Research Report: Summer 2019

Paul Scanlon and Matt Murdoch

Colorado State University, Undergraduate Research for Dr. Sanjay Rajopadhye with Nirmal Prajapati

Introduction

Over the course of the summer of 2019, we were tasked with optimizing the matrix chain ordering problem, MCO, a sub-problem of the optimal string parenthesization problem, OSP, used in matrix chain multiplication. The triply nested loop structure featured in the dynamic programming implementation invites memory traversal optimization for increased locality and vectorization, and it suggests the attainability of cubic run-time [1].

Pursuing this optimization gives us insight about roofline modeling within the department. Finding the limit of memory boundedness and computational intensity boundedness gives us a direction for optimizing this algorithm on department specific machines. Reaching machine peak is important for advancing roofline analysis.

The memory structure for this problem, given a problem size, matrix chain length N, can be considered a dynamic triangular table with an i and j axis where $i \leq j \leq N$ ($Figure$ 1). Additionally, 'k' minimization operations must be done in each cell of the structure to build the reduction solution, where $i \leq k < j$.

The specific computation performed on each cell is a piece-wise function dependent on the current location in the table given by:

$$M[i,j] = \begin{cases} 0 \quad if \quad i = j \\ \min\limits_{\mathbf{i \leq k < j}} \{M[i,k] + M[k+1,j] + p_{i-1}p_kp_j\} \quad if \quad i < j \end{cases}$$

Where M is the table and p is the array of dimensions given as input.

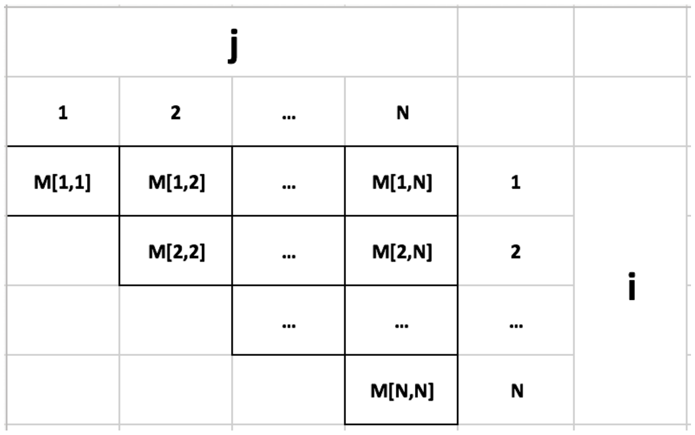


Figure 1: Visualization of Memory Structure

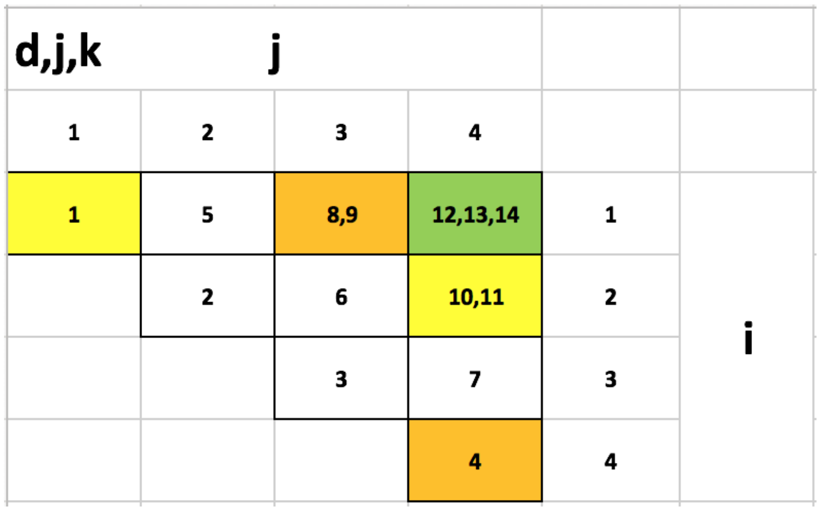
The innermost loop of k can be thought of as a third dimension of the table, allowing one to conceptualize three dimensional navigation of the table's cells. There many different ways to navigate this new $3D$ space. The three dimensions can be traversed in 3! different orders (i, j, k or j, i, k etc.), and further, each dimension can be moved through in two different directions ($+/-$). Additionally, combining the j and i axes into $j - i$ creates a diagonal direction of traversal, d, which replaces the i dimension. Thus in total there are 96 permutations of these loop variables given by:

$$2(i\ or\ d)\ \times\ 2^3(+/-) \times\ 3!\ (Permutations\ of\ Ordering\ Dimensions) = 96$$

All permutations of these variables for memory structure traversal are considered as they may be vectorized differently, support more or less memory locality, and will produce different auto generated C scripts.

Visualization

Because evaluating a cell in the k dimension requires that both $M[i, k]$ and $M[k + 1, j]$ be fully evaluated first (k dimension fully iterated through at these two points), many of these 96 mappings result in illegal orderings. In order to prove the validity of a mapping, a system of visualizing the navigation was created. One visualization that was considered was a triangular pyramid with the $2D$ dynamic memory table acting as the base and the k dimension rising up to a single point above the $[1, N]$ corner. However, this form of visualizing, while accurately describing the third dimension, is difficult to sketch and quickly becomes uselessly cluttered. So, a more compact form of visualizing the $3D$ navigation was created using time stamps on the original $2D$ memory structure visualization above. Below is an example of how the default mapping of d, j, k where $N = 4$ is visualized:

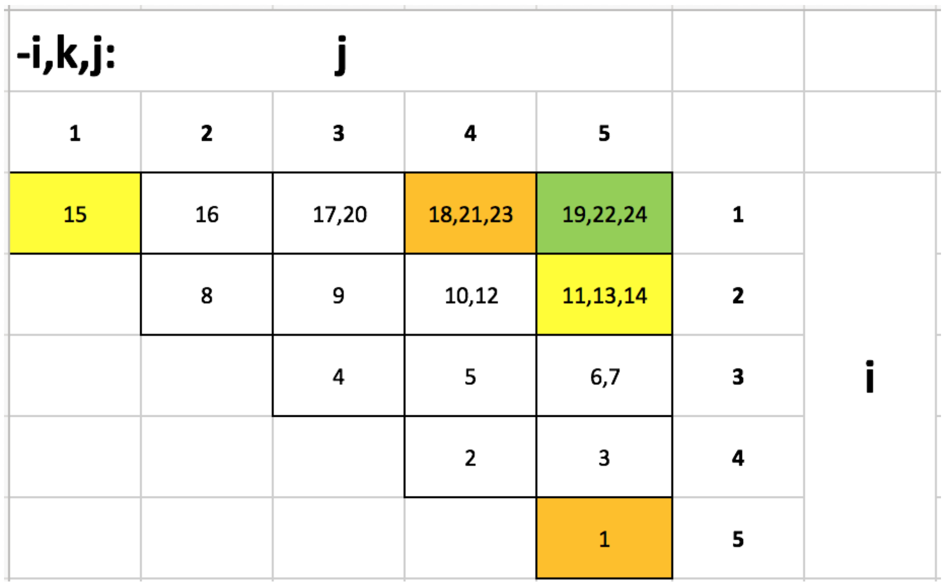


d,j,k		j			
1	2	3	4		
1	5	8,9	12,13,14	1	
	2	6	10,11	2	i
		3	7	3	
			4	4	

Figure 2: Example of the Visualization Constructed for Proving Mappings (d, j, k)

The time stamps represent the order of building the solution for each cell. A cell is complete when it has been visited $j - i$ (size of k dimension at that point) times or populated with zero if along the diagonal. Listing the time steps for this specific mapping shows that whenever a cell is encountered, its k dimension can be fully iterated through as all of the $M[i,k]$ and $M[k+1,j]$ pairs that will be considered are complete with information. For the case of $N = 4$ and mapping d, j, k, the final result requires that cells $(1,3)$, $(4,4)$, $(1,2)$, $(3,4)$, $(2,4)$, and $(1,1)$ be completed for three pairs of $M[i,k]$ and $M[k+1,j]$ in the range of k. This visualization also shows that k can be traversed from $i \longrightarrow j$ or from $j \longrightarrow j$. The squares highlighted yellow and orange show the completed pairs that would be considered first in either direction of k traversal, $(+/-)$. With k as the innermost loop for a diagonal scheme, any direction of traversal is permitted.

A more complicated example of the proof process is shown below for the intricate mapping $-i, k, j$ with $N = 5$. This mapping features k as the middle loop meaning that the range of k for each cell is not fully traversed in one touch - one $M[i,k]$ and $M[k+1,j]$ pair is considered, and then the cell is revisited later after the others along the same axes have been touched. This mapping relies on the diagonal of zeros being evaluated first. This is accomplished through initialization of the loop variables to promote the diagonal's evaluation first and a $-i, k, j$ traversal order elsewhere; this requires tracking the state of each loop level at the displayed time steps and transforming them to support a modified diagonal first time stamp ordering.



-i,k,j:		j				
1	2	3	4	5		
15	16	17,20	18,21,23	19,22,24	1	
	8	9	10,12	11,13,14	2	
		4	5	6,7	3	i
			2	3	4	
				1	5	

Figure 3: Example of the Visualization Constructed for Proving Mappings $(-i, k, j)$

A problem size as small as $N = 4$ can shows all of the trends of a mapping. In this visualization of the mapping $-i, k, j$, the final result is highlighted green, a complete pair is highlighted yellow, and an incomplete pair is highlighted orange. The traversal direction of k will determine which pair is considered first and thus the mapping's success. For this specific scheme, it is only legal to traverse k in the positive direction because the pair that would be considered first with $-k$ has not been completed. It also can be seen that on the next pair considered, $(1,3)$ and $(4,5)$, when the final result is revisited after incrementing k, this pair is now complete for use. This is the strategy for mapping proof - if a direction of iteration can be found to populate the final result cell legally with completed cells below it, the mapping is on track for success. The strategy can be used for finding trends or dependency and memory contiguity.

Illegal Mapping Patterns

There are a few general patterns for observing entire sets of space time mappings to be illegal. One of the most significant is with k or $-k$ permuted as the outermost loop. In these mapping schemes, an entire pass through the memory structure is attempted without completing any $M[i,k]$ and $M[k+1,j]$ pairs. In other words, for cell's with a range of $k > 1$, only one iteration of the minimization is performed over the while memory structure with each k incrimination. This guarantees that when the final result, cell $(1, N)$, is reached on the first pass, no comparison of a $M[i,k]$ and $M[k+1,j]$ pair can be made to build its minimization accurately.

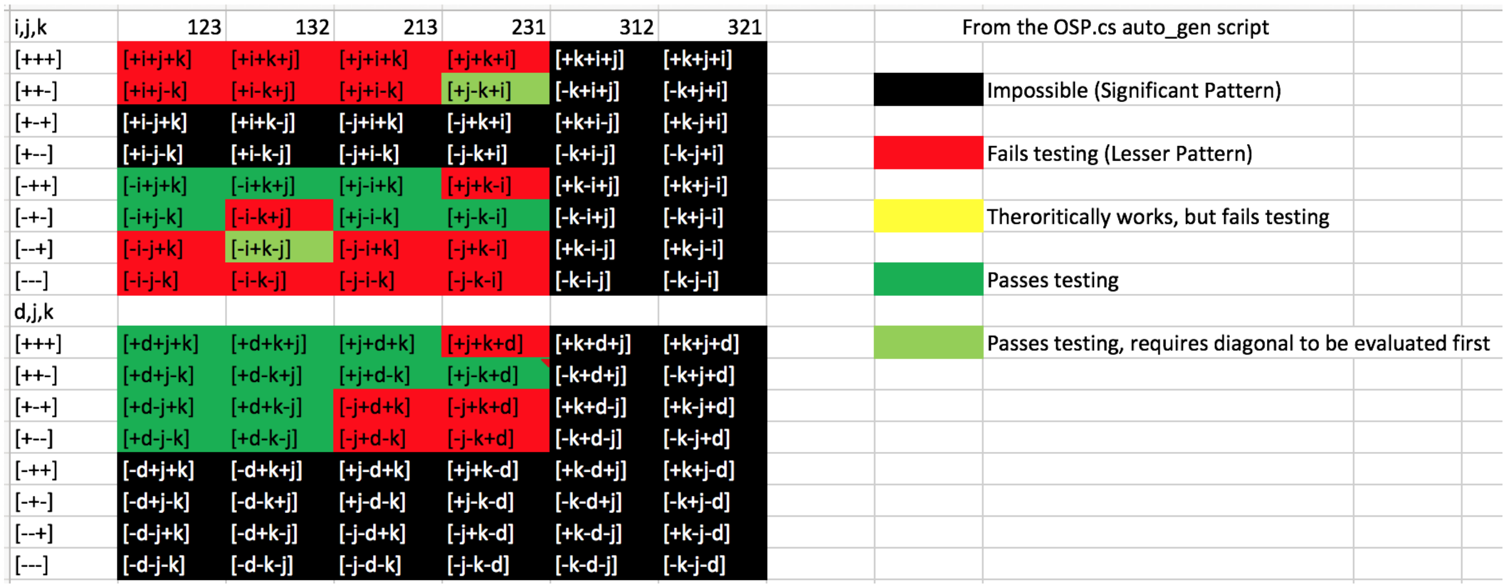
The next clear illegal mapping is with $-d$ permuted to any loop level. When $-d$ is present, the mapping scheme will start step-wise computation at the final result, cell $(1, N)$. None of the minimization has been completed in the cells of $M[i,k]$ and $M[k+1,j]$ for any pairing that the final result could consider, so failure is inevitable.

Some other mappings that begin iteration at the final result are those containing $-j$ and $+i$ together or mappings where $-j$ comes before $+d$ in the permutation. These mappings fail due to a lack of $M[i,k]$ and $M[k+1,j]$ completion for the final result to consider when it is visited first. This is the rule of thumb for the visualization proof process; if the $M[i,k]$ and $M[k+1,j]$ pair is incomplete when considered by the final result, cell $(1, N)$, specific to the direction of iteration of k $(+/-)$, the mapping is invalid. This is the case displayed by the cells highlighted in orange in $Figure$ 3.

Permuting k to the middle loop poses a similar problem of sparse information in the $M[i,k]$ and $M[k+1,j]$ cells. Rather than a clear failure by no complete cells, these mappings depend on the direction of iteration of k like the mapping $-i, k, j$ in $Figure$ 3. The mappings $j, -k, d$ and $j, -k, i$ are legal because information has been completed in cells $M[i,k]$ and $M[k+1,j]$ when k is traversed in the negative direction. If the traversal direction of k is reversed to positive, the $M[i,k]$ and $M[k+1,j]$ cells will not be complete. A similar scenario occurs for the mappings $-i, k, j$ and $-i, k, -j$; if the k traversal direction is reversed within these mappings to negative, then information is pulled from incomplete cells.

Legal Mappings

Using this information about the nature of relationship between the problem and its memory structure, the following truth table for legal mappings was created.



i,j,k	123	132	213	231	312	321
[+++]	🟥 [+i+j+k]	🟥 [+i+k+j]	🟥 [+j+i+k]	🟥 [+j+k+i]	⬛ [+k+i+j]	⬛ [+k+j+i]
[++-]	🟥 [+i+j-k]	🟥 [+i-k+j]	🟥 [+j+i-k]	🟢 [+j-k+i]	⬛ [-k+i+j]	⬛ [-k+j+i]
[+-+]	⬛ [+i-j+k]	⬛ [+i+k-j]	⬛ [-j+i+k]	⬛ [-j+k+i]	⬛ [+k+i-j]	⬛ [+k-j+i]
[+--]	⬛ [+i-j-k]	⬛ [+i-k-j]	⬛ [-j+i-k]	⬛ [-j-k+i]	⬛ [-k+i-j]	⬛ [-k-j+i]
[-++]	🟩 [-i+j+k]	🟩 [-i+k+j]	🟩 [+j-i+k]	🟥 [+j+k-i]	⬛ [+k-i+j]	⬛ [+k+j-i]
[-+-]	🟩 [-i+j-k]	🟥 [-i-k+j]	🟩 [+j-i-k]	🟩 [+j-k-i]	⬛ [-k-i+j]	⬛ [-k+j-i]
[--+]	🟥 [-i-j+k]	🟢 [-i+k-j]	🟥 [-j-i+k]	🟥 [-j+k-i]	⬛ [+k-i-j]	⬛ [+k-j-i]
[---]	🟥 [-i-j-k]	🟥 [-i-k-j]	🟥 [-j-i-k]	🟥 [-j-k-i]	⬛ [-k-i-j]	⬛ [-k-j-i]
d,j,k						
[+++]	🟩 [+d+j+k]	🟩 [+d+k+j]	🟩 [+j+d+k]	🟥 [+j+k+d]	⬛ [+k+d+j]	⬛ [+k+j+d]
[++-]	🟩 [+d+j-k]	🟩 [+d-k+j]	🟩 [+j+d-k]	🟩 [+j-k+d]	⬛ [-k+d+j]	⬛ [-k+j+d]
[+-+]	🟩 [+d-j+k]	🟩 [+d+k-j]	🟥 [-j+d+k]	🟥 [-j+k+d]	⬛ [+k+d-j]	⬛ [+k-j+d]
[+--]	🟩 [+d-j-k]	🟩 [+d-k-j]	🟥 [-j+d-k]	🟥 [-j-k+d]	⬛ [-k+d-j]	⬛ [-k-j+d]
[-++]	⬛ [-d+j+k]	⬛ [-d+k+j]	⬛ [+j-d+k]	⬛ [+j+k-d]	⬛ [+k-d+j]	⬛ [+k+j-d]
[-+-]	⬛ [-d+j-k]	⬛ [-d-k+j]	⬛ [+j-d-k]	⬛ [+j-k-d]	⬛ [-k-d+j]	⬛ [-k+j-d]
[--+]	⬛ [-d-j+k]	⬛ [-d+k-j]	⬛ [-j-d+k]	⬛ [-j+k-d]	⬛ [+k-d-j]	⬛ [+k-j-d]
[---]	⬛ [-d-j-k]	⬛ [-d-k-j]	⬛ [-j-d-k]	⬛ [-j-k-d]	⬛ [-k-d-j]	⬛ [-k-j-d]

From the OSP.cs auto_gen script	
⬛	Impossible (Significant Pattern)
🟥	Fails testing (Lesser Pattern)
🟨	Theroritically works, but fails testing
🟩	Passes testing
🟢	Passes testing, requires diagonal to be evaluated first

Figure 4: Table of Legal Memory Structure Mappings

With the table showing all possible legal traversals that solve the problem correctly finished, the next step is to determine the fastest mapping for an implementation recommendation and a further optimization starting point.

Methodology

To prove which mapping is the fastest, an experiment was constructed. The purpose of the experiment is to collect a range of tightly grouped repeatable timing data points for each mapping. The data must also take the machine's zero calibration into account. We found the zero calibration time of the department machines, specifically Berlin, to be on the order of tenths of milliseconds. This means problem sizes yielding run times greater than a few milliseconds are required to ensure noise is not a significant part of the measurement.

Factors of 1000 for N immediately stuck out as appropriate problem sizes as $N = 1000$ produces run times ranging from tenths to hundredths of seconds depending on the mapping. A problem size of 5000 produces run times ranging from about ten seconds to several minutes depending on the mapping. The final data set selected was five trials on each of nine problem sizes between 1000 and 5000 in steps of 500 for each mapping. Confining the trials to Berlin controls the roofline variables of attainable $GFlops/sec$, computational intensity, and the roofs themselves.

The values of the matrices' dimensions, the p array, were generated randomly to simulate a complicated problem in which it will be unlikely to find an easy parenthesization, and corner cases will be avoided. This was facilitated by the random flag in the makefile which was configured to produce a special executable tailored for this experiment's tests. Correctness verification was done beforehand on each mapping against the $WriteC$ style script generated by the mapping d, j, k. This mapping is verified by the textbook [1]. Additionally, to get the most accurate timing data, the original timer in the wrapper script based on time of day was augmented to system time on the order of clock ticks. We are convinced this timer is more accurate and less prone to interruption scenarios.

The modern Intel C++ Compiler, *icc*, was used to generate the executable scripts for each mapping. This augmentation over *gcc* takes vectorization, fused multiply-add units and may other levels of optimization into account that apply to numerous areas of the roofline model. The use of *icc* also enables the capture of vectorization reports for result analysis.

After the data was collected, outliers were identified and thrown out with a $Z - score$ based protocol of 2% or the two maximum. The data of these trials was found to be within 2% standard deviation, so throwing out two of the most extreme outliers from each trial did not significantly change the result of their average used for graphing and logarithmic slope fitting, but this step guaranteed that the data was tightly grouped.

Results

The averaged timing data was first plotted to determine its shape and order. A cubic order is expected due to the triply nested structure [1]. The exponential shape of the data invites log scale linear fitting to see if any of these mapping are of the cubic order, but already, a significant difference between mappings is evident. The raw timing data for the nine problem sizes between 1000 and 5000 is plotted below for all 23 legal mappings.

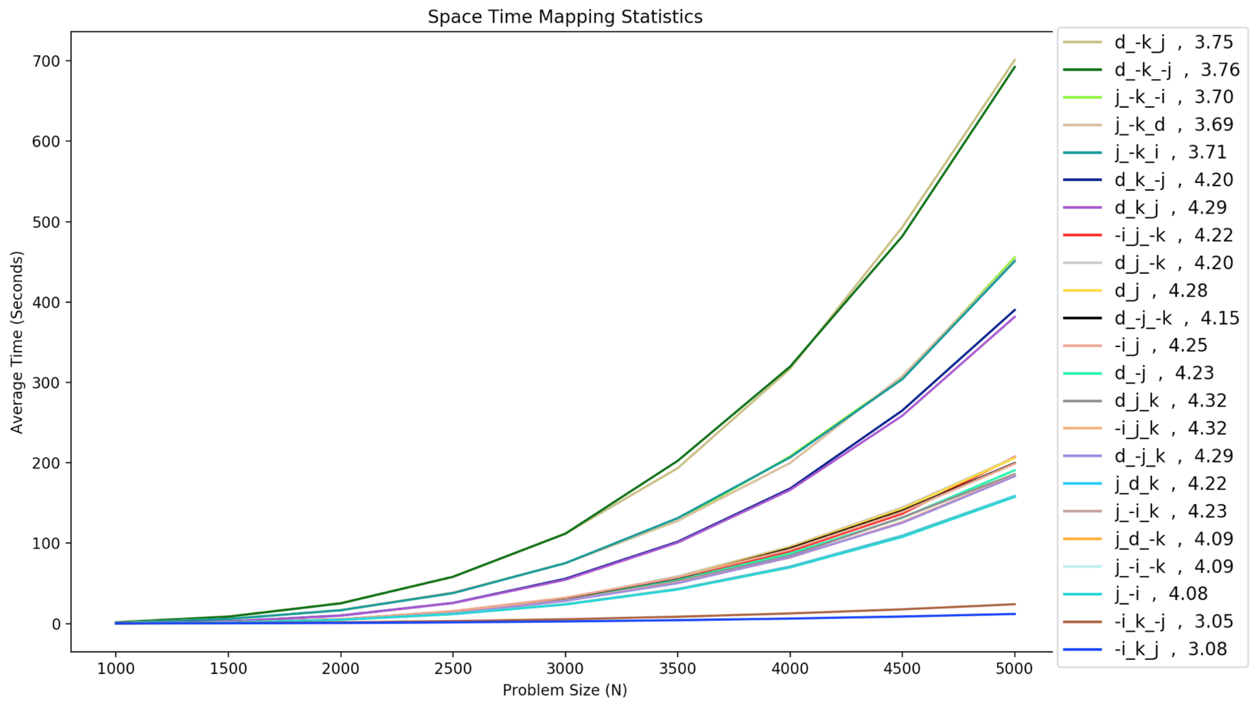


Figure 5: Raw Data Visualization of Run Time for all 23 Legal Mappings

There are about five visible trends or performance. The mappings close to each of the five distinct bands share an extent of vectorization and factors that contribute to their performance bound. Five mappings that represent the major performance trends are shown below.

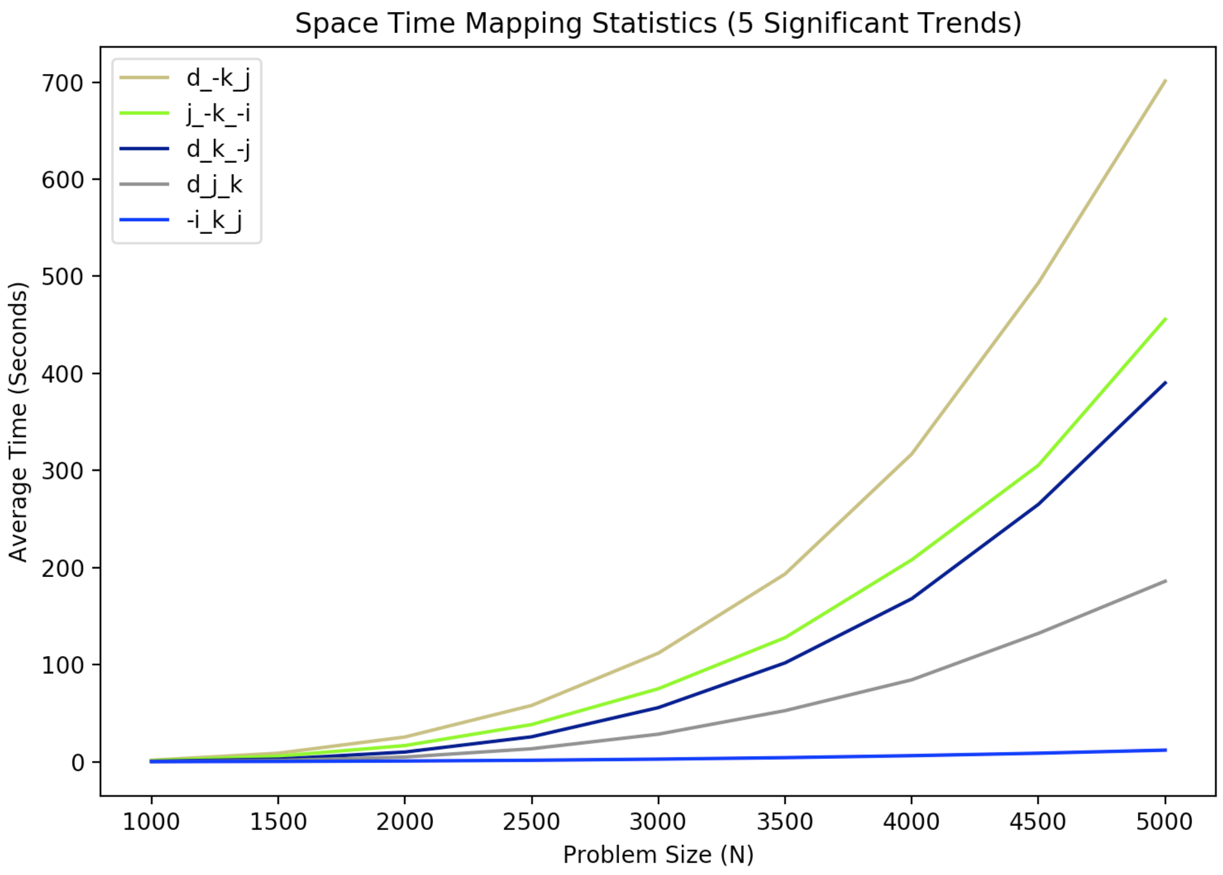


Figure 6: Raw Data Visualization of Run Time for 5 Significant Trends

Logarithmic scaling tells us the exponential order of the raw data via a linearly fitted slope. Steeper slopes and higher intercepts indicate worse performance. The intercept is specific to the problem size considered in this experiment. This simply means that a variable is needed to represent initial overhead within the scope of the experiment that will be shadowed by growing problem size. This point is shown below in a plot of the five trends scaled logarithmically as well as the full set of 23 logarithmically scaled mappings. Although some mappings have a lower run time at the lowest problem size in the experiment, they have a steeper slope indicating that they will surpass their higher starting counter parts as problem size increases. This point is directly visible within the experiment in $Figure$ 8 of all 23 mappings logarithmically scaled where many of the timing slopes cross.

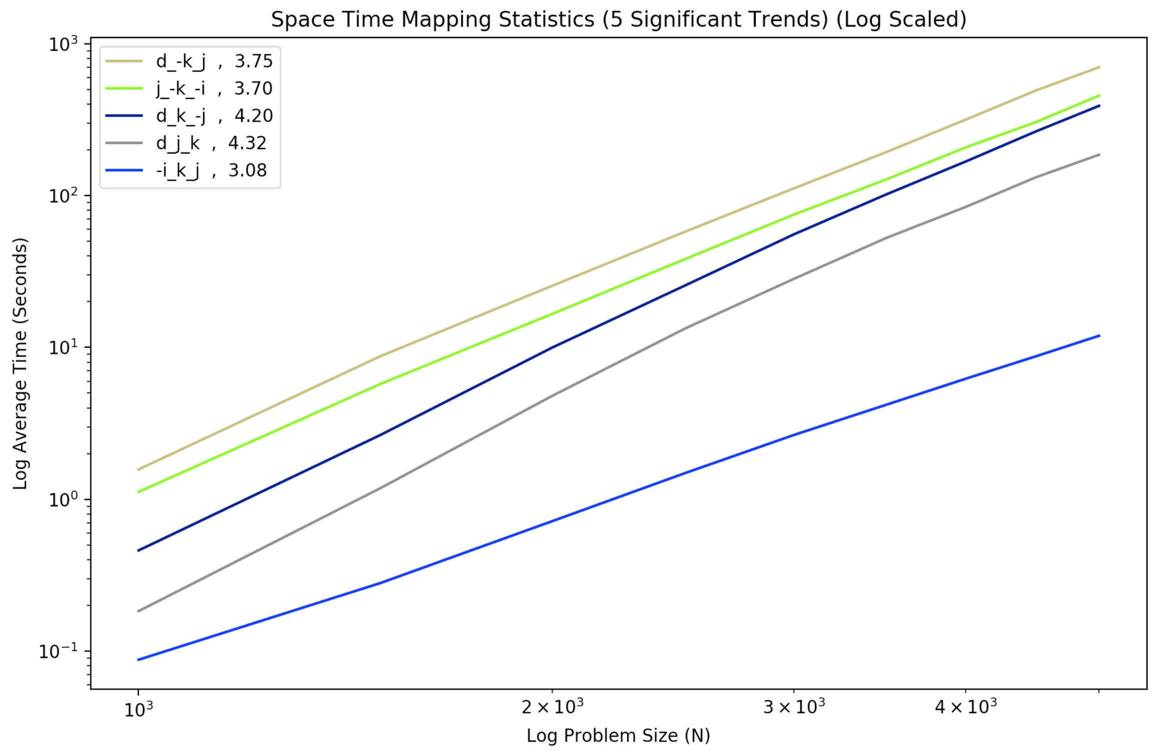


Figure 7: Log Scaled Data Visualization of Run Time for 5 Significant Trends

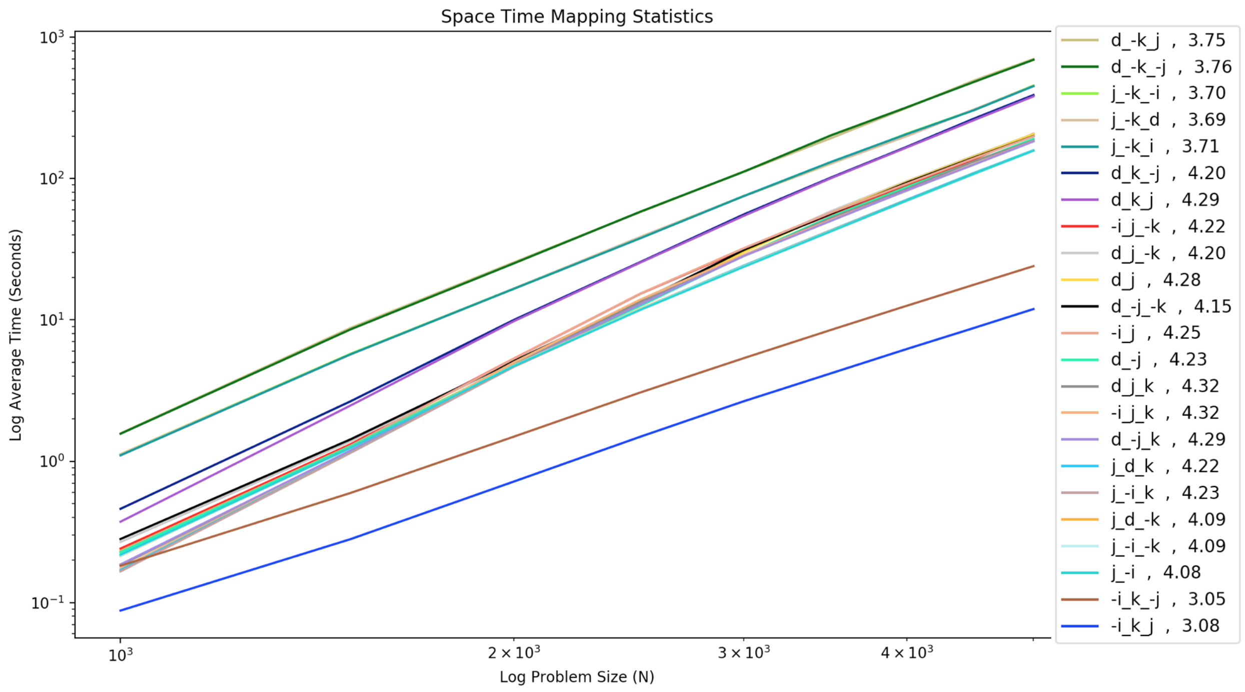
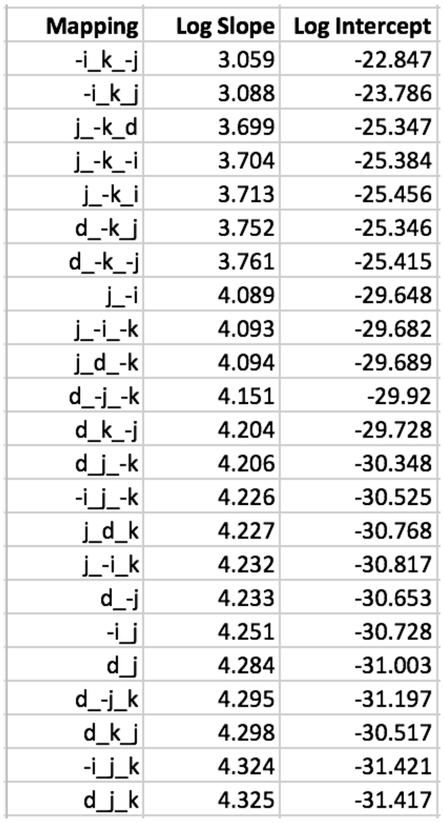


Figure 8: Log Scaled Data Visualization of Run Time for all 23 Legal Mappings

The slope of each logarithmically scaled mapping tells us its exponential order of performance. It is observed that the order of table entries sorted by log slope does not match that of the log scaled graph. This because the table is sorted solely on log slop to portray a result related to arbitrarily large problem sizes which are most relevant to overall performance. The intercept has an effect specific to this set of problem sizes - it can be viewed as baseline overhead that has a diminishing effect as problem size increases. A tabular leader-board sorted by log slope is shown below.



Mapping	Log Slope	Log Intercept
-i_k_-j	3.059	-22.847
-i_k_j	3.088	-23.786
j_-k_d	3.699	-25.347
j_-k_-i	3.704	-25.384
j_-k_i	3.713	-25.456
d_-k_j	3.752	-25.346
d_-k_-j	3.761	-25.415
j_-i	4.089	-29.648
j_-i_-k	4.093	-29.682
j_d_-k	4.094	-29.689
d_-j_-k	4.151	-29.92
d_k_-j	4.204	-29.728
d_j_-k	4.206	-30.348
-i_j_-k	4.226	-30.525
j_d_k	4.227	-30.768
j_-i_k	4.232	-30.817
d_-j	4.233	-30.653
-i_j	4.251	-30.728
d_j	4.284	-31.003
d_-j_k	4.295	-31.197
d_k_j	4.298	-30.517
-i_j_k	4.324	-31.421
d_j_k	4.325	-31.417

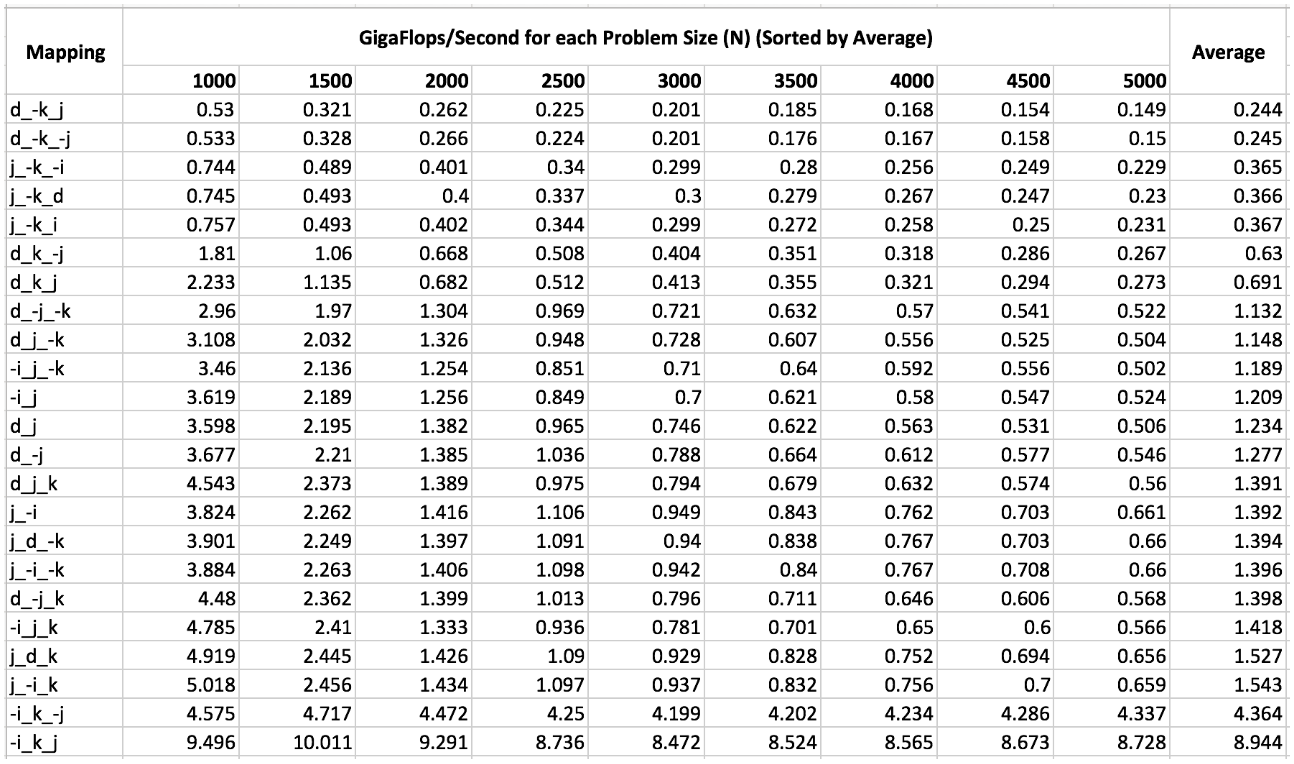
Figure 9: Table of Exponential Orders for each Mapping

It is observed that a cubic order is achieved by two of the mappings, $-i, k, -j$, and $-i, k, j$. The slowest performances are an order 1.3 greater than cubic displayed by the mappings $-i, j, k$ and d, j, k.

Next, the results need to be converted to $GigaFlops/Second$ for the roofline model. The operational intensity of this algorithm is five operations per six memory accesses scaled by the size of the data type. The size of the dynamic memory table is $N^3/6$. This yields the following equation for converting timing data to $GigaFlops/Second$:

$$Operational\ Speed = (5(N^3/6))/time$$

Operational speed is dependent on problem size, N, and a trend is observed that as problem size increases, operational speed decreases. This trend is not followed by all mappings, though - some have a seemingly constant operational speeds independent of problem size. Also, the factors influencing a mapping's performance are heavily influenced by their vectorization report. More vectorization is found to yield higher operational speed in many cases when combined with memory locality.



Mapping	GigaFlops/Second for each Problem Size (N) (Sorted by Average)									Average
	1000	1500	2000	2500	3000	3500	4000	4500	5000	
d_-k_j	0.53	0.321	0.262	0.225	0.201	0.185	0.168	0.154	0.149	0.244
d_-k_-j	0.533	0.328	0.266	0.224	0.201	0.176	0.167	0.158	0.15	0.245
j_-k_-i	0.744	0.489	0.401	0.34	0.299	0.28	0.256	0.249	0.229	0.365
j_-k_d	0.745	0.493	0.4	0.337	0.3	0.279	0.267	0.247	0.23	0.366
j_-k_i	0.757	0.493	0.402	0.344	0.299	0.272	0.258	0.25	0.231	0.367
d_k_-j	1.81	1.06	0.668	0.508	0.404	0.351	0.318	0.286	0.267	0.63
d_k_j	2.233	1.135	0.682	0.512	0.413	0.355	0.321	0.294	0.273	0.691
d_-j_-k	2.96	1.97	1.304	0.969	0.721	0.632	0.57	0.541	0.522	1.132
d_j_-k	3.108	2.032	1.326	0.948	0.728	0.607	0.556	0.525	0.504	1.148
-i_j_-k	3.46	2.136	1.254	0.851	0.71	0.64	0.592	0.556	0.502	1.189
-i_j	3.619	2.189	1.256	0.849	0.7	0.621	0.58	0.547	0.524	1.209
d_j	3.598	2.195	1.382	0.965	0.746	0.622	0.563	0.531	0.506	1.234
d_-j	3.677	2.21	1.385	1.036	0.788	0.664	0.612	0.577	0.546	1.277
d_j_k	4.543	2.373	1.389	0.975	0.794	0.679	0.632	0.574	0.56	1.391
j_-i	3.824	2.262	1.416	1.106	0.949	0.843	0.762	0.703	0.661	1.392
j_d_-k	3.901	2.249	1.397	1.091	0.94	0.838	0.767	0.703	0.66	1.394
j_-i_-k	3.884	2.263	1.406	1.098	0.942	0.84	0.767	0.708	0.66	1.396
d_-j_k	4.48	2.362	1.399	1.013	0.796	0.711	0.646	0.606	0.568	1.398
-i_j_k	4.785	2.41	1.333	0.936	0.781	0.701	0.65	0.6	0.566	1.418
j_d_k	4.919	2.445	1.426	1.09	0.929	0.828	0.752	0.694	0.656	1.527
j_-i_k	5.018	2.456	1.434	1.097	0.937	0.832	0.756	0.7	0.659	1.543
-i_k_-j	4.575	4.717	4.472	4.25	4.199	4.202	4.234	4.286	4.337	4.364
-i_k_j	9.496	10.011	9.291	8.736	8.472	8.524	8.565	8.673	8.728	8.944

Figure 10: Timing Data Converted to Operational Speed Sorted by Average

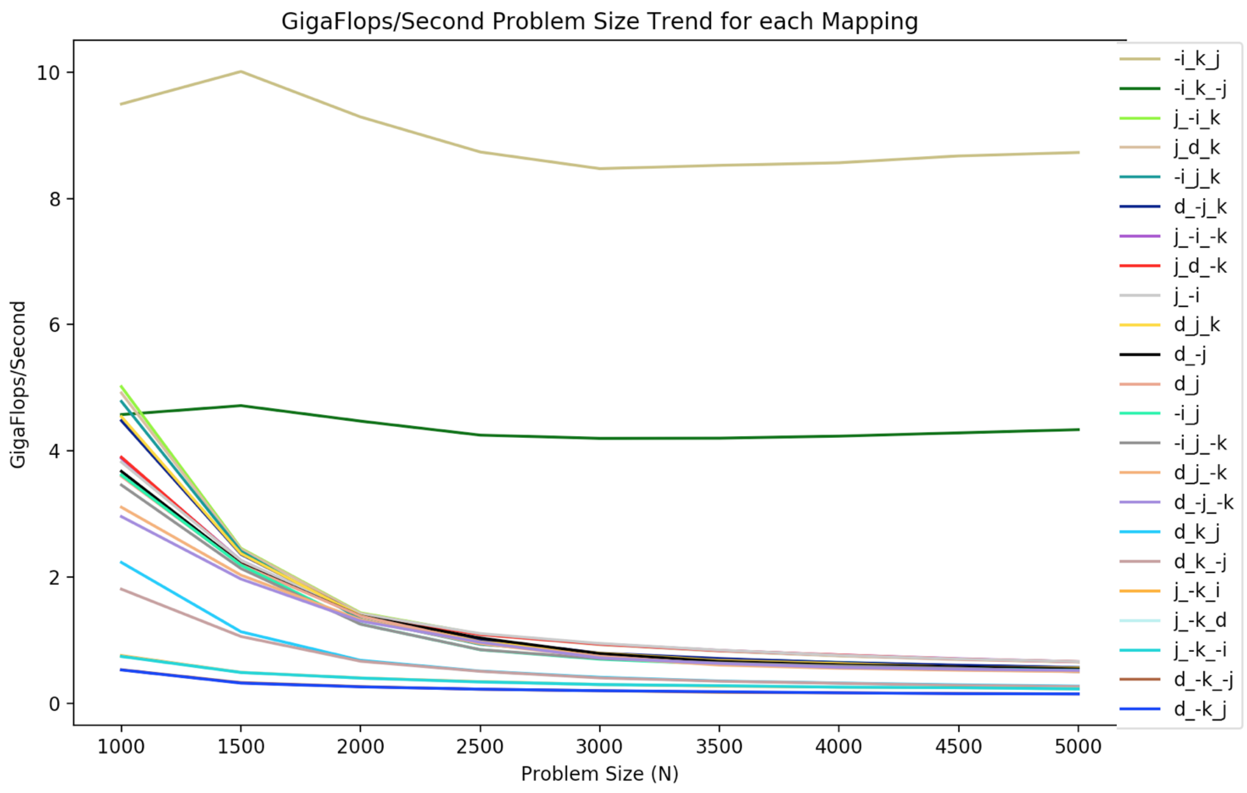


Figure 11: Problem Size Trends for each Mapping's Operational Speed

Vectorization Reports

The Intel C++ Compiler displays its success in vectorization through flags that build a report. Vectorization of the innermost loop has the most significant impact as it will be utilized most often. For the slowest mapping, $d, -k, j$, it is revealed that none of the loops were vectorized because the compiler deemed it inefficient or impossible due to vector dependencies. The critical innermost loop of the computation was not vectorized due to vector dependence and lack of unit-stride loading. This begins to explain its poor performance compared to the fastest mapping, $-i, k, j$.

All efficient loops were vectorized in the fastest mapping. In the computational body of $-i, k, j$, the table's initialization steps were separated from the iterative computation to support vectorization of length four offering a potential factor 3.65 speed up. The first computational loop only has a depth of two, but is vectorized with a length of four offering a potential 2.43 factor speed up. The critical innermost loop was also vectorized with a length of four offering a potential factor 2.43 speed up. Vectorization accounts for a magnitude of more than eight of potential speed up over the other mappings for $-i, k, j$.

The auto generated code for each mapping has a vast difference in loop structure and complexity. Of the 19 mappings, seven support vectorization. The mappings $-i, j, k$, $d, -j, k$, d, j, k, $j, -i, k$, and j, d, k all received one vectorization of the innermost loop offering a potential factor 1.03 speed up. They have the commonality of k $(+)$ as the innermost loop. The mapping d, k, j received two vectorizations of doubly nested computational loops and one vectorization of the innermost loop offering potential factors 1.0, 1.0 and 1.03 speed ups respectively. It has a commonality with the fastest mapping of k $(+)$ as the middle loop, but not all mappings with k $(+)$ as the middle loop were vectorized.

Vectorized loops have the similar traits of aligned access and unit-stride loading. Loops that are not vectorized may have vector dependencies, non unit-stride loading, failure to be recognized for auto-vectorization, or simply be deemed inefficient to vectorize by the compiler. The compiler will often reject vectorization if the potential speed up factor is less than one. $SIMD$ instructions can augment the extent of vectorization when the compiler rejects it for inefficiency.

Conclusion

From the tables showing $GFlops/Sec$ and the corresponding graph of timing data conversion, it is observed that the mapping $-i, k, j$ operates closest to machine peak, likely escaping memeory boundedness and hitting a flat roof in the roofline model. This is confirmed by the log scaled linear fit to the timing data collected for this mapping. Thus, the highest roofline starting point for optimization of this algorithm under what is percieved a flat computational roof is with the mapping $-i, k, j$.

The success of this result comes from having j as the innermost loop combined with the aligned assess of k. Traversing through rows of j allows memory to be accessed contiguously the way it is understood by the compiler.

The target machine, Berlin, features an $x86\_64$ processor with a $Linux$ operating system. The CPU is an $Intel\ Xeon$ operating at $3.40GHz$ with eight cores. Its maximum memory bandwidth is $25.6GB/s$. Online benchmark reports reveal this processor to be able to achieve roughly ten $GFlops/sec$ for single core use and upwards of 24 $GFlops/sec$ for multicore use [2][3].

As mentioned previously, the operational intensity of this algorithm is $5/(6 \times size)$ where the size refers to the size of the data type [1]. The generated code uses four byte integers yielding a base operational intensity of 5/24 at first glance. The mappings have an effect on the number of reads and writes occurring in the innermost loop, though, as some can be outsources to higher loop levels if independent of the innermost.

For example, when k is permuted to the innermost loop in any mapping, the write of $M[i, j]$, the read of p_i and the read of p_j can be outsourced leaving only three accesses in the innermost loop instead of six. Additionally, because two elements of the two step multiplication are outsourced as a single variable, two multiply operations are reduced to one leaving four operations in the innermost loop instead of one. If k is permuted to the middle, a similar scenario occurs with the accesses of the innermost variable.

When d is present in the mapping with a variation of j as the innermost loop, only the variables based solely on k can be outsourced. This allows for only one access reduction and no reduction in operations.

With this information, each mapping can be plotted as a kernel in a theoretical roofline model. The fastest mapping lies at 4/12, or 1/3, $Flops/Byte$ on the $x - axis$ and rises 8.94 $GFlops/Sec$ on the $y - axis$. The slowest mapping lies at 5/20, or 1/4, $Flops/Byte$ on the $x - axis$ and rises only 0.24 $GFlops/Sec$ on the $y - axis$. Their positions alone with with other mappings are displayed below.

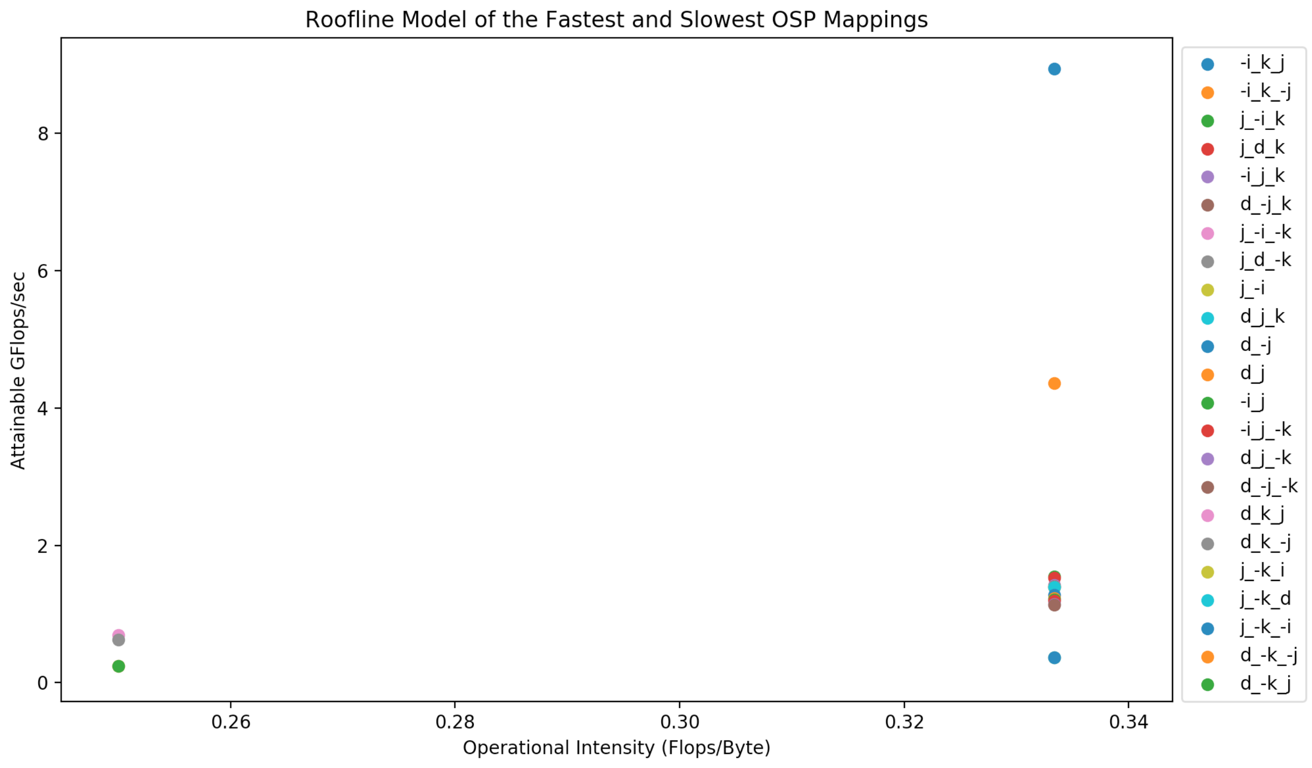


Figure 12: Roofline Model for All mappings as Kernel Points

The slowest mapping is barely distinguishable in terms of *Attainable GFlops/sec* from the fastest mapping. From these positions, it is expected that the slowest mapping is memory bound, and the fastest mapping is computationally bound. This is because their positions suggest a memory bound slope for the sloped part of the roof in the conventional roofline model. These assertions cannot be proven without additional graphical roofline data - specifically exact memory and computational bounds offered by the *Intel Advisor Toolkit* for the target machine which will be available soon. It may be that the roofline is steep, and the bounds are as described above. Or, it may be that the roofline is shallow, and both mappings are, in fact, memory bound. The next step is to apply memory maps and parallelization.

Bibliography

[1] Thomas Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. "Introduction to Algorithms, Third Edition". *2009*
`https://ms.sapientia.ro/~kasa/Algorithms_3rd.pdf`

[2] Intel. "Intel® Xeon® Processor E3-1231 v3". *2015*
`https://ark.intel.com/content/www/us/en/ark/products/80910/intel-xeon-processor-e3-1231-v3-8m-cache-3-40-ghz.html`

[3] Primate Labs. "14-08-12 Intel Xeon E3-1231 v3 Generic". *2012*
`https://browser.geekbench.com/geekbench3/706660`