University of Miami, Electrical and Computer Engineering

Project 4: Caching in on Cache Performance
Assigned: April 1st, 2014
Due: April 21st, 2014 11:59pm

EEN 312, Spring 2014
Professor Eric W. D. Rozier

Maximum Score: 70 + 10pts

1 Pre-Lab

For this lab we will be examining the effects of caches on memory intensive code. We will be using the `Cachegrind` tool to simulate and investigate the cache performance of your code. `Cachegrind` is part of the `Valgrind` tool for memory debugging, memory leak detection, and profiling.

Read the cachegrind documentation to familiarize yourself with the tool:

`http://valgrind.org/docs/manual/cg-manual.html`

We will be running cachegrind with L1 instruction and data caches, and an L2 unified cache. Cachgrind's command-line options to specify the size and associativity of each cache with the following options:

```
Cachegrind-specific options are:

--I1=<size>,<associativity>,<line size>
   Specify the size, associativity and line size of the level 1 instruction cache.

--D1=<size>,<associativity>,<line size>
   Specify the size, associativity and line size of the level 1 data cache.

--LL=<size>,<associativity>,<line size>
   Specify the size, associativity and line size of the last-level cache.
```

2 Matrix Multiplication

30 points

Given matrices $A, B, andC$ with dimensions $i \times k$, $k \times j$, and $i \times j$ implement the matrix multiplication algorithms from Lecture 13, slides 33 to 40 in C++. Your program should read in data for the matrices from standard in, in the following form:

```
<i> <j> <k>

<a0,0> <a0,1> <a0,2> ... <a0,k-1>
<a1,0> <a1,1> <a1,2> ... <a1,k-1>
...
<ai-1,0> <ai-1,1> <ai-1,2> ... <ai-1, k-1>

<b0,0> <b0,1> <b0,2> ... <b0,j-1>
<b1,0> <b1,1> <b1,2> ... <b1,j-1>
...
<bk-1,0> <bk-1,1> <bk-1,2> ... <bk-1, j-1>
```

Where i, j, and k are the dimensions for the matrices. Your program should calculate the value

$$C = A \cdot B$$

Collect data for cache misses for a cache with the following parameters:

- L1 instruction cache
  - 4KB
  - 8-way associativity
  - 64 bytes per block
- L1 data cache
  - 4KB
  - 8-way associativity
  - 64 bytes per block
- L2 cache
  - 32KB
  - 8-way associativity
  - 64 bytes per block

Multiplying matrices of the following sizes:

- 32×32
- 128×128
- 512×512

Plot the resulting miss rates for each of the arrangements of i, j, and k.

Rerun the experiments, but this time make the L1 caches direct-mapped (i.e. 1-way associative), 2-way associative, and 4-way associative. Plot the resulting miss rates.

3 Blocked Matrix Multiplication

30 points

Implement a new version of your code from Section 2 using the algorithm for blocked matrix multiplication from Lecture 13, slide 48. Your program should read in data for the matrices from standard in, in the following form:

```
<i> <j> <k>

<a0,0> <a0,1> <a0,2> ... <a0,k-1>
<a1,0> <a1,1> <a1,2> ... <a1,k-1>
...
<ai-1,0> <ai-1,1> <ai-1,2> ... <ai-1, k-1>

<b0,0> <b0,1> <b0,2> ... <b0,j-1>
<b1,0> <b1,1> <b1,2> ... <b1,j-1>
...
<bk-1,0> <bk-1,1> <bk-1,2> ... <bk-1, j-1>
```

Where i, j, and k are the dimensions for the matrices. Your program should calculate the value

$$C = A \cdot B$$

Collect data for cache misses for a cache with the following parameters:

- L1 instruction cache
  - 4KB
  - 8-way associativity
  - 64 bytes per block
- L1 data cache
  - 4KB
  - 8-way associativity
  - 64 bytes per block
- L2 cache
  - 32KB
  - 8-way associativity
  - 64 bytes per block

Multiplying matrices of the following sizes:

- 32×32
- 128×128
- 512×512

Plot the resulting miss rates for each of the arrangements of i, j, and k.

Rerun the experiments, but this time make the L1 caches direct-mapped (i.e. 1-way associative), 2-way associative, and 4-way associative. Plot the resulting miss rates.

What size did you choose for the block matrices? Why? How and why did your results differ vs the non-blocked matrix multiplies?

4 Optimal Cache Sizes

10 bonus points

For the previous experiments, experiment with L1 data cache sizes and give an optimal cache size to minimize the miss rate for each of the matrix sizes and each of the associativities when using the algorithm from Section 2 (i.e., non-block matrix multiplication).

	1-way	2-way	4-way	8-way
32×32				
64×64				
128×128				
256×256				
512×512				
1024×1024				

5 Post-Lab

10 points

Prepare a post-lab report based on your results. Explain your performance results, and how the algorithm, matrix sizes, and cache associativity effect the results.

Lab 4 – Cache Performance

Nathan Paternoster

Description

We ran a series of experiments on performing matrix multiplication in different arrangements to analyze cache performance. We graphed the resulting miss rates from the different combinations of i, j, and k single matrix multiplication. Based on our graphs, we concluded that KIJ has the best miss rate (followed by IKJ) and KJI has the worst miss rate.

The larger the block size for the block matrix multiplication, the better. However we found that any block size above 16 did not produce efficient code.

The algorithm we used for our block matrix multiplication is shown below. We re-arranged the outer three for loops to get the different i, j, k combinations.

```
#include <stdlib.h>
#include <iostream>
#include <stdio.h>
#include <fstream>
#include <string>

using namespace std;

int main() {
	ifstream fin("/home/grayjin/project4/Matrix32");
	if (fin.fail())
		return -1;
	int i1, j1, k1;
	fin >> i1 >> j1 >> k1;
	int size = i1;
	int a[i1][k1], b[k1][j1], c[i1][j1];
	/* a */
	for (int x=0; x<i1; x++) {
		for (int y=0; y<k1; y++)
			fin >> a[y][x];
	}
	/* b*/
	for (int x=0; x<k1; x++) {
		for (int y=0; y<j1; y++)
			fin >> b[y][x];
	}
	  int B = 16;
```

```
            int n = i1;
            int sum = 0;
            int i, j, k;
            for (i = 0; i<n; i+=B) {
                for (j = 0; j<n; j+=B) {
                    for (k = 0; k<n; k+=B) {
                        /* B x B mini matrix multiplications */
                        for (int i0 = i; i0 < i+B; i0++) {
                            for (int j0 = j; j0 < j+B; j0++) {
                                for (int k0 = k; k0 < k+B; k0++) {
                                    c[i0][j0] += a[i0][k0]*b[k0][j0];                    }
                    }
                }
            }

            fin.close();

            return 0;
}
```

Post-Lab Report

Our performance results showed mostly consistent data for single-matrix multiplication. We verified that KIJ and IKJ are the most efficient algorithms and JKI and KJI are the least efficient. Greater associativity resulted in lower miss rates. Interestingly, our L1 data cache seemed to perform the worst of the three caches. Greater matrix sizes showed mostly slower performance with a few exceptions.

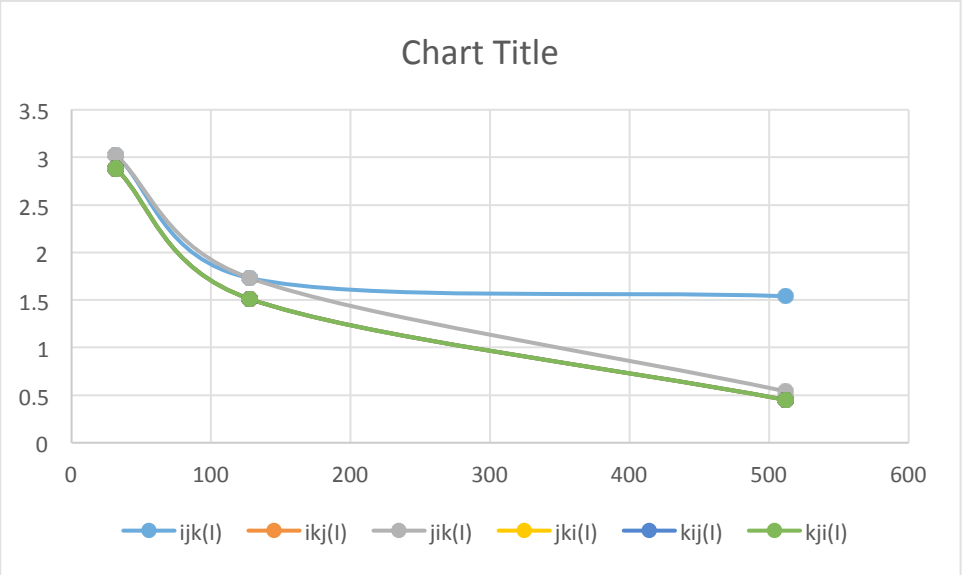
Using the block multiplication algorithm provided for us we seemed to obtain very similar results between i, j, k algorithms. However, although the miss rates were nearly uniform, the total number of misses varied between algorithms. In general we noticed that block multiplication seemed to favor larger matrix sizes (probably because larger matrix sizes allow the algorithm to make full use of sub-matrix blocks).

Results

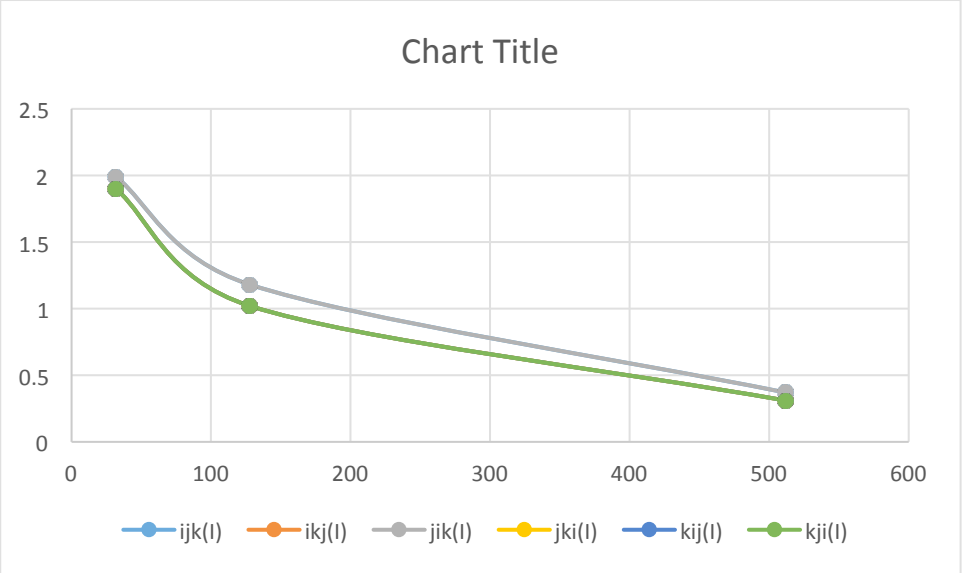
All of the graphs show miss rate (%) in the y-axis and matrix size in the x-axis. Different i, j, k algorithms are shown in different colors.

L1 Instruction Cache

Single Matrix Multiplication



1-Way



2-Way

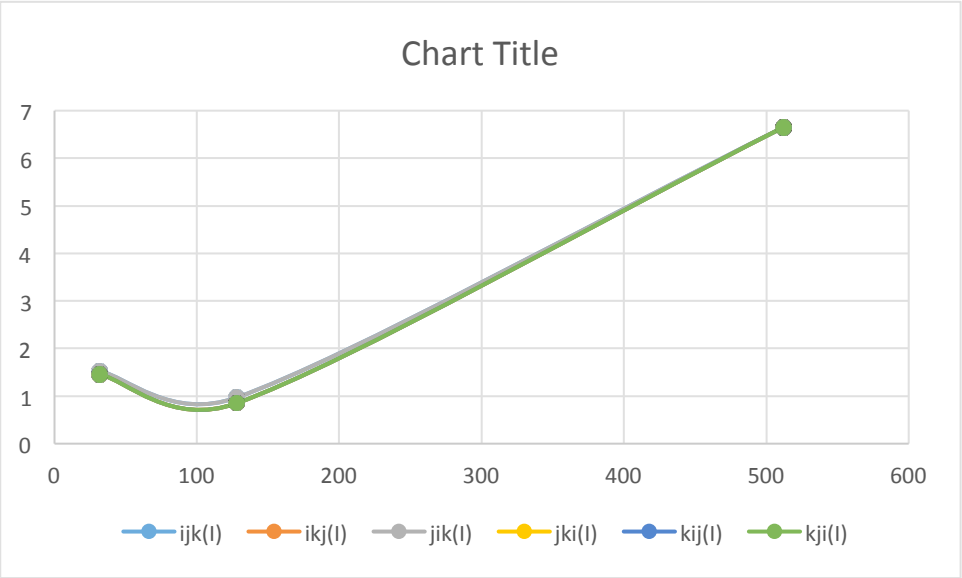


Chart Title
7
6
5
4
3
2
1
0
0
100
200
300
400
500
600
ijk(I)
ikj(I)
jik(I)
jki(I)
kij(I)
kji(I)

4-Way

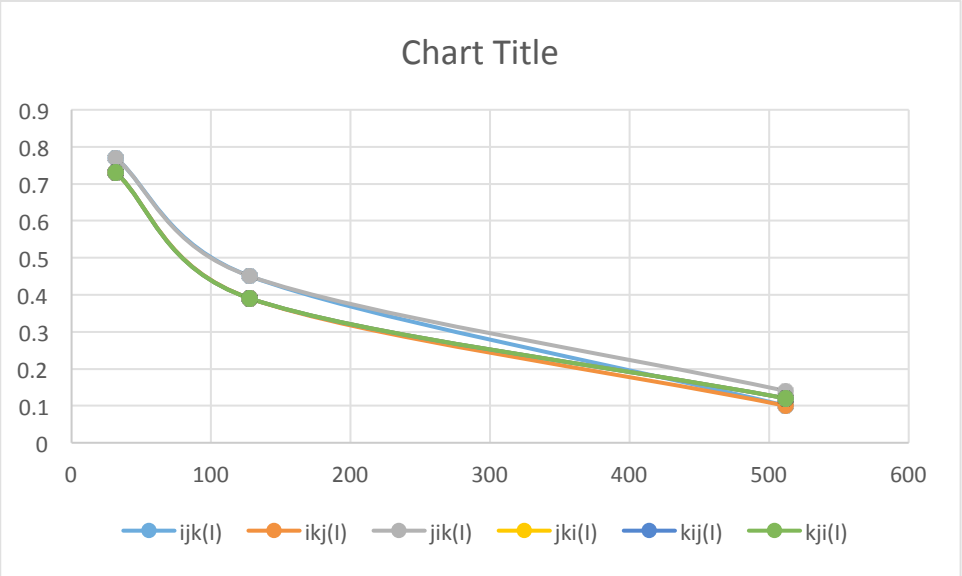
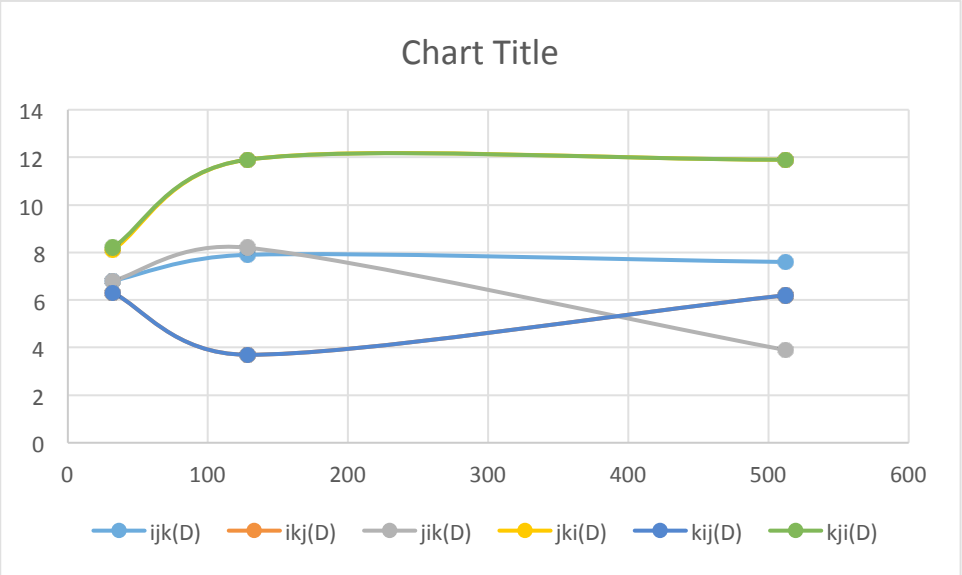


Chart Title
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
100
200
300
400
500
600
ijk(I)
ikj(I)
jik(I)
jki(I)
kij(I)
kji(I)

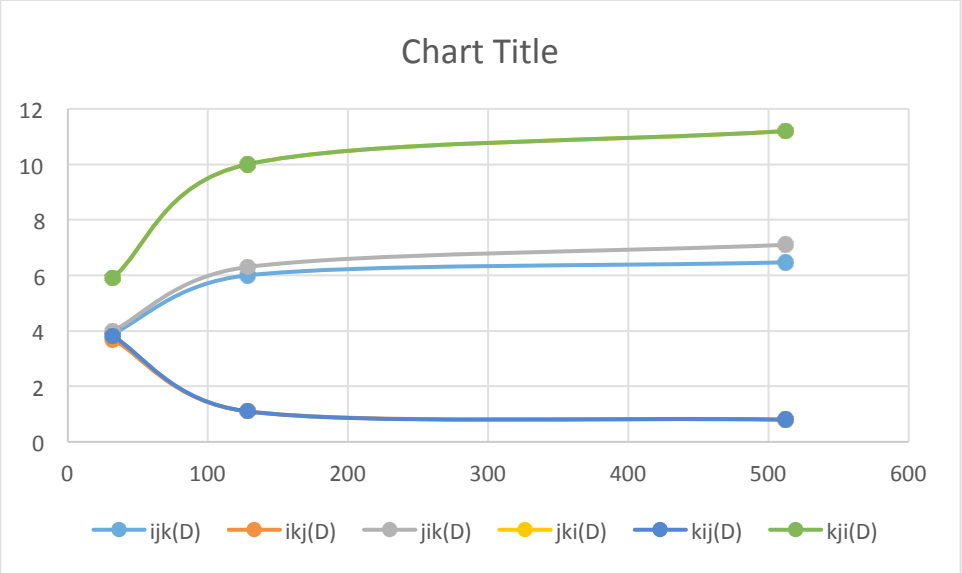
8-Way

L1 Data

Single Matrix Multiplication



1-Way



2-Way

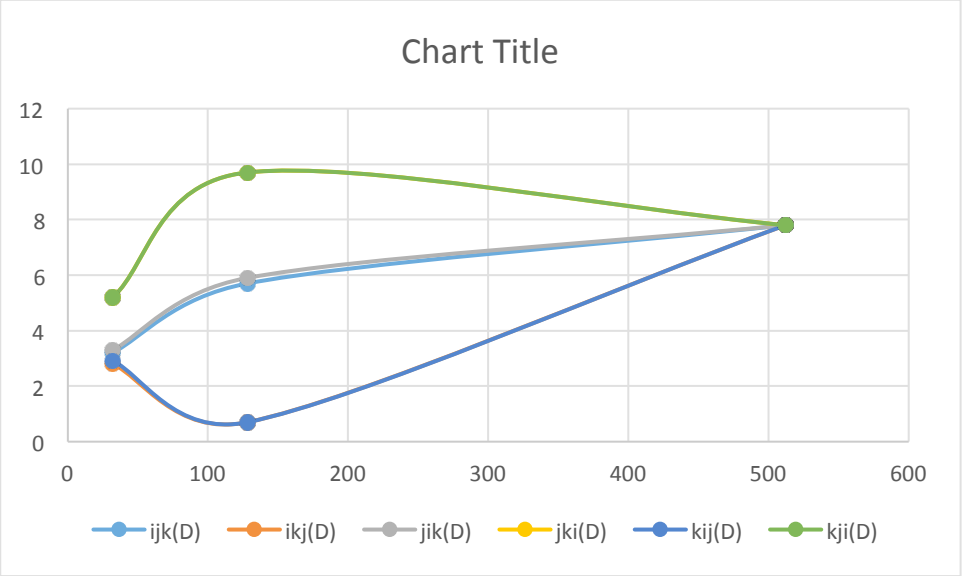


Chart Title
12
10
8
6
4
2
0
0
100
200
300
400
500
600
ijk(D)
ikj(D)
jik(D)
jki(D)
kij(D)
kji(D)

4-Way

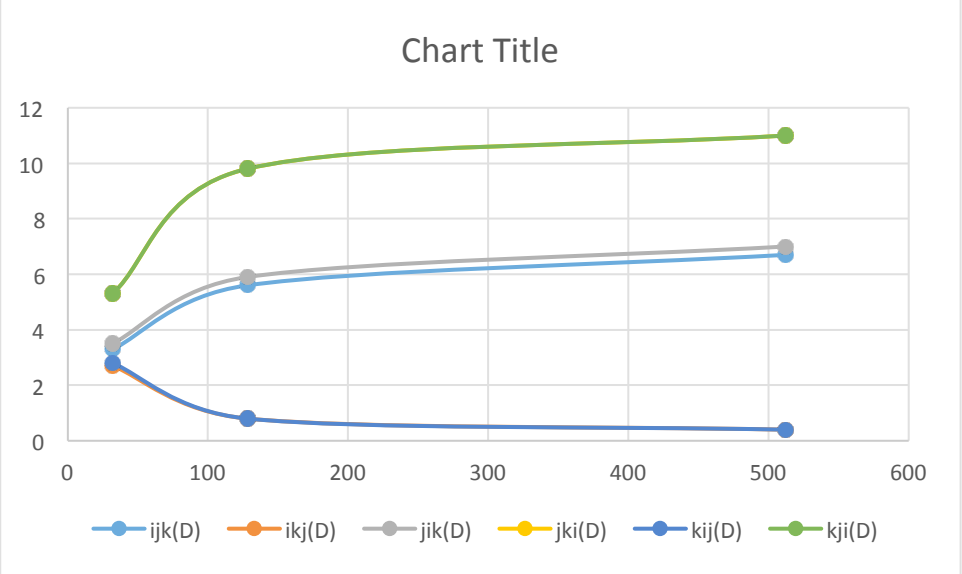
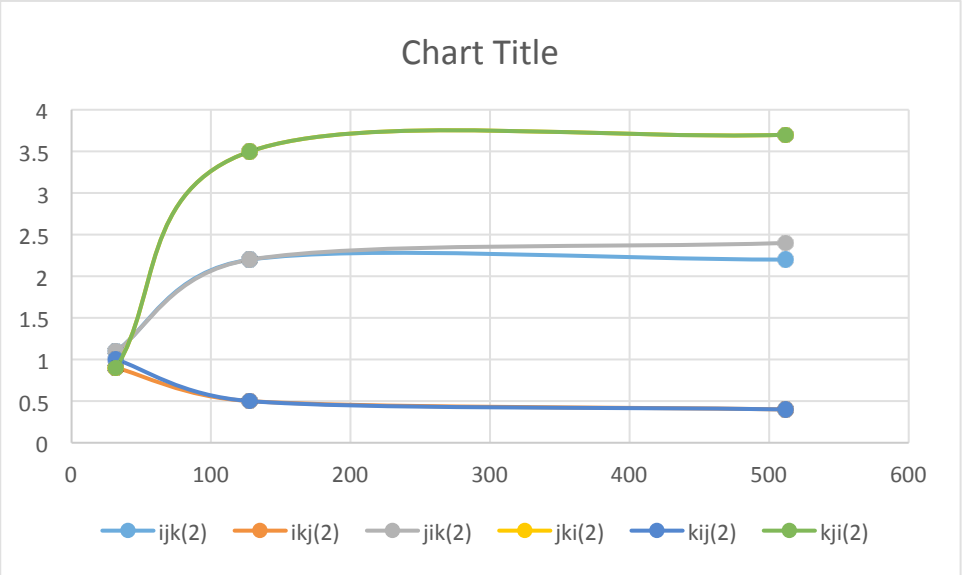


Chart Title
12
10
8
6
4
2
0
0
100
200
300
400
500
600
ijk(D)
ikj(D)
jik(D)
jki(D)
kij(D)
kji(D)

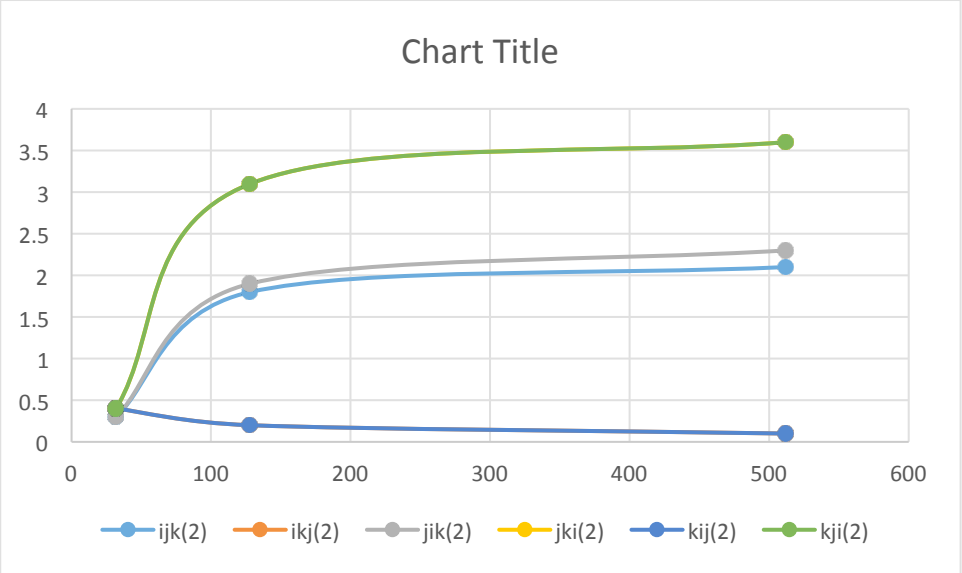
8-Way

L2 Cache

Single Matrix Multiplication



1-Way



2-Way

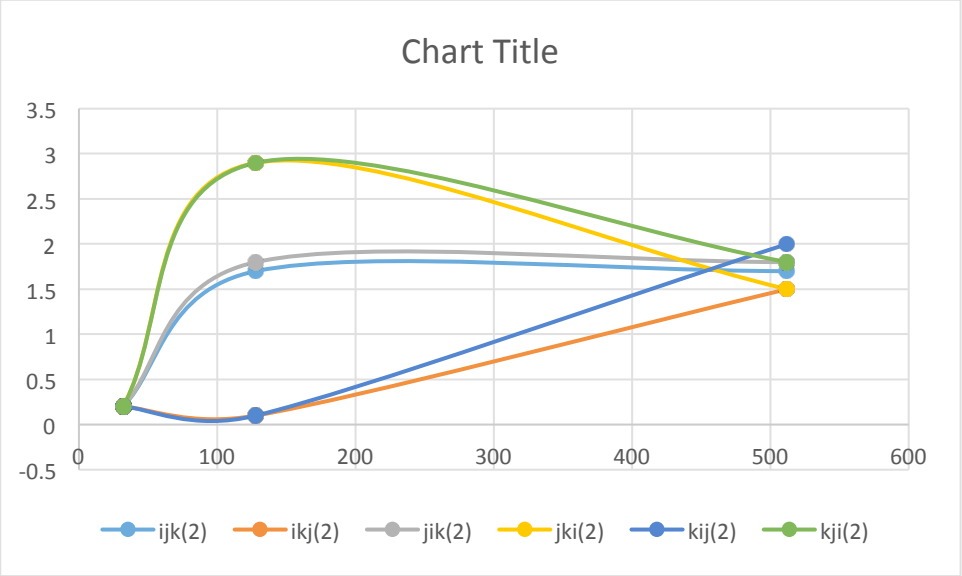


Chart Title
3.5
3
2.5
2
1.5
1
0.5
0
-0.5
0
100
200
300
400
500
600
ijk(2)
ikj(2)
jik(2)
jki(2)
kij(2)
kji(2)

4-Way

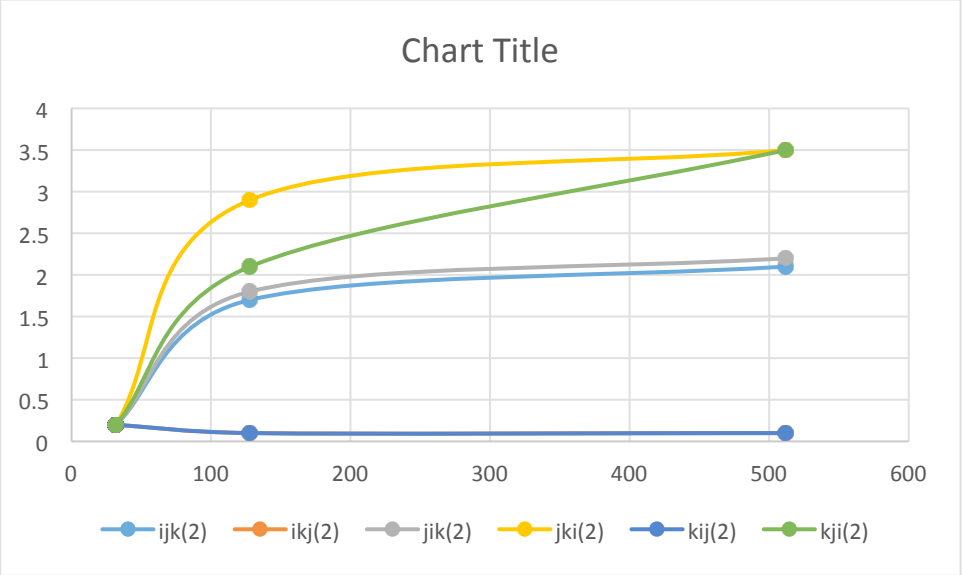


Chart Title
4
3.5
3
2.5
2
1.5
1
0.5
0
0
100
200
300
400
500
600
ijk(2)
ikj(2)
jik(2)
jki(2)
kij(2)
kji(2)

8-Way