CSE 611 Homework #4: Impact of Loop-Order on Iterative Matrix Multiplication Latency

Report and Programs by Kaleb Byrum

10/1/2020

Introduction

This report will explore the impact of loop-order on iterative operations, such as matrix multiplication. Matrix multiplication for column-majored arrays can be described as:

```
for (int i = 0; i < n; i++)
   for (int j = 0; j < n; j++)
      for (int k = 0; k < n; k++)
         C[i+j*n] += A[i+k*n] * B[k+j*n];
```

Figure 1: Example Matrix Multiplication Code

…where variables i, j, and k are placed in I-J-K loop order. This report will explore the six different ways this snippet of code can be arranged, and its subsequent impact on program latency and performance. Software will be developed that implements this snippet in various loop orders, and results will be analyzed from these program's performance.

Implemented Programs

The following program is one of six implemented programs to determine the impact of loop order:

```c
/*

CSE 611 HW4 Matrix Multiplication Latency Experiment
Configured for IJK Loop Order.

By Kaleb Byrum
9/26/20

*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

struct Matrix //Structure used to generate matrices
{
    int width;
    int height;
    int* ptr;
};

static struct Matrix matrix_create(int width, int height)
{
    struct Matrix new_matrix;
    new_matrix.width = width;
    new_matrix.height = height;
    new_matrix.ptr = malloc(width * height * sizeof(int));
    return new_matrix;
}

static void matrix_destroy(struct Matrix* m)
{
    free(m->ptr);
}

int main()
{
    int r = 0;
    int c = 0;

    //The first step is to create the two matrices that will be multiplied.
    int n = 5000; //5000x5000 matrices will be created.

    struct Matrix A = matrix_create(n, n); //Creating the first 5000x5000 matri
x
    struct Matrix B = matrix_create(n, n); //Creating the second 5000x5000 matr
ix

    struct Matrix C = matrix_create(n, n); //This will contain the product 5000
x5000 matrix.

    printf("Generating Matrix A...\n"); //Generates Matrix A full of example da
ta. In this case, its itr % n.
    for (long itr = 0; itr < A.height * A.width; itr++)
    {
        A.ptr[itr] = itr % n;
    }

    printf("Generating Matrix B...\n"); //Generates Matrix B full of example da
ta. In this case, its itr % n.
    for (long itr = 0; itr < B.height * B.width; itr++)
    {
        B.ptr[itr] = itr % n;
    }

    //Matrix C is what holds the multiplication matrix.
    clock_t t; //Timer variable
    t = clock(); //Snapshots the timer variable.
    //n x n is the size of the a b c matrices
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            for (int k = 0; k < n; k++)
            {
                //Need to access matrices in column major form. Results are sto
red in the retrieved element in C.
                C.ptr[i + j * n] += A.ptr[i + k * n] * B.ptr[k + j * n];
            }
        }
    }
    t = clock() - t; //Calculates the latency of this loop operation.
    double time_taken = ((double)t) / CLOCKS_PER_SEC; //Converts into seconds.

    printf("This loop configuration took %f seconds to execute.\n", time_taken)
;

    matrix_destroy(&A); matrix_destroy(&B); matrix_destroy(&C); //Frees the mal
locs.

    return 0;

}
```

Snippet 1: Implemented Program that times the loop operation of Figure 1 in IJK loop order.

The program described in Snippet 1 generates the three 5000x5000 matrices used to calculate an example matrix multiplication operation, and times the latency of said operation when using IJK order. Five other versions of this program exist but in the other loop orders: IKJ, JIK, JKI, KIJ, KJI.

The program begins by first generating Matrix A and B, which are multiplied together to create Matrix C. Before this, the three matrices are allocated in memory via *malloc*, which allocates 5000*5000 (or, 25,000,000 elements of memory) for each matrix. As malloc does not initialize allocated locations to a value. These values must be assigned values before being used. Matrices A and B are assigned values using the equations:

$$A[i] = i \,\%\, n \tag{1}$$

$$B[i] = i \,\%\, n \tag{2}$$

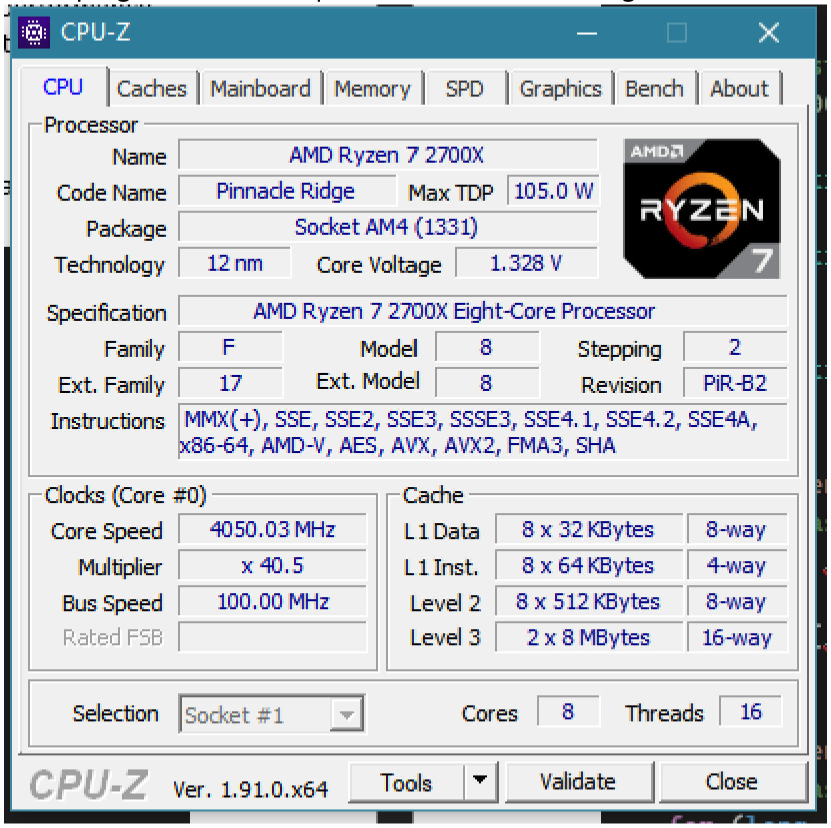
…which essentially assigns each corresponding column in the matrix with the same value. In other words, every k-th column in Matrices A and B will have the value k stored in that location. As what can be observed in the code, Matrices A and B are identical in values. After generating Matrices, A and B, Matrix C can be generated by implementing the loop as described in Figure 1. Before implementing this nested-loop, a *clock* variable is set to the current clock time so that the latency of the loop operation can be tracked. After this, the Matrix Multiplication Loop Operation is executed. This part of the code varies depending on the specified loop order. Once it completes, the time elapsed during this operation is taken, and then converted to seconds for the user to observe. The six versions of this software are automated using the following PowerShell script:

```
./loop_timing_IJK.exe
./loop_timing_IKJ.exe
./loop_timing_JIK.exe
./loop_timing_JKI.exe
```

```
./loop_timing_KIJ.exe
./loop_timing_KJI.exe
```
Snippet 2: PowerShell script used to gather data from each loop order.

Experiment Setup and Results

This experiment is conducted on a PC with an AMD Ryzen 7 2700X, which has 8 cores and 16 threads, clocked at 3.7 GHz. The programs were compiled on Windows 10 using MinGW GCC.



Processor
Name: AMD Ryzen 7 2700X
Code Name: Pinnacle Ridge — Max TDP: 105.0 W
Package: Socket AM4 (1331)
Technology: 12 nm — Core Voltage: 1.328 V
Specification: AMD Ryzen 7 2700X Eight-Core Processor
Family: F — Model: 8 — Stepping: 2
Ext. Family: 17 — Ext. Model: 8 — Revision: PiR-B2
Instructions: MMX(+), SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, SSE4A, x86-64, AMD-V, AES, AVX, AVX2, FMA3, SHA

Clocks (Core #0)
Core Speed: 4050.03 MHz
Multiplier: x 40.5
Bus Speed: 100.00 MHz
Rated FSB:

Cache
L1 Data	8 x 32 KBytes	8-way
L1 Inst.	8 x 64 KBytes	4-way
Level 2	8 x 512 KBytes	8-way
Level 3	2 x 8 MBytes	16-way

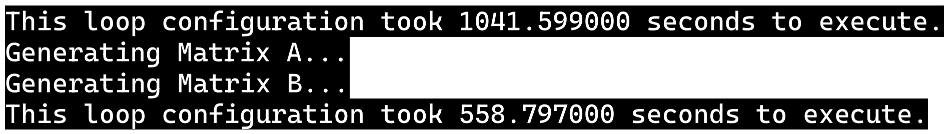
Selection: Socket #1 — Cores: 8 — Threads: 16

CPU-Z Ver. 1.91.0.x64

Figure 2: Profile of the CPU used to conduct the Experiment.

Upon completion of this PowerShell script, the following output is presented:

```
PS C:\Users\kaleb\Documents\GitHub\CSE611HW4> powershell -ExecutionPolicy
Bypass -File auto_run.ps1
Generating Matrix A...
Generating Matrix B...
This loop configuration took 726.472000 seconds to execute.
Generating Matrix A...
Generating Matrix B...
This loop configuration took 1027.663000 seconds to execute.
Generating Matrix A...
Generating Matrix B...
This loop configuration took 855.515000 seconds to execute.
Generating Matrix A...
Generating Matrix B...
This loop configuration took 409.224000 seconds to execute.
Generating Matrix A...
Generating Matrix B...
```



```
This loop configuration took 1041.599000 seconds to execute.
Generating Matrix A...
Generating Matrix B...
This loop configuration took 558.797000 seconds to execute.
```

Snippet 3: PowerShell Terminal output from running Snippet 2.

Using the data acquired from Snippet 3, the following table and figure can be generated:

Arrangement	**Latency (sec)**	**Latency (min)**		**Place**
JKI	409.224	6.8204		1
KJI	558.797	9.313283333		2
IJK	726.472	12.10786667		3
JIK	855.515	14.25858333		4
IKJ	1027.663	17.12771667		5
KIJ	1041.599	17.35998333		6

Table 1: Tabulation of Experiment Results

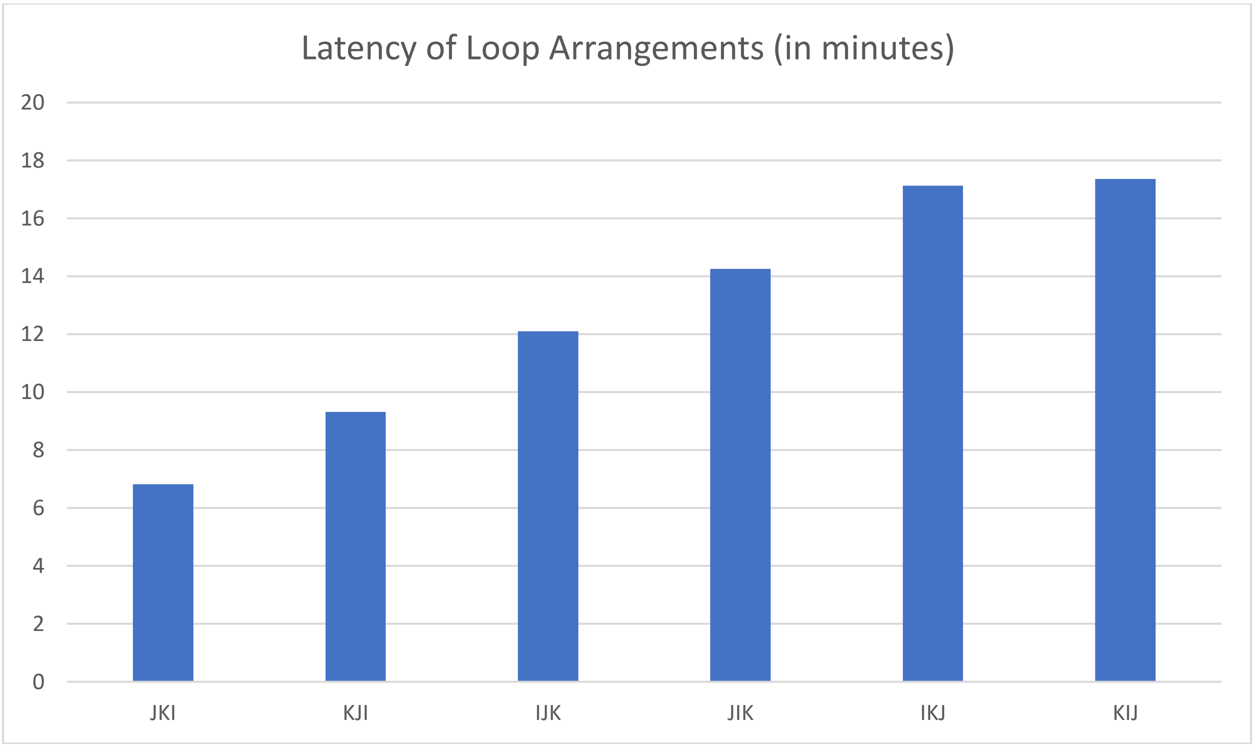


Figure 3: Latency of Loop Arrangements (in minutes)

As observed from the tabulated data, the loop-arrangement *JKI* yields the fastest matrix multiplication operation, while loop-arrangement *KIJ* yields the slowest matrix multiplication arrangement.

Investigation of Element Access Patterns

The main question that requires answering is: *what impact does loop order have on matrix multiplication latency?* To answer this question, an investigation of how elements of Matrices A, B and C are accessed in each loop order is necessary. The following Python program is used to analyze the order of which elements are accessed in Matrices A, B and C in each loop-order:

```
"""
CSE 611 HW4 Loop Order Testing Code
This program will analyze what elements are accessed when certain loop orders a
re used.
"""

import csv

n = 3

with open('IJKresults.csv', 'w', newline='') as csvfile:
    resultWriter = csv.writer(csvfile, delimiter=',', quotechar= "'", quoting=c
sv.QUOTE_MINIMAL)
    resultWriter.writerow(['i', 'j', 'k', 'A Location', 'B Location', 'C Locati
on'])

    for i in range(0,n):
        for j in range(0,n):
            for k in range(0,n):
                A_loc = (i+k*n)
                B_loc = (k+j*n)
                C_loc = (i+j*n)

                resultWriter.writerow([str(i), str(j), str(k), str(A_loc), str(
B_loc), str(C_loc)])
```

Snippet 4: Loop Order Testing program. Outputs order of elements accessed in each matrix.

Six variations of this program exist, one for each loop-order. The program simply generates a CSV file containing the current I, J, K values for each loop iteration, and the corresponding element locations for A, B, and C. The variable *n* determines the size of the matrix being analyzed, with 3 meaning a 3x3 matrix. As this code's purpose is to simply observe the pattern of how elements are accessed, a 5000x5000 matrix is not necessary for this program. A 3x3 matrix will be used instead to save computing time and output CSV file size. The following tables describe the results of the six variations of Snippet 4:

i	j	k	A Location	B Location	C Location
0	0	0	0	0	0
0	0	1	3	1	0
0	0	2	6	2	0
0	1	0	0	3	3
0	1	1	3	4	3
0	1	2	6	5	3
0	2	0	0	6	6
0	2	1	3	7	6
0	2	2	6	8	6

1	0	0	1	0	1
1	0	1	4	1	1
1	0	2	7	2	1
1	1	0	1	3	4
1	1	1	4	4	4
1	1	2	7	5	4
1	2	0	1	6	7
1	2	1	4	7	7
1	2	2	7	8	7
2	0	0	2	0	2
2	0	1	5	1	2
2	0	2	8	2	2
2	1	0	2	3	5
2	1	1	5	4	5
2	1	2	8	5	5
2	2	0	2	6	8
2	2	1	5	7	8
2	2	2	8	8	8

Table 2: IJK Loop Order Accessing Pattern, for a 3x3 Matrix.

i	j	k	A Location	B Location	C Location
0	0	0	0	0	0
0	1	0	0	3	3
0	2	0	0	6	6
0	0	1	3	1	0
0	1	1	3	4	3
0	2	1	3	7	6
0	0	2	6	2	0
0	1	2	6	5	3
0	2	2	6	8	6
1	0	0	1	0	1
1	1	0	1	3	4
1	2	0	1	6	7
1	0	1	4	1	1
1	1	1	4	4	4
1	2	1	4	7	7
1	0	2	7	2	1
1	1	2	7	5	4
1	2	2	7	8	7
2	0	0	2	0	2
2	1	0	2	3	5
2	2	0	2	6	8

2	0	1	5	1	2
2	1	1	5	4	5
2	2	1	5	7	8
2	0	2	8	2	2
2	1	2	8	5	5
2	2	2	8	8	8

Table 3: IKJ Loop Order Accessing Pattern, for a 3x3 Matrix

i	j	k	A Location	B Location	C Location
0	0	0	0	0	0
0	0	1	3	1	0
0	0	2	6	2	0
1	0	0	1	0	1
1	0	1	4	1	1
1	0	2	7	2	1
2	0	0	2	0	2
2	0	1	5	1	2
2	0	2	8	2	2
0	1	0	0	3	3
0	1	1	3	4	3
0	1	2	6	5	3
1	1	0	1	3	4
1	1	1	4	4	4
1	1	2	7	5	4
2	1	0	2	3	5
2	1	1	5	4	5
2	1	2	8	5	5
0	2	0	0	6	6
0	2	1	3	7	6
0	2	2	6	8	6
1	2	0	1	6	7
1	2	1	4	7	7
1	2	2	7	8	7
2	2	0	2	6	8
2	2	1	5	7	8
2	2	2	8	8	8

Table 4: JIK Loop Order Accessing Pattern, for a 3x3 Matrix

i	j	k	A Location	B Location	C Location
0	0	0	0	0	0
1	0	0	1	0	1
2	0	0	2	0	2

0	0	1	3	1	0
1	0	1	4	1	1
2	0	1	5	1	2
0	0	2	6	2	0
1	0	2	7	2	1
2	0	2	8	2	2
0	1	0	0	3	3
1	1	0	1	3	4
2	1	0	2	3	5
0	1	1	3	4	3
1	1	1	4	4	4
2	1	1	5	4	5
0	1	2	6	5	3
1	1	2	7	5	4
2	1	2	8	5	5
0	2	0	0	6	6
1	2	0	1	6	7
2	2	0	2	6	8
0	2	1	3	7	6
1	2	1	4	7	7
2	2	1	5	7	8
0	2	2	6	8	6
1	2	2	7	8	7
2	2	2	8	8	8

Table 5: JKI Loop Order Accessing Pattern, for a 3x3 Matrix

i	j	k	A Location	B Location	C Location
0	0	0	0	0	0
0	1	0	0	3	3
0	2	0	0	6	6
1	0	0	1	0	1
1	1	0	1	3	4
1	2	0	1	6	7
2	0	0	2	0	2
2	1	0	2	3	5
2	2	0	2	6	8
0	0	1	3	1	0
0	1	1	3	4	3
0	2	1	3	7	6
1	0	1	4	1	1
1	1	1	4	4	4
1	2	1	4	7	7

2	0	1	5	1	2
2	1	1	5	4	5
2	2	1	5	7	8
0	0	2	6	2	0
0	1	2	6	5	3
0	2	2	6	8	6
1	0	2	7	2	1
1	1	2	7	5	4
1	2	2	7	8	7
2	0	2	8	2	2
2	1	2	8	5	5
2	2	2	8	8	8

Table 6: KIJ Loop Order Accessing Pattern for a 3x3 Matrix

i	j	k	A Location	B Location	C Location
0	0	0	0	0	0
1	0	0	1	0	1
2	0	0	2	0	2
0	1	0	0	3	3
1	1	0	1	3	4
2	1	0	2	3	5
0	2	0	0	6	6
1	2	0	1	6	7
2	2	0	2	6	8
0	0	1	3	1	0
1	0	1	4	1	1
2	0	1	5	1	2
0	1	1	3	4	3
1	1	1	4	4	4
2	1	1	5	4	5
0	2	1	3	7	6
1	2	1	4	7	7
2	2	1	5	7	8
0	0	2	6	2	0
1	0	2	7	2	1
2	0	2	8	2	2
0	1	2	6	5	3
1	1	2	7	5	4
2	1	2	8	5	5
0	2	2	6	8	6
1	2	2	7	8	7

2	2	2	8	8	8

Table 7: KJI Loop Order Accessing Pattern for a 3x3 Matrix

Observing Loop-Order Accessing Patterns and Determination of Latency Impact
After analysis of these results, it can be determined that the reason loop-order JKI has the shortest operation latency is since iterative operations within the nested loop make greater use of cache locality than others. As observed in the columns "A Location," "B Location," and "C Location," elements in the A, B and C matrices are accessed in locations that are successive of one another, going through each element of the array of elements *in-order*, rather than jumping around to various values. This contrasts with what is observed in Table 6, which contains the order for Loop-Order KIJ, the loop-order that has the highest operation latency. In loop-order KIJ, values in matrices B and C are accessed in a pattern that has poor cache locality, as seen in instances where the C location is constantly jumped 3 spaces, from 3 to 6, and from 1 to 4, and 4 to 7, so on. Jumping to different locations in memory that are not continuous with the current location severely impacts program performance, as memory in C and C++ is allocated in a monolithic array. This means that memory elements that are closer to the original location are accessed faster than elements that are numerous locations away.

As it is now clear that cache locality is the main factor that decides iterative matrix multiplication latency, it can be concluded that the order of which loop-orders have the shortest latency as seen in Table 1, reflects that *loop-orders that yield lower latency integrate greater use of cache locality than those with slower latencies,* with loop-order JKI making use of the most cache locality while loop-order KIJ makes the least use of cache locality of the six arrangements. The Tables that correspond to each loop-order correlate to this finding, as for example, loop-order KJI makes more use of locality than loop0-order IJK, which are respectively 2nd and 3rd place in Table 1. The trend follows as Table 1 is progressed downwards.

Conclusion

In conclusion, this assignment explores the impact of loop-order in creating iterative matrix operations. As discussed previously, C and C++ allocates memory for values in monolithic single-dimension arrays, where the concept of a *matrix* is an *abstract of this fact*. Because of this, loop-order arrangements are just different patterns of accessing an allocated array of data, and as seen by the retrieved data from conducted experiments, loop-orders that preserve *cache locality*, or, iterative operations make use of data in memory locations that are near, if not beside, each other complete iterative operations with lower latencies compared to loop orders that make heavy use of jumps to different elements in a non-sequential order.