loops on small and large matrices. Then analyze blocking algorithms on matrices of the same sizes of the previous part. In the third part, we go over the performance of the implementations of each algorithm. Finally, we combine register blocking from Assignment 1 and the cache blocking from this current assignment to utilize both sequential optimization techniques and analyze its performance on two compiler versions: gcc 7.3.0 and gcc 5.4.0. We then compare each optimization flag effects between versions.

II. Ease of Use

A. CPU and Compiler Information

CPU used was Intel Xeon Gold 4180 CPUs on normal nodes to gather performance metrics. Compiler used was gcc (GCC) 7.3.0. gcc 5.4.0 was also used in Part IV. Optimization flag was set to default -O0 least optimized for parts I-III. Part IV utilized all optimization flags to compare performance.

B. Using the Makefile

- "Make" to compile *srrmm.c, bmm.c , b_rb_combo.c* and other matrix files.
- "Make clean" to remove binary and executable files
- "Make submit" to submit srrmm.job, bmm.job and combo.job

III. Part I Observations

A. 10x10 Matrix Multiplication with Single Register Reuse

We will first analyze the cache miss rate with single register reuse for 10x10 matrices for the following *simple triple-loop* algorithms: *ijk, ikj, jik, jki, kij, kji.*

1) ijk and jik

For *ijk* 10x10, after first j loop iteration, k loop finishes and all of matrix B will have been read into cache along with first row of matrix A [11 lines of cache used]. We go down the column and miss each of the first elements. After the miss, the following 10 elements are read into the cache and we finish reading B. We go through the rest of the j loop only missing the first element of each row in matrix A (excluding the first we already read) [9 more lines], as B is already read into cache. We read C matrix in during the j loop when loading C[i][j] into sum so it will look just like A [1 + 9 more lines = 10 lines] for a total of 30 misses. For *jik*, it is similar to ijk for the inner loop. But for reading in matrix C, we will be reading in rows instead going through the i iterations. As we are accessing elements in C by advancing rows, we will access similar to B instead of A. total misses still equal to 30. An equation to model the number of misses for each element in each matrix for both of these versions can be seen in the following equation (1).

$$\text{Miss Rate} = \begin{cases} A[i][0]\ 0 \le i \le 9 \\ B[k][0] 0 \le k \le 9 \\ C[i][0] 0 \le i \le 9 \end{cases} \quad (1)$$

Generalizing this equation gives us the following equation (2).

$$\text{Miss Rate} = \frac{n^2 + n^2 + n^2}{10(2n^3)} \quad (2)$$

When n=10, then we can calculate Miss Rate = 30/200 = 0.15 or 15%.

2) kij and ikj

Here we are accessing A first and it is fixed. It will read like C in the *ijk, jik* loops. In the inner most loops we are accessing B and C by rows so we will miss every first element of each row and miss the next nine. Therefore, the Miss rate equation is the same as equation (1) and the generalization is the same as equation (2). Resulting again in a Miss Rate = 0.15 or 15%.

3) jki and kji

Now we are accessing C in the k loop in place so access of matrix B will look like C was in ijk. We are reading both A and C column wise so there will be 20 lines read again. B is fixed and will be read in after middle loop [10 lines again] for a total of 30 lines. The order of misses differs, but as total number of doubles are smaller than our overall cache size, the Miss Rate

equation will be the same as equation (1) and the generalization of the Miss rate will be the same as equation (2) with Miss Rate = 0.15 or 15%.

4) Conclusion

In conclusion, all miss rates are the same by equation (1) and equation (2) for 10x10 matrices for each element in each matrix A, B, C. See Figure (1) below for summary of calculations.

	ijk	jik	kij	ikj	jki	kji
Miss Rate	0.15	0.15	0.15	0.15	0.15	0.15

Figure 1. Miss Rate for 10x10

B. *10000x10000 Matrix Multiplication with Single Register Reuse*

1) ijk and jik

For *ijk,* C is fixed and has 0 misses per inner loop iteration. For inner loop accessing A[i][k], we are moving down column of A, so on each row we visit column 0 we will miss the first time and hit the next 9, and the continue the same way. This means A[i][k] = 10000 when k%10==0 with miss rate of 0.25. for B[k][j], we are moving down rows in the inner loop and read in the first 10 columns. That means we have B[k][j] = 10000 for a miss rate of 1.0. For *jik*, same as above for A[i][k] and B[k][j], C[i][j] in inner loop. This gives us an equation we can use for Miss Rate will be modeled in equation (3).

$$\text{Miss Rate} \begin{cases} A[i][k] = 10000, \ k\%10 = 0 \\ B[k][j] = 10000 \end{cases} \quad (3)$$

We can therefore calculate our Miss Rate using the following generalized equation (4).

$$\text{Miss Rate} = \frac{n^3/10 + n^3}{(2n^3)} \quad (4)$$

Using equation (4), with n=10000 we get the calculated Miss Rate = 0.55 or 55%.

2) kij and ikj

A is fixed and has 0 misses per inner loop iteration. B[k][j] being read row-wise so cold miss on every row first column and hit on 9 following elements. C[i][j] is also being row-wise so misses are similar to B. This gives us an equation to model Miss Rate as seen in equation (5).

$$\text{Miss Rate} \begin{cases} B[k][j] = 10000, \ j\%10 = 0 \\ C[i][j] = 10000, \ j\%10 = 0 \end{cases} \quad (5)$$

We can therefore calculate our Miss Rate using the following generalized equation (6).

$$\text{Miss Rate} = \frac{n^3/10 + n^3/10}{(2n^3)} \quad (6)$$

Using equation (6) with n=10000, we get the calculated Miss Rate = 0.1 or 10%.

3) jki and kji

B is fixed and has 0 misses per inner loop iteration. A and C are both read column-wise so each miss on every row visited giving us a miss rate per iteration of 1.0. We can model the Miss Rate of *jki* and *kji* in the following equation (7).

$$\text{Miss Rate} \begin{cases} A[i][k] = 10000 \\ C[i][j] = 10000 \end{cases} \quad (7)$$

This gives us the following generalization labeled equation (8).

$$\text{Miss Rate} = \frac{n^3 + n^3}{(2n^3)} \quad (8)$$

Calculation given any size n gives us a Miss Rate = 1.0 or 100% in the inner loop. It may be noted that B has 0 misses per inner loop but as n→ ∞, the Miss Rate will approach 1.0 or 100%.

4) Conclusion

In conclusion, we have three separate groupings for miss rate depending on the ordering of which matrices are read in certain ways. See the following Figure (2) for a summary of the Miss Rates.

	ijk	jik	kij	ikj	jki	kji
Miss Rate	0.55	0.55	0.1	0.1	1	1

Figure 2. Miss Rate for 10000x10000

IV. Part II Observations

A. *Blocking Matrix Multiplication*

Using blocking with matrix multiplication can help drastically with many versions of the algorithm. For analyzing block use in matrix multiplication, we just need to slightly modify general algorithms for cache miss from Part I. First, subdivide 10000x10000 by 10x10 submatrices 1000x1000 new matrices. Following the procedure from the Sequential Optimization lecture, we can reach a general formula of the following in equation (9c).

$$\text{First Block Iteration} = \frac{2n}{B} \times \frac{B^2}{10} = \frac{nB}{5} \quad (9a)$$

$$\text{Second Block Iteration} = \frac{2n}{B} \times \frac{B^2}{10} = \frac{nB}{5} \quad (9b)$$

$$\text{Miss Rate} = \frac{nB}{5} \times \left(\frac{nB}{5}\right)^2 = \frac{n^3}{5B} \quad (9c)$$

When n=10000, B=10 equation (9c) gives us Miss Rate = 0.01 or 1%. This equation can also be modeled in the same way as Part I, which we will do now.

B. *Modification of previous Part I equations*

1) *ijk and jik*

$$\text{Miss Rate} \begin{cases} A[i][k] = 1000, k\%10 = 0 \\ B[k][j] = 1000, j\%10 = 0 \end{cases} \quad (3b)$$

$$\text{Miss Rate} = \frac{n^3/10B + n^3/10B}{(2n^3)} \quad (4b)$$

This gives us a new Miss Rate when n=10000, B=10 of 0.01 or 1%.

2) *ij and ikj*

$$\text{Miss Rate} \begin{cases} B[k][j] = 1000, k\%10 = 0 \\ C[i][j] = 1000, j\%10 = 0 \end{cases} \quad (5b)$$

$$\text{Miss Rate} = \frac{n^3/10B + n^3/10B}{(2n^3)} \quad (6b)$$

This gives us a new Miss Rate when n=10000, B=10 of 0.01 or 1%.

3) *jki and kji*

$$\text{Miss Rate} \begin{cases} A[k][j] = 1000, k\%10 = 0 \\ C[i][j] = 1000, j\%10 = 0 \end{cases} \quad (7b)$$

$$\text{Miss Rate} = \frac{n^3/10B + n^3/10B}{(2n^3)} \quad (8b)$$

This gives us a new Miss Rate when n=10000, B=10 of 0.01 or 1%.

C. *Conclusion*

As you can see, we have shown Part II-A's method to indeed be correct, as all reductions of Part I have been modified to fit 10x10 blocks, with each Miss Rate still being 0.01 or 1%.

V: Part III

A. *Simple Register Reuse Matrix Multiplication Implementation*

The simple register reuse matrix multiplication (srrmm) natural does not use blocking, so each of the 6 versions of the algorithms GFLOPS were calculated below in Figure (3). Per the image, *ikj* had the best performance only slightly outperforming kij.

	ijk	jik	ikj	kij	jki	kji
GFLOPS	0.774847	0.614022	1.379492	1.370672	0.271218	0.23454

Figure (3) srrmm GFLOPS performance

B. *Blocking Matrix Multiplication Implementation*

Blocks used are set {16, 32, 64, 128, 256, 512} over the 6 versions of the algorithm. Performance was again measured in GFLOPS seen below in Figure (4) for blocking matrix multiplication (bmm).

	ijk	jik	ikj	kij	jki	kji
16	1.403387	1.351818	1.321562	1.326711	0.512471	0.630856
32	1.484794	1.357386	1.379682	1.363916	0.498399	0.506869
64	1.496316	1.402321	1.406499	1.382087	0.497388	0.501146
128	0.782911	0.774912	1.339661	1.275623	0.330302	0.475397
256	0.796609	0.753738	1.394318	1.346473	0.457817	0.542614
512	0.801143	0.782286	1.41336	1.408996	0.402167	0.533043

Figure (4) bmm GFLOPS Performance

For 16, 32, 64, 128, 256, and 512 the respective best performing algorithms as per the chart were measured to be: *ijk, ijk, ijk, ikj, ikj, ikj.*

VI: Part IV

A. *Combination of Cache Blocking and Register Blocking*

In Part IV, the six triple loop algorithms for matrix multiplication were implemented using cache blocking alongside register blocking from Assignment 1. Below are shown the results tables for each optimization tag, as well as compiler version. We will then conclude this section by giving a short analysis of each optimization flag and compiler version to highlight the highest performing algorithm.

1) *O0 optimization (least)*

a) *gcc 7.3.0* ***Figure (5)***

	ijk	jik	ikj	kij	jki	kji
16	3.591561	3.419653	3.749825	3.751255	2.695864	2.859195
32	3.778166	3.664244	3.861342	3.802993	2.629922	2.803311
64	3.771746	3.656441	3.92213	3.509196	2.443805	2.586338
128	2.391988	2.458115	3.857597	3.54987	1.718247	2.015485
256	2.514728	2.479785	3.904554	3.799822	2.006773	1.971327
512	2.502276	2.313585	3.890833	3.881294	1.79078	1.868282

b) *gcc 5.4.0* ***Figure (6)***

	ijk	jik	ikj	kij	jki	kji
16	3.638075	3.43091	3.74063	3.633908	2.529033	2.808342
32	3.776615	3.659051	3.847738	3.797807	2.450996	2.58391
64	3.772893	3.673265	3.885556	3.53304	2.380775	2.441094
128	2.202185	2.255468	3.805668	3.359298	1.614144	1.823875
256	2.330225	2.296942	3.823628	3.745013	1.826948	2.031577
512	2.472745	2.400545	3.92412	3.878991	1.619728	2.022184

2) *O1 optimization*

a) *gcc 7.3.0* ***Figure (7)***

	ijk	jik	ikj	kij	jki	kji
16	9.372604	7.939907	16.245967	17.1658	3.338697	3.897384
32	10.66317	9.607099	19.585978	17.92987	3.484414	3.709926
64	10.42273	9.682036	21.062944	14.97272	3.596869	3.686763
128	4.610153	4.292503	13.911613	13.5205	2.374242	2.548507
256	4.597293	4.344388	17.329444	15.0974	2.553714	2.540338
512	4.603608	4.359086	20.10518	18.30724	2.593296	2.65965

b) *gcc 5.4.0* ***Figure (8)***

	ijk	jik	ikj	kij	jki	kji
16	8.564064	7.46447	9.742984	9.530384	3.380115	3.802261
32	10.00583	9.054451	10.09151	9.558745	3.506705	3.683163
64	9.942103	9.090273	10.21311	7.872676	3.541134	3.674461
128	4.725924	4.494074	9.541397	7.638709	2.407791	2.567498
256	4.699166	4.517026	9.951332	9.047458	2.586521	2.616898
512	4.710188	4.366039	10.19857	9.822696	2.500498	2.661552

3) *O2 optimization*

a) *gcc 7.3.0* ***Figure (9)***

	ijk	jik	ikj	kij	jki	kji
16	9.345595	7.934035	16.25181	17.17647	3.356044	3.889526
32	10.65299	9.623081	19.64663	18.96443	3.493586	3.715063
64	10.54198	9.439612	20.91606	14.80773	3.589542	3.666432
128	4.503255	4.286747	13.79227	13.25134	2.419705	2.514823
256	4.574409	4.298871	17.07442	14.91817	2.544286	2.544363
512	4.589657	4.348766	19.56183	18.2149	2.548916	2.562751

b) *gcc 5.4.0* ***Figure (10)***

	ijk	jik	ikj	kij	jki	kji
16	9.280832	7.847772	11.8099	12.05753	3.395967	3.828474
32	10.7017	9.60636	12.95562	12.34939	3.507708	3.698911
64	10.56019	9.598815	13.22054	9.838723	3.606405	3.680877
128	4.702569	4.407431	11.49548	9.228348	2.471343	2.576284
256	4.751572	4.477769	12.80804	11.57592	2.585124	2.574441
512	4.74155	4.485505	13.30169	12.71032	2.58707	2.474358

4) *O3 optimization (most)*

a) *gcc 7.3.0* ***Figure (11)***

	ijk	jik	ikj	jki	jki2	kji
16	9.407716	7.963557	16.26048	17.21402	3.398287	3.820176
32	10.71349	9.66653	19.63804	18.89317	3.507177	3.694673
64	10.57296	9.653871	21.09311	15.01516	3.609118	3.676628
128	4.72191	4.453589	14.14638	13.56811	2.534001	2.623481
256	4.762548	4.685156	17.01621	14.90436	2.721806	2.723718
512	4.963873	4.738492	20.0081	18.32858	2.690079	2.670804

b) *gcc 5.4.0* ***Figure (12)***

	ijk	jik	ikj	kij	jki	kji
16	9.039135	7.601066	11.84572	12.04369	3.306945	3.845128
32	10.55653	9.463478	12.82002	12.39054	3.479949	3.702924
64	10.41837	9.757161	13.21663	9.776028	3.58066	3.681629
128	4.636235	4.34481	11.44788	9.46775	2.624369	2.631362
256	4.693574	4.474541	12.74949	11.4457	2.626075	2.654346
512	4.779394	4.555084	13.27006	12.76509	2.655343	2.639042

B. *Combination of Cache Blocking and Register Blocking Short Analysis*

1) *O0 optimization*

For the O0 optimized algorithms, gcc 7.3.0 and gcc 5.4.0 had similar performance. The best performing for each was *ikj* at 64B with 7.3.0 performing slightly ahead of 5.4.0

2) *O1 optimization*

For the O1 optimized algorithms, 7.3.0 stayed similar for 16B and 32B, then greatly increased in performance increase of block size after that. The best performing for each compiler version was *ikj* at 64B with 7.3.0 vastly outperforming 5.4.0 in *ikj* and *kij* after 32B size. There was another spike in performance at 512B.

3) *O2 optimization*

For the O2 optimized algorithms, there is a pattern similar to the O1 optimized algorithms, without any improvement for 7.3.0, but a decent improvement for 5.4.0 in the larger block sizes.

4) *O3 optimization*

For the O3 optimized algorithms, we see our peak performance in the 7.3.0 *ikj* algorithm at 64B at around 21 GFLOPS, our highest out of any algorithm. This is notedly only slightly higher than the previous O2 7.3.0 version, but still is the highest recorded performance. After O2, no significant improvements are made for 7.3.0 algorithms or 5.4.0 algorithms.

VI: Source Code And Results

Source code is attached in tar file. Input and output matrices are available in the tar file in proj2/testFiles and proj2/output/[algorithm] respectively. You may also find the error logs on proj1/output/[algorithm]. Sample output from each is included as well.

VII: Appendix

Figure (1) Miss Rate for 10x10

	ijk	jik	kij	ikj	jki	kji
Miss Rate	0.15	0.15	0.15	0.15	0.15	0.15

Figure (2) Miss Rate for 10000x10000

	ijk	jik	kij	ikj	jki	kji
Miss Rate	0.55	0.55	0.1	0.1	1	1

Figure (3) srrmm GFLOPS performance

	ijk	jik	ikj	kij	jki	kji
GFLOPS	0.774847	0.614022	1.379492	1.370672	0.271218	0.23454

Figure (4) bmm GFLOPS Performance

	ijk	jik	ikj	kij	jki	kji
16	1.403387	1.351818	1.321562	1.326711	0.512471	0.630856
32	1.484794	1.357386	1.379682	1.363916	0.498399	0.506869
64	1.496316	1.402321	1.406499	1.382087	0.497388	0.501146
128	0.782911	0.774912	1.339661	1.275623	0.330302	0.475397
256	0.796609	0.753738	1.394318	1.346473	0.457817	0.542614
512	0.801143	0.782286	1.41336	1.408996	0.402167	0.533043

Figure (5) O0 optimization (least) 7.3.0

	ijk	jik	ikj	kij	jki	kji
16	3.591561	3.419653	3.749825	3.751255	2.695864	2.859195
32	3.778166	3.664244	3.861342	3.802993	2.629922	2.803311
64	3.771746	3.656441	3.92213	3.509196	2.443805	2.586338
128	2.391988	2.458115	3.857597	3.54987	1.718247	2.015485
256	2.514728	2.479785	3.904554	3.799822	2.006773	1.971327
512	2.502276	2.313585	3.890833	3.881294	1.79078	1.868282

Figure (6) O0 optimization (least) 5.4.0

	ijk	jik	ikj	kij	jki	kji
16	3.638075	3.43091	3.74063	3.633908	2.529033	2.808342
32	3.776615	3.659051	3.847738	3.797807	2.450996	2.58391
64	3.772893	3.673265	3.885556	3.53304	2.380775	2.441094
128	2.202185	2.255468	3.805668	3.359298	1.614144	1.823875
256	2.330225	2.296942	3.823628	3.745013	1.826948	2.031577
512	2.472745	2.400545	3.92412	3.878991	1.619728	2.022184

Figure (7) O1 optimization 7.3.0

	ijk	jik	ikj	kij	jki	kji
16	9.372604	7.939907	16.245967	17.1658	3.338697	3.897384
32	10.66317	9.607099	19.585978	17.92987	3.484414	3.709926
64	10.42273	9.682036	21.062944	14.97272	3.596869	3.686763
128	4.610153	4.292503	13.911613	13.5205	2.374242	2.548507
256	4.597293	4.344388	17.329444	15.0974	2.553714	2.540338
512	4.603608	4.359086	20.10518	18.30724	2.593296	2.65965

Figure (8) O1 optimization 5.4.0

	ijk	jik	ikj	kij	jki	kji
16	8.564064	7.46447	9.742984	9.530384	3.380115	3.802261
32	10.00583	9.054451	10.09151	9.558745	3.506705	3.683163
64	9.942103	9.090273	10.21311	7.872676	3.541134	3.674461
128	4.725924	4.494074	9.541397	7.638709	2.407791	2.567498
256	4.699166	4.517026	9.951332	9.047458	2.586521	2.616898
512	4.710188	4.366039	10.19857	9.822696	2.500498	2.661552

Figure (9) O2 optimization 7.3.0

	ijk	jik	ikj	kij	jki	kji
16	9.345595	7.934035	16.25181	17.17647	3.356044	3.889526
32	10.65299	9.623081	19.64663	18.96443	3.493586	3.715063
64	10.54198	9.439612	20.91606	14.80773	3.589542	3.666432
128	4.503255	4.286747	13.79227	13.25134	2.419705	2.514823
256	4.574409	4.298871	17.07442	14.91817	2.544286	2.544363
512	4.589657	4.348766	19.56183	18.2149	2.548916	2.562751

Figure (10) O2 optimization 5.4.0

	ijk	jik	ikj	kij	jki	kji
16	9.280832	7.847772	11.8099	12.05753	3.395967	3.828474
32	10.7017	9.60636	12.95562	12.34939	3.507708	3.698911
64	10.56019	9.598815	13.22054	9.838723	3.606405	3.680877
128	4.702569	4.407431	11.49548	9.228348	2.471343	2.576284
256	4.751572	4.477769	12.80804	11.57592	2.585124	2.574441
512	4.74155	4.485505	13.30169	12.71032	2.58707	2.474358

Figure (11) O3 optimization (most) 7.3.0

	ijk	jik	ikj	jki	jki2	kji
16	9.407716	7.963557	16.26048	17.21402	3.398287	3.820176
32	10.71349	9.66653	19.63804	18.89317	3.507177	3.694673
64	10.57296	9.653871	21.09311	15.01516	3.609118	3.676628
128	4.72191	4.453589	14.14638	13.56811	2.534001	2.623481
256	4.762548	4.685156	17.01621	14.90436	2.721806	2.723718
512	4.963873	4.738492	20.0081	18.32858	2.690079	2.670804

Figure (12) O3 optimization (most) 5.4.0

	ijk	jik	ikj	kij	jki	kji
16	9.039135	7.601066	11.84572	12.04369	3.306945	3.845128
32	10.55653	9.463478	12.82002	12.39054	3.479949	3.702924
64	10.41837	9.757161	13.21663	9.776028	3.58066	3.681629
128	4.636235	4.34481	11.44788	9.46775	2.624369	2.631362
256	4.693574	4.474541	12.74949	11.4457	2.626075	2.654346
512	4.779394	4.555084	13.27006	12.76509	2.655343	2.639042