Performance Study of Matrix Multiplication Algorithm
Term Project, CSC 699, Spring 2023

Zachary Colbert*
SFSU

ABSTRACT

Text similarity is useful in a variety of applications including machine learning, plagiarism detection, and data association. Matrix multiplication is the core operation of common text similarity algorithms, and comprises a substantial portion of the runtime and computational cost. The problem we aim to study is how this matrix multiplication algorithm can be improved to minimize runtime and scale more effectively to larger data sets.

Our approach explores these variations on the algorithm:

- Loop interchange
- Blocking
- Vectorization and SIMD instructions
- Compiler optimizations
- Parallelism

The results show that certain combinations of these optimizations produce a speedup of more than 500x when compared to a naive implementation of the matrix multiplication algorithm.

1 INTRODUCTION

Matrix multiplication is the primary computational kernel for common implementations of lexical text similarity algorithms. Because matrix multiplication is a computationally demanding algorithm that scales poorly as the input size grows, it is crucial that the algorithm is executed with as much efficiency as possible to allow processing of large data volumes with minimal performance degradation.

In addition to careful selection of data structures, the runtime characteristics of a matrix multiplication algorithm can be greatly improved by leveraging the architectural properties of the target system. The purpose of this paper is to describe the effects of various optimizations of the matrix multiplication algorithm in the context of lexical text analysis.

Certain lexical text analysis algorithms use the cosine angle between documents as a measurement of similarity. This process involves numerically tokenizing the documents, and storing the tokens in a term-frequency matrix where each row vector contains the tokenized representation of a single document. To compute similarity, the normalized matrix is multiplied against its transpose to produce a result matrix where each element describes the similarity between two documents.

By normalizing the matrix prior to multiplication, the relationship between the dot-product and the identity of the cosine function can be exploited to produce a cosine angle by computing the dot product of two document vectors.

The definition of cosine is $\cos\theta = \frac{\vec{a}\cdot\vec{b}}{||\vec{a}||||\vec{b}||}$

If the magnitudes of $\vec{a}$ and $\vec{b}$ are 1 (as is the case with normalized vectors), then this simplifies to $\cos\theta = \vec{a}\cdot\vec{b}$

*email:zcolbert@sfsu.edu

Since matrix multiplication is performed as a series of dot products, then the result matrix can be interpreted as a collection of cosine angles describing the relationship between the multiplied vectors.

The computational demands of this algorithm scale not only to the number of documents, but also the number of unique tokens in the document set, referred to as the vocabulary. When the term-frequency matrix is assembled, it will have the dimensions of N x M where N is the number of documents, and M is the vocabulary size. In general, the algorithmic complexity for a matrix multiplication of an N x M matrix is $O(N^2M)$. For cases where $M \geq N$, meaning the number of unique tokens is at least as large as the number of documents (a common case), this is approximately $O(N^3)$.

The performance of this algorithm can be initially improved through careful selection of data structures. Since each document will not necessarily contain every token in the vocabulary, the resulting term frequency matrix is quite sparse with a high frequency of zeroes in each vector. Memory usage and execution time can both be improved if this is first converted into a dense matrix. However, these improvements are based on reduction of N, so any gains are overshadowed when N scales up again by introduction of a larger data set. With these factors considered, it is clear that the number of operations is unavoidably large and so those operations must be performed with efficiency in mind.

Through analysis and measurement of the matrix multiplication algorithm, I was able to identify performance bottlenecks and opportunities for improvement which ultimately resulted in more than a 500 times increase in execution speed compared to a base implementation by way of improved cache locality, SIMD vectorization, and OpenMP parallelism.

2 RELATED WORK

In their book *"Computer Architecture: A Quantitative Approach"*, John L. Hennessy and David A. Patterson describe two potential improvements to the performance of matrix multiplication. The first technique is referred to as "loop interchange", which is the reordering of nested loops with the goal of improving memory access patterns. In certain cases, loops can be ordered in such a way that results in fewer cache misses, which should reduce the runtime of a memory-bound algorithm. The second technique called "blocking" aims to improve temporal cache locality by subdividing the work and deliberately placing small subsets of the problem's data into the cache [1]. Since we are developing for a cache-based architecture, both of these techniques seem quite promising and will be explored in detail. However, the examples offered by Hennesy and Patterson are purely theoretical and do not explore the results of these optimizations in the context of a real problem on a concrete system. One goal of this paper is to quantify the effects of these optimizations on a real-world system.

3 IMPLEMENTATION

To understand the runtime characteristics of matrix multiplication in the context of lexical text similarity, a basic implementation must be measured as a point of reference. The first step was to implement

the lexical text similarity algorithm and collect initial benchmarks as a point of reference. The algorithm was implemented naively at first, with the design approach preferring the simplest code possible without consideration for performance implications.

I fed the program with a large volume of test data consisting of movie reviews pulled from the internet. To ensure correctness, I compared the computed results against a benchmark implementation using the Python SciKitLearn package. To stress test the matrix multiplication subsystem, I parameterized the program to deterministically generate an NxN matrix of floating point values. This allowed control over the dimensions of the input matrix, as well as the ability to easily scale beyond the size of the available data set. The program may also be invoked with additional parameters to specify which version of the algorithm should be performed. These parameters included selection of the loop order, a flag to enable or disable blocking optimization, selection of block size, matrix dimensions, and test data sources.

I captured performance data using the LIKWID suite of hardware performance counters. The initial runs captured a variety of metrics including FLOPS, memory and cache bandwidth and data volume, and runtimes (per core and cumulative). This data was used as a starting point to identify possible bottlenecks and opportunities for improvement.

Several techniques were chosen for their potential to measurably improve specific aspects of the program's runtime performance, including cache locality, vectorization, and parallelism. Each time an optimization was added, new benchmarks were collected and compared against previous versions to quantify the effects of the change. This process was repeated until all of the chosen techniques were implemented and an optimal combination of these techniques was identified.

Base Implementation

As a starting point, the basic implementation of matrix multiplication (see listing 1) populates a result array with the dot products of each element in the left and right operand matrices. This will be referred to as the "ijk" version of the algorithm, named after the ordering of its loop index variables. This naming convention will also be used to describe the other permutations of this triple nested loop.

```
// result is an N x N matrix storing the
// results of the multiplication,
// lhs and rhs are the left and right
// operand matrices, respectively.
void matrixMultiply(lhs[], rhs[], result[], n, m)
{
    for (i = 0; i < n; ++i) {
      for (j = 0; j < n; ++j)
        for (k = 0; k < m; ++k)
          result[i*n+j] += lhs[i*m+k]*rhs[k*n+j];
}
```

Listing 1: Basic matrix multiplication.

The index into the result matrix is calculated using i and j to find an offset into the array. With this pattern, i * n calculates the starting position of a row, which is then offset by j columns. The order in which these indices are incremented dictates the stride, or the pattern of memory access into this array. Depending on the layout of memory addresses in the target system, the stride can have a significant effect on the algorithm's performance. We can test the effects of different strides by changing the order of these nested loops, a process known as loop interchange.

Loop Interchange

Loop interchange is simply the reordering of nested loops. For the matrix multiplication algorithm, the nested loops will be referred to by their index variables i, j, and k. These three loops can be reordered into six permutations, all of which were evaluated.

Since calculation of the index into the result array is dependent on these loop indices, the expected consequence of loop interchange is a variation in cache usage due to differences in stride, which is the memory traversal pattern. When i is the outermost loop, the array will be accessed in a row-wise pattern. For each row that is loaded, the columns offset by j are accessed in sequence before retrieving the next row. When j is the outermost loop, the pattern would be column-wise instead. The effects of these patterns are system dependent, since different platforms may store arrays in either a row-major or column-major memory layout. Since C++ uses a row-major memory layout, a row-wise access pattern would be preferable to column-wise traversal. For a row-wise traversal, the sequential access of memory addresses should benefit from the spatial locality of data in the cache. If traversing across the rows in a column, the memory accesses will stride across addresses separated by N, effectively guaranteeing a cache miss when N is larger than the size of the cache line.

Blocking Optimization

For a memory bound algorithm, runtime can be improved by reducing the time spent waiting for data to move between the CPU and main memory. This is achieved by maximizing usage of the CPU caches. One way to use the caches more effectively is to leverage the property of temporal locality – the likelihood that recently used data is likely to be used again soon. When data is loaded into the cache, it should be kept there until it is no longer needed to prevent repeated loading and unloading of the same data. This can be done deliberately by a technique known as blocking: subdividing a problem into blocks which are small enough to fit in the CPU cache, and computing a result of that subproblem before moving onto the next.

To measure the effects of blocking on the matrix multiplication, the problem is divided into three B x B square submatrices: One each for the result, the left operand, and the right operand.

Several values of B ("block size") were chosen as powers of two, covering a range that would:

- fit entirely in the L1 cache (TotalBytes $<$ L1)
- fit into L2 but not L1 (L1 $<$ TotalBytes $<$ L2)
- fit into L3 but not L2 (L2 $<$ TotalBytes $<$ L3)
- exceed the size of L3 (L3 $<$ TotalBytes)

The size of each block (in bytes) is computed as:

$$BlockSize = sizeof(float) \times B \times B = 4B^2$$

Since there are three blocks required, the total number of bytes is:

$$TotalBytes = 3 \times BlockSize = 12B^2$$

The expected result of this blocking optimization is that, for a memory-bound operation, runtime should be reduced when the block size is appropriate (those which fit into the L1, L2, or L3 cache) when compared to the same algorithm with block sizes exceeding the largest cache size, or with no blocking optimization at all. Table 1 shows a breakdown of each selection of block size as it relates to the various CPU caches.

Compiler Optimization and Vectorization

B	Block Size (bytes)	Total Bytes	Fits in Cache
16	1,024	3,072	L1
32	4,096	12,288	L1
64	16,384	49,152	L2
128	65,536	196,608	L2
256	262,144	786,432	L3
512	1,048,576	3,145,728	L3
1024	4,194,304	12,582,912	>L3
2048	16,777,216	50,331,648	>L3

Table 1: Table showing which selections of block size will fit into each of the CPU cache levels. Selections that fit into a lower-level cache (e.g. L1) will also fit into higher-level caches (e.g. L2, L3). Rows with cache $> L3$ indicate that the block data is large to fit into the L3 or any lower level cache.

Optimizing compilers open the door to a whole suite of optimizations which are available for little to no programmer effort. When working with large volumes of data that have a similar type, such as a matrix of floating point values, one optimization of particular interest is vectorization. Since the same operation is being performed repeatedly on many values, the use of SIMD (single instruction, multiple data) instructions is a natural fit.

The g++ compiler supports a number of optimization flags, including several aimed at automatic vectorization. Vectorization optimizations begin appearing at optimization level 2 (-O2), and any higher optimization levels above this.

Opportunities for vectorization can be challenging for the compiler to identify, and sometimes code must be structured with this optimization in mind. To compare whether loop interchange also aids the compiler in performing vectorization, all six loop permutations are tested at various optimization levels from 0 (no optimization) to 3 (all optimizations at level 1, 2, and 3).

Paralellism

The objective of the previous steps was to find the best performing serial implementation. With this identified, we can take advantage of the system's additional cores to perform the already optimized work in parallel. Parallelism can be achieved in a number of ways including multiprocessing or the lighter-weight multithreading. The latter has less overhead, but still requires careful organization of the code to prevent data integrity issues such as deadlocks or race conditions. Fortunately, this burden is partially alleviated by taking advantage of OpenMP which is supported in modern versions of the Gnu Compiler Collection. OpenMP can be enabled by passing the -fopenmp flag during compilation and then placing the appropriate pragmas into the code.

While OpenMP does alleviate the burden of creating and managing multiple threads, it does not prevent the possibility of data races so the code and memory access must still be carefully organized and critical regions must be protected where necessary. Critical regions are sometimes necessary, but must be used judiciously to avoid over serializing the computation and eliminating the benefits of multithreading.

When creating a parallel region around the triple-nested matrix multiplication loops, we can use the properties OpenMP's work division to our advantage. For loop orders where the array index calculation is not dependent on the inner-most loop variable, each thread will have non-overlapping ranges of array indices. This means that we are guaranteed that threads will not access the same regions of memory, so we can avoid the cost of a serialized critical region. (See listing 2).

When the index calculation is dependent on the inner loop variable, there is no guarantee that the threads will not attempt to access the same memory location at the same time so measures must be

```
void matrixMultiply(lhs[], rhs[], result[], n, m)
{
    // i will divide into non-overlapping ranges
    #pragma omp parallel for
    for (i = 0; i < n; ++i) {
      for (j = 0; j < n; ++j)
        for (k = 0; k < m; ++k)
          result[i*n+j] += lhs[i*m+k]*rhs[k*n+j];
}
```

Listing 2: OpenMP parallelism of Matrix Multiplication with non-overlapping indices.

taken to protect the shared memory. One approach would be to add a critical region around the memory access into the result array. However, this effectively serializes the entire workload, eliminating the benefit of using multiple threads altogether. (See listing 3).

```
void matrixMultiply(lhs[], rhs[], result[], n, m)
{
    #pragma omp for
    for (k = 0; k < m; ++k)
      for (i = 0; i < n; ++i)
        for (j = 0; j < n; ++j) {
          #pragma omp critical
          result[i*n+j] += lhs[i*m+k]*rhs[k*n+j];
        }
}
```

Listing 3: Careless locking of critical region causes over-serialization.

A better solution is to implement thread local storage to hold partial solutions for each thread, then combine these into the final result in a protected serial region (see listing 4). This approach has a tradeoff of reducing the portion of serialized code at the cost of more memory, since each thread requires its own copy of the result array.

```
void matrixMultiply(lhs[], rhs[], result[], n, m)
{
    #pragma omp parallel  // begin parallel region
    {
    // Create thread-local storage to hold
    // results of computation in each thread.
    // Each thread will have its own copy of
    // this vector containing a partial result.
      vector<float> tls(result.size(), 0);

      #pragma omp for
      for (int k = 0; k < m; ++k)
        for (int i = 0; i < n; ++i)
          for (int j = 0; j < n; ++j)
            tls[i*n+j] += lhs[i*m+k]*rhs[k*n+j];

      // Update the shared result vector
      // with this thread's partial result.
      for (size_t i = 0; i < tls.size(); ++i) {
        #pragma omp atomic
        result[i] += tls[i];
      }
    }  // end parallel region
}
```

Listing 4: Minimizing serial region using thread-local copies of the result matrix.

4 EVALUATION

All tests were performed using the same system running an Intel CPU on a Linux operating system. See table 2 for a detailed breakdown of system specifications.

System Specifications		
Operating System	Linux Mint v19.3 (Tricia)	
Linux Kernel	5.4.0-139-generic	
Compiler	g++ 7.5.0	
CPU	Architecture	Haswell
	Cores	4
	Clock	3.20 GHz
	L1	32 kB (per core)
	L2	256 kB (per core)
	L3	6 MB (shared)
Memory	16GB DDR3 @ 1600 MHz (2x 8GB DIMM)	

Table 2: System specifications for the testing platform.

Tests were performed to measure the effects of a combination of several parameters including:

- Loop Order: ijk, ikj, jik, jki, kij, kji
- Block Size: None, 16, 32, 64, 128, 256, 512, 1024, 2048
- Compiler optimization flags: O0, O1, O2, O3
- Number of threads: 1, 2, 4

Altogether there are 648 possible combinations of these parameters, all of which were tested and measured for comparison. The program was designed to accept parameters for selection of various features at runtime including loop order, block size, test data source, and an optional override for the matrix dimensions. The number of threads was specified by setting the `OMP_NUM_THREADS` environment variable prior to execution. To test different compiler optimization levels, four different versions of the executable were compiled using optimization levels 0 through 3, inclusive.

I captured runtime data using LIKWID performance counters. The primary metrics of interest were the program's runtime, and the volume of data being moved through the L3 cache.

Results - Loop Interchange

When comparing all permutations of loop order for the matrix multiplication algorithm, the results is that loops with j as the innermost loop had the best performance, while loops with i as the innermost loop had the worst. In table 3 we can see that the total volume of data loaded into and evicted from the L3 cache correlates with the runtimes (figure 1). For the loops with the lowest runtime, there is very little data being moved through the cache, indicating that spatial locality has been improved when compared to the worst performers who have a very high volume of data being moved through the L3 cache from main memory. The decrease in runtime for cases of lower memory volumes indicates that this algorithm is memory-bound and there is a bottleneck in performance when the CPU is waiting for data to load from main memory. These results agree with the prediction that variations on the algorithm which use row-wise, sequential memory access patterns will perform better than those with random or column-wise access patterns due to improved spatial cache locality.

Results - Blocking

Table 4 depicts the effect of blocking at various sizes of N. As predicted, for block sizes that fit into any of the three caches, the runtime is significantly improved. When the block size is 2048, the blocks no longer fit into the caches and the runtime resembles

Loop Interchange Runtimes			
Loop	Runtime (s)	L3 Vol. (GB)	Speedup
ijk	343.525	559.74	1.00
ikj	78.394	37.64	4.38
jik	331.566	590.65	1.04
jki	387.778	1,539.17	0.89
kij	78.606	72.37	4.37
kji	390.453	1,542.70	0.88

Table 3: Comparison of runtimes for variations of loop order. Values in the speedup column indicate runtime improvement relative to the base implementation (highlighted).

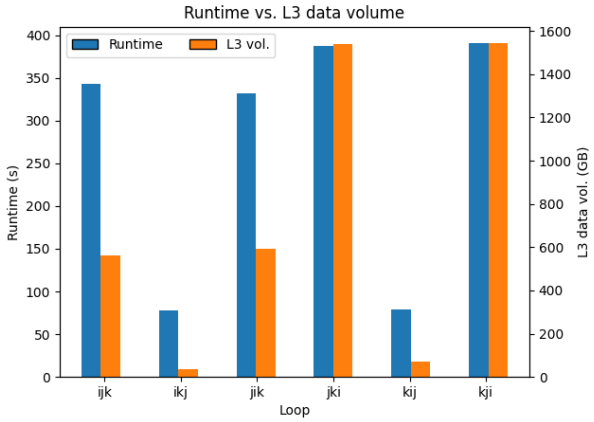

Figure 1: Comparison of runtime vs. volume of data moved through the L3 cache for different orders of the matrix multiplication algorithm's nested loops.

when there was no blocking at all (figure 2). For certain loop orders (where spatial cache locality was already high), there is a negligible change in runtime since the array values were already in the cache when needed (see table 5, figure 3). There was approximately a 4x speedup for any given block size, which is the same speedup that was achieved without blocking optimization.

Runtime vs. L3 data volume for ijk loop				
Loop	Block Size	Runtime (s)	L3 Vol. (GB)	Speedup
ijk	0 (no blocking)	343.525	559.74	1.00
ijk	16	93.545	11.28	3.67
ijk	32	85.027	6.63	4.04
ijk	64	81.484	4.77	4.22
ijk	128	80.286	5.00	4.28
ijk	256	84.103	378.42	4.08
ijk	512	92.594	552.30	3.71
ijk	1024	158.357	554.66	2.17
ijk	2048	332.200	559.00	1.03

Table 4: Comparison of runtimes vs. data volume moved through the L3 cache using different block sizes. Values in the speedup column indicate runtime improvement relative to the base implementation (highlighted).

Runtime vs L3 data volume for ikj loop				
Loop	Block Size	Runtime (s)	L3 Vol. (GB)	Speedup
ikj	0 (no blocking)	78.394	37.64	4.38
ikj	16	93.947	11.34	3.66
ikj	32	84.911	6.64	4.05
ikj	64	81.024	4.70	4.24
ikj	128	79.708	4.64	4.31
ikj	256	78.760	18.24	4.36
ikj	512	78.454	38.28	4.38
ikj	1024	78.868	37.73	4.36
ikj	2048	78.501	37.85	4.38
ijk	0 (no blocking)	343.525	559.74	1.00

Table 5: Comparison of runtimes vs. data volume moved through the L3 cache using different block sizes. Values in the speedup column indicate runtime improvement relative to the base implementation (highlighted).

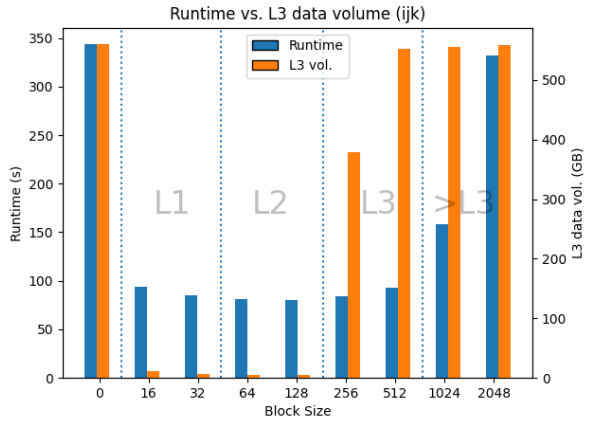

Figure 2: Comparison of runtime vs. volume of data moved through the L3 cache for different block sizes of the I-J-K loop order.

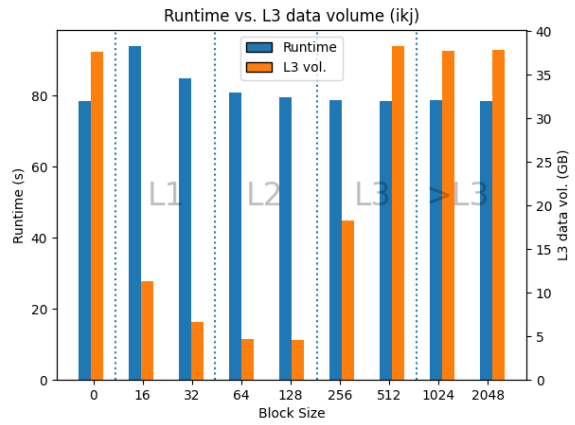

Figure 3: Comparison of runtime vs. volume of data moved through the L3 cache for different block sizes of the I-K-J loop order.

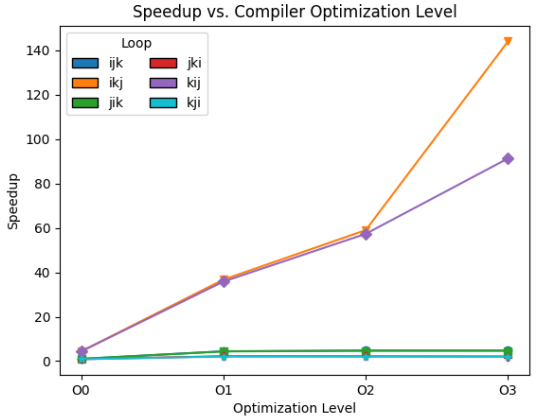

Figure 4: Comparison of speedup at different compiler optimization levels.

Results - Compiler Optimization and Vectorization

The GNU C Compiler lists vectorization among the optimizations performed at level 2 optimization (-O2), and since higher optimization levels include optimizations from lower levels, vectorization will also be performed at optimization level 3 (-O3). A vectorization report produced by the compiler indicates that vectorization was attempted only at optimization level 3, and only in certain loop orders: ikj, and kij. The lack of vectorization when compiling at optimization level 2 indicates that some additional optimization which is present only in level 3 may have facilitated vectorization or altered the code to make the potential for vectorization more apparent to the compiler versus the code produced by level 2 optimization.

Inspection of the assembly for this code reveals the usage of the 128-bit XMM registers and their related arithmetic instructions, which indicates that 128 bit vectorization has occurred successfully, allowing computation of four 4-byte floating point numbers simultaneously with a single instruction. When compared with the remaining four loop orders, the ikj and kij loops have a consistent increase in speedup with each successive compiler optimization level (table 6, figure 4). Since vectorization is not even attempted until level 2 optimization, this indicates that some other optimization or combination of optimizations is occurring on these two loops but not on the others.

Results - Parallelism

In general, when each variation of the matrix multiplication algorithm was run using multiple threads there was a decrease in runtime when compared to that same algorithm's serial execution (table 7, figure 5). This speedup factor was decreased in cases where critical regions were used (listing 4) due to the presence of a serial region in the code.

Results - Cumulative

The overall result of this experiment was a speedup of 515x over the base implementation when all of these optimizations were applied with the appropriate parameters (table 8). The base implementation falls into the bottom 5 worst performers (table 9), at number 644 out of 648 total combinations tested.

When the best and worst results are compared, there are some noticeable similarities that reveal patterns supporting the conclusions for each of the explored optimizations.

Loop order: All of the top 10 performers were of loop order ikj, where the stride was row-wise and memory was accessed sequentially. The row order for the bottom ten was varied, but none had j (which produces the column index) as their inner-most loop.

Runtime vs. Compiler Optimization Level				
Loop	Opt. Flag	Vectorized	Runtime (s)	Speedup
ijk	O0	N	343.525	1.00
ijk	O1	N	77.299	4.44
ijk	O2	N	71.580	4.80
ijk	O3	N	72.030	4.77
ikj	O0	N	78.394	4.38
ikj	O1	N	9.361	36.70
ikj	O2	N	5.822	59.00
ikj	O3	Y	2.386	143.99
jik	O0	N	331.566	1.04
jik	O1	N	78.480	4.38
jik	O2	N	73.977	4.64
jik	O3	N	73.585	4.67
jki	O0	N	387.778	0.89
jki	O1	N	155.530	2.21
jki	O2	N	158.402	2.17
jki	O3	N	163.763	2.10
kij	O0	N	78.606	4.37
kij	O1	N	9.587	35.83
kij	O2	N	5.994	57.31
kij	O3	Y	3.763	91.29
kji	O0	N	390.453	0.88
kji	O1	N	165.381	2.08
kji	O2	N	166.523	2.06
kji	O3	N	171.213	2.01

Table 6: Comparison of runtimes for different loop orders at various levels of compiler optimization. Values in the speedup column indicate runtime improvement relative to the base implementation with no optimization (highlighted).

Runtime vs. Number of Threads			
Loop	NThreads	Runtime (s)	Speedup
ijk	1	343.525	1.00
ijk	2	164.118	2.09
ijk	4	96.493	3.56
ikj	1	78.394	4.38
ikj	2	40.342	8.52
ikj	4	21.591	15.91
jik	1	331.566	1.04
jik	2	179.288	1.92
jik	4	86.084	3.99
jki	1	387.778	0.89
jki	2	207.612	1.65
jki	4	123.322	2.79
kij	1	78.606	4.37
kij	2	38.692	8.88
kij	4	20.818	16.50
kji	1	390.453	0.88
kji	2	207.485	1.66
kji	4	113.994	3.01

Table 7: Comparison of runtimes for different loop orders at 1, 2, and 4 threads. Values in the speedup column indicate runtime improvement relative to the base implementation (highlighted).

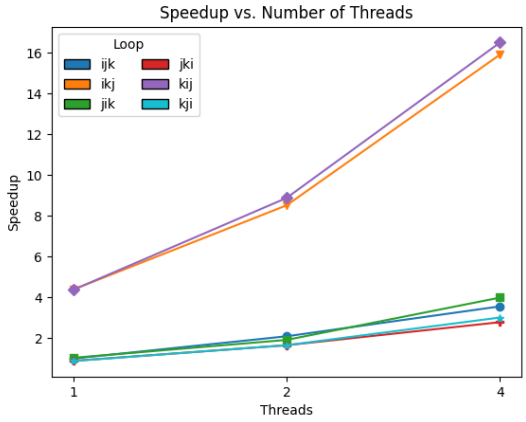

Figure 5: Comparison of speedup at different compiler optimization levels.

Block size: The top ten performers generally have block sizes that fit into the CPU caches. There are two exceptions: the best performer has a block size of 1024 which is larger than the L3 cache, and the 4th best uses no blocking optimization at all. Previous data revealed that for the ikj loop, cache locality was already optimal which minimized the effect of blocking for versions using this loop order. While the effects of blocking were small for this particular loop order, there is a general trend of improvements resulting from appropriately sized blocking. In contrast, the bottom 10 performers all had a block size of 0 (no blocking) or 2048 (too large to fit in the cache).

Compiler Optimization and Vectorization: Every member of the top 10 have both level 3 optimization and vectorization. The bottom 10 are all using compiler flag O0, and have no vectorization.

Parallelism: All of the top 10 are using multiple threads, while only two of the worst performers use more than a single thread and both of these beat the other worst performers in terms of runtime.

Cache usage and L3 data volume: There is a strong correlation between faster runtimes and reduced volumes of data in and out of the L3, indicating that performance is greatly improved when the cache is utilized efficiently. The lowest L3 data volume in the worst 10 performers (559.00 GB) is still more than 15 times as high as the highest data volume in the best performers (35.77 GB).

5 Conclusions

With careful consideration for the architectural properties of the target system, significant improvements can be made to the runtime performance of computationally intensive algorithms such as text analysis and matrix multiplication.

References

[1] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th ed., 2011.

Top 10 Best Performers								
#	OptFlag	NThreads	Loop	Block Size	Vectorized	Runtime (s)	L3 Vol. (GB)	Speedup
1	O3	4	ikj	1024	Y	0.667	35.48	515.11
2	O3	4	ikj	512	Y	0.670	35.72	512.80
3	O3	4	ikj	256	Y	0.707	24.71	486.17
4	O3	4	ikj	0	Y	0.859	35.39	400.05
5	O3	4	ikj	2048	Y	0.886	35.62	387.94
6	O3	4	ikj	128	Y	0.888	3.54	387.03
7	O3	4	ikj	64	Y	0.924	6.01	371.94
8	O3	2	ikj	1024	Y	1.024	35.49	335.44
9	O3	2	ikj	512	Y	1.030	35.77	333.58
10	O3	2	ikj	256	Y	1.100	6.57	312.18
...	...	...	...	...	...	...	...	...
644	O0	1	ijk	0	N	343.525	559.74	1.00
...	...	...	...	...	...	...	...	...

Table 8: Table showing the ten fastest runtimes among all tested variations of the matrix multiplication algorithm. Values in the speedup column indicate runtime improvement relative to the base implementation (highlighted).

Bottom 10 Worst Performers								
#	Opt. Flag	NThreads	Loop	Block Size	Vectorized	Runtime (s)	L3 Vol. (GB)	Speedup
...	...	...	...	...	...	...	...	...
639	O0	2	kji	2048	N	211.661	1,586.94	1.62
640	O0	2	jki	2048	N	214.290	1,578.30	1.60
641	O0	1	jik	0	N	331.566	590.65	1.04
642	O0	1	ijk	2048	N	332.200	559.00	1.03
643	O0	1	jik	2048	N	333.697	593.93	1.03
644	O0	1	ijk	0	N	343.525	559.74	1.00
645	O0	1	kji	2048	N	378.345	1,534.53	0.91
646	O0	1	jki	2048	N	383.713	1,530.07	0.90
647	O0	1	jki	0	N	387.778	1,539.17	0.89
648	O0	1	kji	0	N	390.453	1,542.70	0.88

Table 9: Table showing the ten slowest runtimes among all tested variations of the matrix multiplication algorithm. Values in the speedup column indicate runtime improvement relative to the base implementation (highlighted).