UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

ANATOMY OF HIGH PERFORMANCE PARALLEL 1-DIMENSIONAL STENCIL

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

By MARK CASTLE

Norman, Oklahoma

2026

ANATOMY OF HIGH PERFORMANCE PARALLEL
1-DIMENSIONAL STENCIL

A THESIS APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY THE COMMITTEE CONSISTING OF

Dr. Richard Veras, Chair
Dr. Sridhar Radhakrishnan
Dr. Ghulam Quadri

Acknowledgments

The computing for this project was performed at the OU Supercomputing Center for Education & Research (OSCER) at the University of Oklahoma (OU).

This material is based upon work supported by the National Science Foundation under Grant No. 2440646.

Table of Contents

List of Figures

List of Tables

Abstract

Stencils, or structured mesh computations, are a fundamental class of computations characterized by convolutional neural networks, finite difference, image processing, computer vision, cellular automata, geophysics, and climate modeling. Reliably extracting performance from these operations across modern hardware is sufficiently challenging that they are included in standard compiler benchmarks. This is in part because the design space for these problems is very large, and only a few designs obtain high performance for a given target. For compilers, it is difficult to explore this design space, so they use heuristics that are still unable to find optimal solutions. Hand tuning stencils is also a challenge as a unique kernel would be required for each problem of interest and each hardware target of interest. Whether the kernel is hand tuned or generated by a compiler, the problem is that many constraints need to be simultaneously satisfied, such as maximizing instruction level parallelism, managing register pressure, utilizing vector instructions, while maximizing floating point throughput. This work provides a model-based approach for high performance 1-Dimensional stencils, along with a BLIS-like algorithm providing a layered interface. We validate our model by demonstrating very high performance across a variety of CPU microarchitectures.

Chapter 1

Introduction

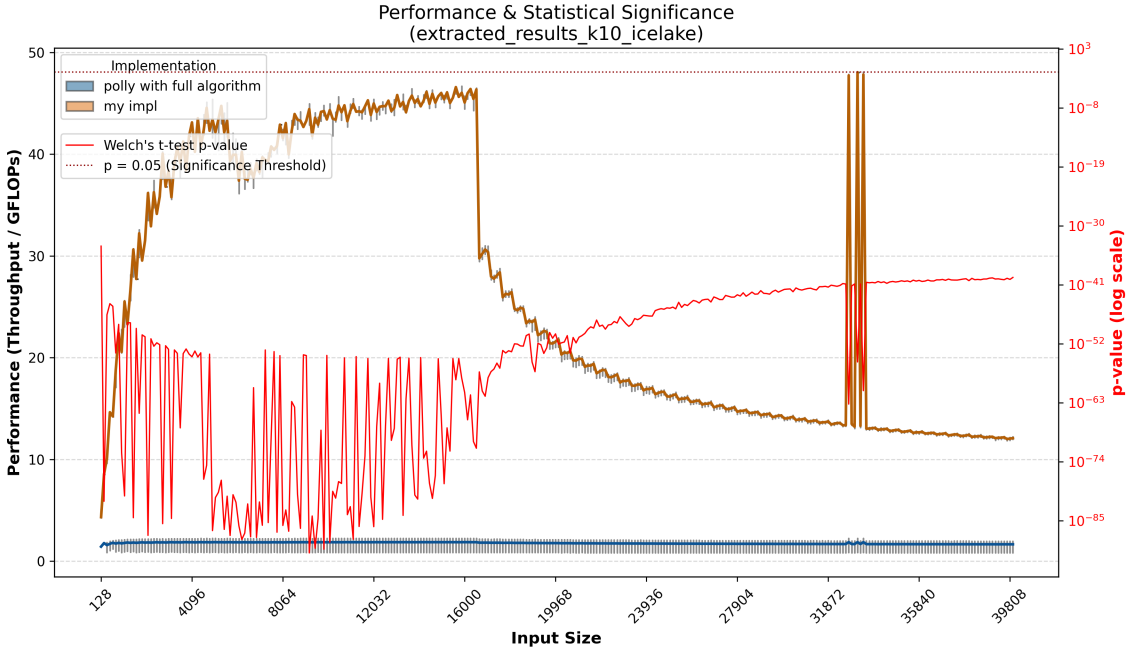

Figure 1.1: Some motivating results. Run on an Intel(R) Xeon(R) Gold 6338 and compiled with GCC. This is the Icelake microarchitecture. This plot has the Welch's t-test p-value between my implementation and Polly's with 30 samples for each implementation. The p-value is far below 0.05 which means we reject the null hypothesis meaning there is a statistical difference between the GFLOP/s achieved by each implementation.

Stencils are an important problem to optimize because they are very prevalent in

many scientific computing applications. These include partial differential equation solvers, convolutional neural networks, finite-difference methods, cellular automata, and the list goes on. Many things that people use every day will see benefits such as weather simulations, phone calls, and streaming video.

The following is an example of a real-world heat transfer problem tranformed to a stencil operation. To find the heat transfer along a rod we use the equation $\frac{du(x,t)}{dt} = C * \frac{d^2u(x,t)}{dx^2}$. When discretizing this formula it turns into the following code block:

```
for m = 1 to final m
  for i = 1 to n-1
    U(i,m+1) = z * U(i-1,m)
               + (1-2*z) * U(i,m)
               + z * U(i+1,m)
  end for
end for
```

[1]

This is our stencil operation. It has a k (width) of 3 and an offset of -1. More specifically this is a stencil operation for many time steps because of the outer m loop.

These problems all have something in common, they are continuous in nature, but have to made discrete for the computer. We want to optimize these problems to allow for more data points to be processed, so that the problem becomes closer to being continuous again. Because optimizing stencils allows them to look closer to being continuous this translates to faster or higher fidelity results using the same hardware as before. It also allows for lower end hardware to achieve the same results.

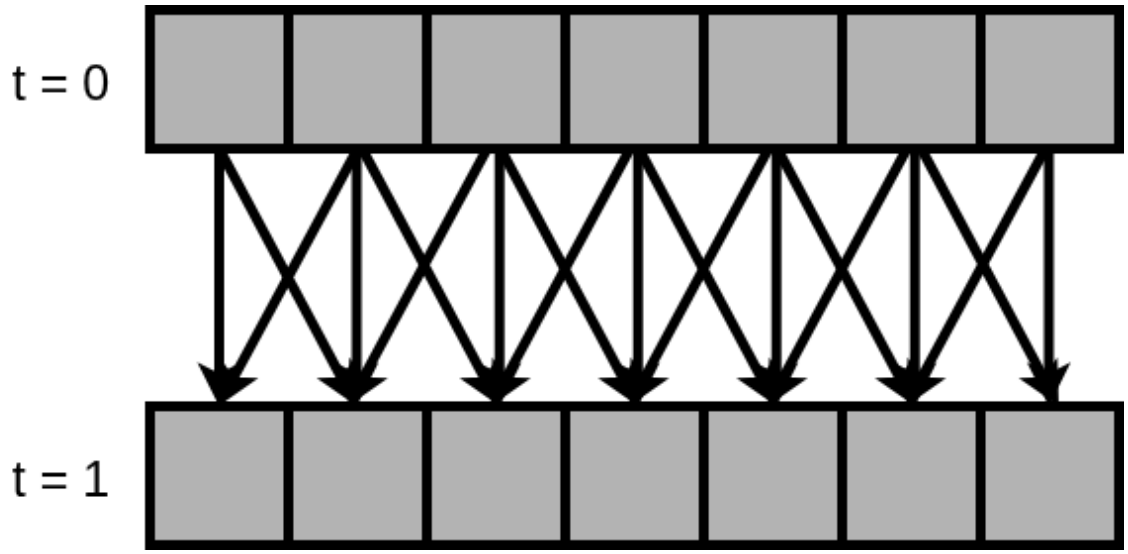

Figure 1.2: This diagram is the calculation of a single time step for the Jacobi 1D Stencil.

Stencils are a challenging operation to optimize. They inheritly have lots of dependent and overlapping work. These are bad characteristics to be able to run optimially on any hardware. We want to maximize the number of independent fused multiply adds (fmas), but we have dualing pressures of the latency of the fma units and the number of registers we have to hide that latency. For the class of stencil operations, we can rearrange the order of computation to maximize instruction level parallelism and minimize register pressure to acheive a high-performance kernel.

There are two current gaps that our solution solves. The hand tuned implementations require lots of effort since stencils are a broad class of problems and each solution has to be tuned to the specific hardware it will run on. The compiler tuned implementations lack high performance. The optimization step for compilers loses lots of information about the actual problem, so it struggles to make many noticeable optimizations.

The main contributions of this work are the following:

- An algorithm for high performance stencil computations on CPUs that performs the operation in an order that maximizes ILP and minimizes the amount of stalls because of dependent work.
- Performance Model to determine the fraction of peak flops that the algorithm can obtain.
- Code generator that can create an optimal kernel based on the problem and hardware constrainsts.
- Validation of performance model across different architectures.
- Sensitivity analysis.
- Comparison against the state-of-the-art polyhedral compiler.
- Comparison against Intel MKL convolution implementation.
- A layered interface for interacting and building with these performant kernels.
- A set of high performance implementations of stencil kernels for various different stencil widths.

Chapter 2

Literature

For the literature that was reviewed there are two main goals; To find the strategies for implementating efficient code and see hwo they are applied and to see the other approaches to stencils to see where they fell short and what can be improved upon.

2.1 Strategies for Implementing Efficient Code

There are a few very important concepts that these past papers dicuss that all go into creating fast stencils. These concpets are data layout transformation, associative reordering, SIMD, and loop nesting.

2.1.1 BLIS

[2]

This paper shows the methods that can be used to amortize the cost of memory operations. It also sets up a generic interface that can be used on a wide range of hardware because lots of the important code can be generated. This interface is a multilayered API, this allows users to access the kernels instead of just the main

functions. Because of this multilayered API, users can implement functionality that is not present in the main functionality. The benefit of this setup is that for a specific platform, developers only need to implement a small set of simple kernels that all the more complicated kernels can be cast in terms of. To achieve hiding the cost of the memory operations, they use tiling and repacking of the data to use it more efficiently.

2.1.1.1 Reimplementing BLIS

I recreated BLIS as a way to learn about a few of the tricks to extract high performance from a linear algebra problem. A few of these tricks are:

- Thinking about the problem in a different way to maximize instruction level parallelism.In the case of matrix multiplication it is thinking about the base case as an outer product instead of an inner product.
- Repacking the data so that way all of your memory accesses are sequential. This makes your algorithm much more cache friendly which reduces the overall time spent retrieving data from main memory.
- Vectorizing the innermost part of the kernel. This allows you to operate on much more data with a single instruction.
- Blocking your data. This works together with repacking the data, but instead focuses on getting block sizes that make sure your data will completely fit in a specific layer of cache.
- Picking a kernel size so that you maximize the use of all the availible hardware registers.

As shown later in the performance plot, these tricks are what is needed to obtain near peak performance. These are the same kind of tricks that will be used in the fast stencil implementation.

This is the microkernel of the BLIS algorithm. Each iteration of the loop inside performs an outer product of two vectors and sums the outer product with the current C submatrix.

```
void microkernel_16x8_crow(float* A_pack, float* B_pack, float* C,
                           int rs_c, int cs_c, int KC){
    int cldx = rs_c;
    __m512 c0 = _mm512_loadu_ps(C + 0*cldx);
    ...
    __m512 c7 = _mm512_loadu_ps(C + 7*cldx);




    for(int k = 0; k < KC; k++){
        __m512 b = _mm512_loadu_ps(B_pack + 16*k);

        __m512 a0 = _mm512_set1_ps(A_pack[k*8 + 0]);
        ...
        __m512 a7 = _mm512_set1_ps(A_pack[k*8 + 7]);

        c0 = _mm512_fmadd_ps(b, a0, c0);
        ...
        c7 = _mm512_fmadd_ps(b, a7, c7);
```

```
    }

    _mm512_storeu_ps(C + 0*cldx, c0);
    ...
    _mm512_storeu_ps(C + 7*cldx, c7);
}
```

This is the kernel of the BLIS algorithm. It iterates through the packed A and B matrices to ge tthe correct size chunks for the micro-kernel. The strides create an MRxNR submatrix of C for the microkernel to update.

```
for(int jr = 0; jr < NC; jr+=NR){ // stride within packed B
    for(int ir = 0; ir < MC; ir+=MR){ // stride within packed A
        microkernel_16x8_crow(&A_pack[ir*KC],
                              &B_pack[jr*KC],
                              &C[(i+ir)*rs_c + (j + jr)*cs_c],
                              rs_c, cs_c, KC);

    }
}
```

This is the C, A blocking loop. These matrices are blocked by grouping rows together. The size of the A submatrix (MC rows, KC cols) that will be repacked should fit in L2 cache.

```
for(int i = 0; i < M; i+=MC){
```

```
    pack_A(A, rs_a, cs_a, A_pack, KC, MC, MR, p, i);
    kernel()
}
```

This is the A, B blocking loop. These matrices are blocked by grouping rows together. The size of the B submatrix (KC rows, NC cols) that will be repacked should fit in L3 cache.

```
for(int p = 0; p < K; p+=KC){
    pack_B(B, rs_b, cs_b, B_pack, NC, KC, NR, j, p);
    CA_Blocking()
}
```

This is theoutmost loop of the BLIS algorithm. It is the C, B blocking loop. This blocks C and B by groups of columns.

```
for(int j = 0; j < N; j+=NC){
    AB_Blocking()
}
```

These are the packing routines for the A and B matrices. They allow the micro-kernel to sequentially grab the data that it needs.

```
void pack_B(float* B, int rs, int cs, float* B_pack,
            int NC, int KC, int NR, int j, int p){
    int row_offset = p;
    int col_offset = j;
```

```
    for(int i = 0; i < NC*KC; i++){
        int block = i / (NR*KC);
        int row   = (i / NR) % KC + row_offset;
        int col   = i % NR + col_offset + block*NR;

        B_pack[i] = B[row*rs + col*cs];
    }
}

void pack_A(float* A, int rs, int cs, float* A_pack,
            int KC, int MC, int MR, int p, int i){
    int row_offset = i;
    int col_offset = p;
    for(int i = 0; i < KC*MC; i++){
        int block = i / (MR*KC);
        int row   = i % MR;
        int col   = (i / MR) % KC;
        A_pack[i] = A[(row_offset + block*MR + row)*rs +
            (col_offset + col)*cs];
    }
}
```

For a full code example, it is in the appendix section A.2.

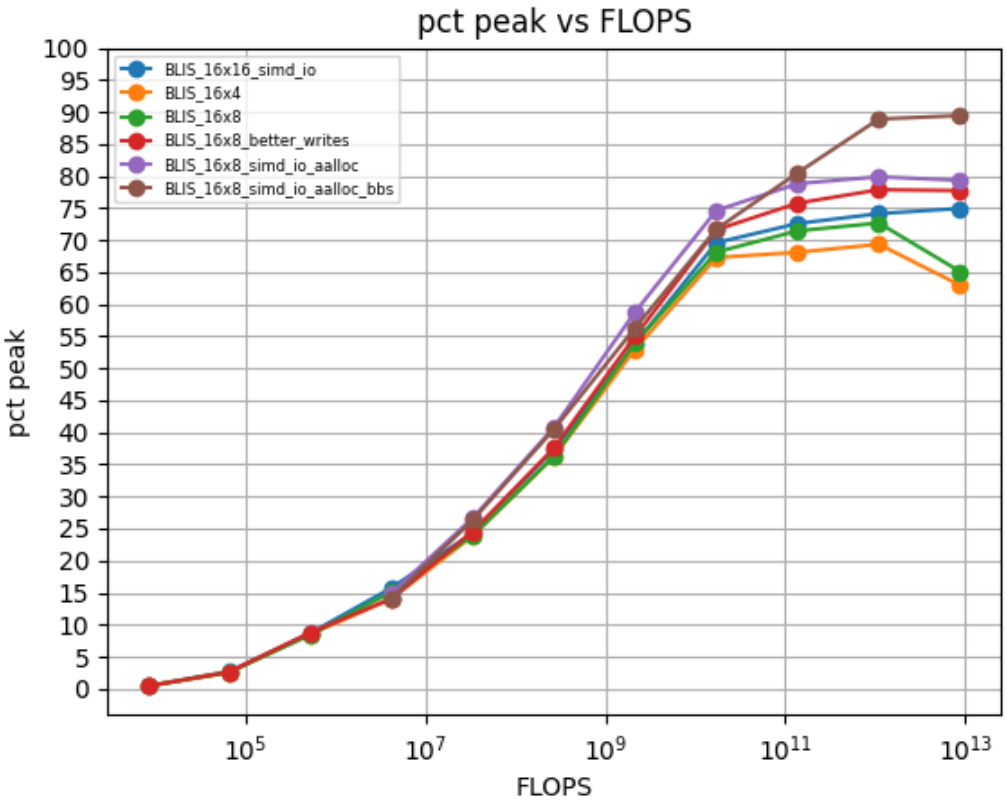

Figure 2.1: Run on an AMD Ryzen 9 7900X and compiled with GCC. This shows the performance of various microkernels for the BLIS matrix multiplication algorithm. The variants are various dimensions of the microkernel and other memory optimizations to try to approach peak GFLOP/s.

2.1.2 A Systematic Approach for Obtaining Performance on Matrix - Like Operations Over Structured and Real - World Data

[3]

This work addresses how to achieve high performance for matrix like computations when the data is not ideal. It explains that you want to reorganize data into forms that can take advantage of dense linear algebra optimizations.

There are two important parts to efficient implementations. The first is efficient memory access pattern (partitioning, algorithm nesting, and data layout transformations). The second is a kernel that is tuned to the system's microarchitecture which can matain the rate of computation that the memory access rate allows.

2.1.3 Data Layout Transformation for Stencil Computations on Short-Vector SIMD Architectures

[4]

This paper addresses the issue of non-fma port pressure by using a data layout transformation which eliminates the need for later shuffles and permutations. The aim of the data layout transformation is to make neighboring stencil elements align with SIMD lanes. This cost of this transformation can be amortized by the work of more complex stencils or by incorporating the transformation in earlier stages of the computation. This work only focuses on the data layout transformation, we use this data layout transform to aid in maximizing SIMD-level kernel performance.

2.1.4 A Framework for Enhancing Data Reuse via Associative Reordering

[5]

This paper introduces the use of associative reordering to reduce register pressure. The goal of this is to increase data reuse by reordering the computations within loop nests while keeping the operatin mathematically the same. It uses the mathematically properties of associativity and commutativity to do the reordering of operations. We use associative reordering to maximize instruction level parallelism for SIMD-level kernels on data layout transformed data.

2.1.5 A Top-Down Method for Performance Analysis and Counters Architecture

[6]

This paper introduces a method for identifying the true bottlenecks in out-of-order processors. The tool creates a tree and assigns weights to each of the nodes, this way you can see the area that the bottleneck is occuring. These weights are determined by the events emitted by the CPU's Performance Monitoring Unit (PMU). They divide issues into frontend bound, bad speculation, retiring, and backend bound. Once you determine which of these is the bottleneck you can traverse down the tree into the problem node's children and repeat. The main benefit here is that instead of inspecting many low level metrics where it would be hard to tell where the bottleneck is, the analysis tool focuses on certain bottleneck categories.

2.2 Other approaches to Fast Stencils

There are many other attempts to make fast stencils. All of them use some of the same tricks, but are missing important pieces. Which is why they fall short because all of the tricks are required to make fast stencils. The big trick that none of them employ is the data layout transformation.

2.2.1 When polyhedral transformations meet SIMD code generation

[7]

This paper relies on search to find the best candidate algorithms. This is done

through polyhedral transformations that restructure programs to try and extract small regions of code that can map cleanly into SIMD instructions. Their use of shuffles and permutations to arrange the data for SIMD operations puts to much pressure on the non-fma ports which puts the bottleneck on those instead of useful computation.

2.2.2 A Stencil Compiler for Short-Vector SIMD Architectures

[8]

This paper combines the ideas of the data layout transformation and code generation to improve the performance of SIMD-level kernels. They do this through a domain specific language (DSL) and a compiler specific for stencil computations. This paper relies on the search of a large space of solutions opposed to our analytical performace models to determine the best implementation.

2.2.3 Polyhedral parallel code generation for CUDA

[9]

This paper uses the tricks of tiling, loop rearranging, loop fusion/fission, etc. to improve the performance of the problems in the PolyBench suite specifically for NVIDIA GPUs. This is done through a source to source compiler that takes C code and translates it to CUDA code for the GPU. It doesn't show any significant improvement for the stencil operations except for jacobi-2d, but even then it is still minimal compare to the other types of problems that the compiler optimizes. This shows that polyhedral compilers are not suited to optimize stencil problems.

2.2.4 A Survey of General-purpose Polyhedral Compilers

[9]

Polyhedral compilers consist of three main steps:

1. "Raising a static control part of a program into its polyhedral representation"

- changing loops into some polyhedron that represents the iterations and their exexcution date

2. "Finding an optimizing schedule for this problem"

- loop fusion/fission, tiling, ...

3. "Lowering the resulting polyhedral representation into code"

The results section shows that for stencil computations most polyhedral compilers show little to no speedup.

2.2.5 High Performance Zero-Memory Overhead Direct Convolutions

[10]

This approach for fast stencils forcuses on N-d stencil operations whereas ours focuses on 1-D. There are still very important commonalities in both. They create a model to try and maximize instruction level parallelism and register usage. They also do loop reordering to achieve instruction level parallelism. One strategy that this approach does not use is a data layout transformation.

The motivation for this approach is to be used in a convolution kernel within a DNN. They are trying to compute direct convolutions without using any auxiliary memory.

Chapter 3

Theory

There are three pieces that determine the best way to create the stencil code for any computing device. These are hardware paramters, the algorithm, and the performance model.

The algorithm makes the overall structure of the code and how to move through data. The hardware determines the size of the microkernel. The performance model tells us if a given configuration is able to reach high peformance.

3.1 Hardware

The hardware that we are targeting has some amount of functional units we care about and in our case these are the fused multiply add (fma), load, and store units.

These units have a few parameters that are platform specific that will help determine the shape of our algorithm.

- Latency, how many cycles it takes for the result of the operation to be ready. This can also be thought of as the wait time.

- Throughput, how many operations are completed per cycle.
- Reciprocal throughput, the number of cycles between invocations of the unit.

If you can maximize the amount of independant work that is happening, the rate of the fma matters much more than the latency. This is because if the work is all indpendent, you can constantly be queuing up more fmas. So once you have gone latency number of cycles, the startup time, you will be queuing an fma and extracting the result of an fma at the rate of the fma unit.

The latency of the loads and stores only really matter if you do not have enough fmas to hide the loads and stores that are happening. This happens with the k (width) of the stencil is small because there is not much computation to be done for each corresponding load. When we have this situation the problem is memory bound and the bottleneck has shifted away from useful computation.

3.2 Algorithm

The 1D stencil is an operation on an input vector x to produce an output vector y. To compute each y_i, you take the weighted sum of the next $0, ..., k-1$ elements of x, $x_i, ...x_{i+k-1}$. You can visualize calculating a single y_i with the image below.

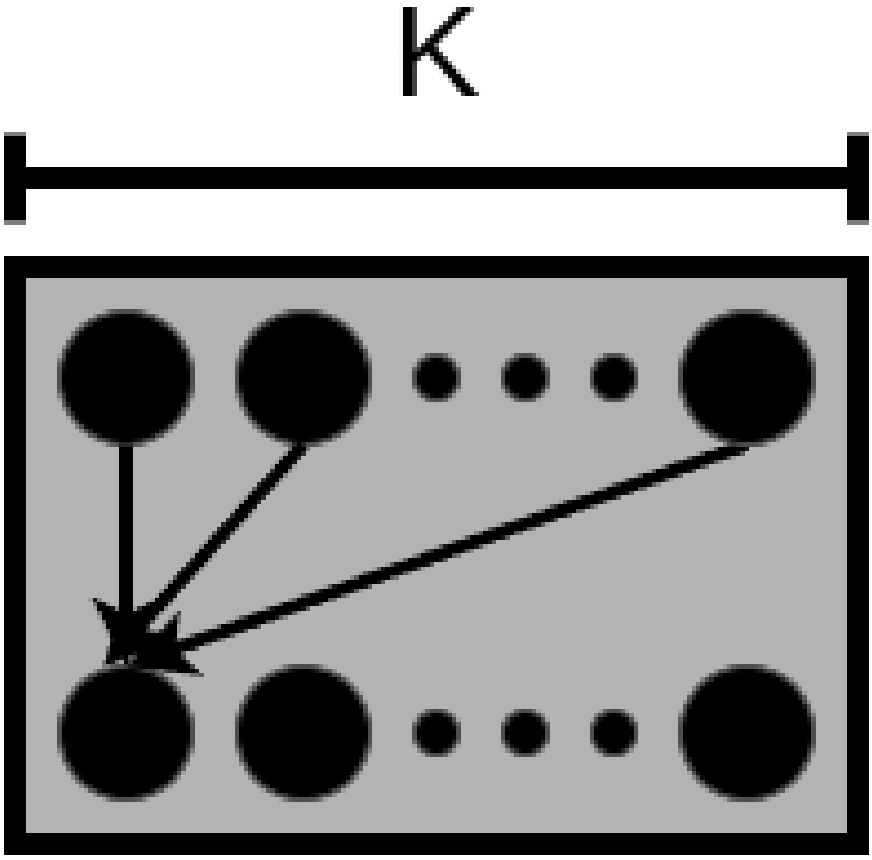

Figure 3.1: This is the reduce block for the baseline of a stencil operation. Many input elements are accumulated to a single output element.

However, this way of performing a stencil limits the performace optimizations we can make. You instead want to think about broadcasting (fma) an input index to many output indices. This is changing a many to one approach to a one to many approach. This is shown below.

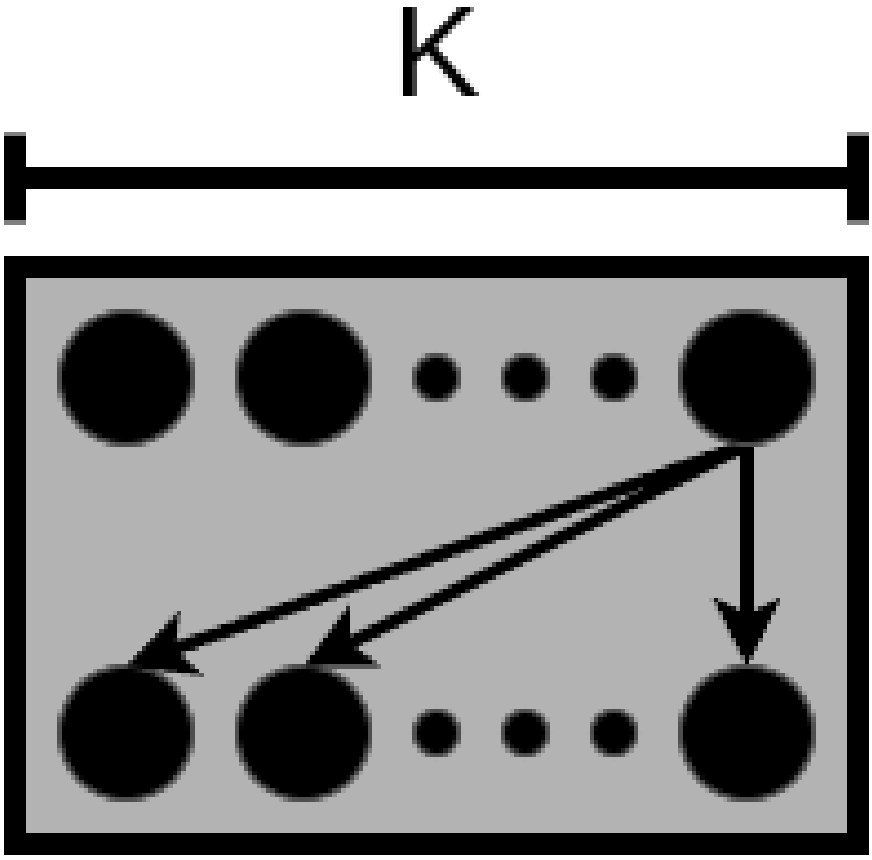

Figure 3.2: This is the broadcast block which is the smallest component of our kernel implementation. A single input element is broadcasted to many output elements.

This helps with two things:

1) Load an input once and completely use it, never having to read an input value from memory more than once

2) Have non overlaping writes to memory that a reduction approach causes

Below is an exmaple of the dependency chain for a reduction. For each of these reductions it takes $k * \lambda$ cycles because you have to wait for each fma.

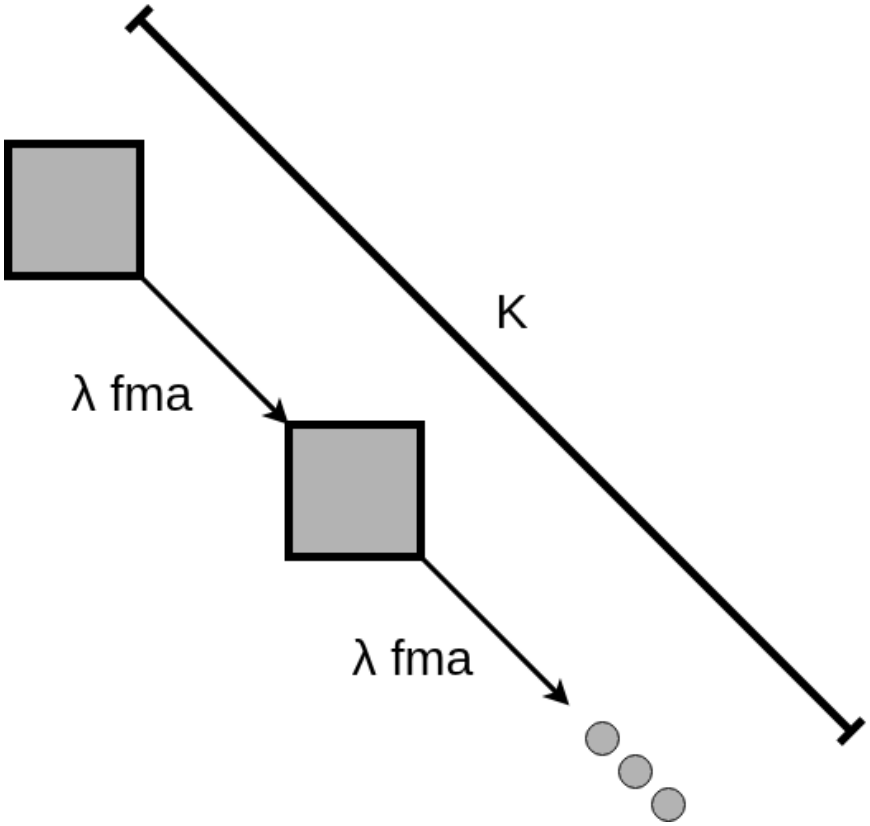

Figure 3.3: This is the FMA dependency chain for the reduction scheme. For a single redction block we have to wait inbetween each FMA because they are all writing to the same output element.

Below is an exmaple of the dependency chain for a broadcast. For each of these broadcasts it takes λ cycles because you don't have to wait for each fma.

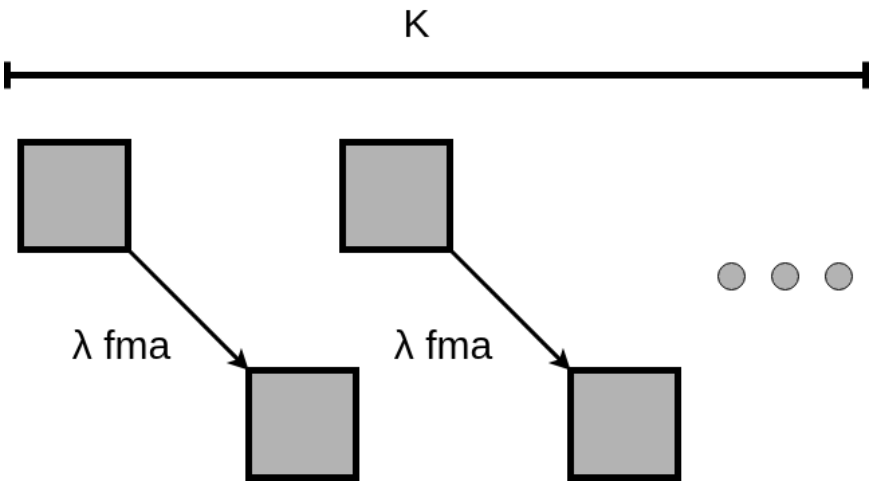

Figure 3.4: This is the FMA dependency chain for the broadcast scheme. For a single broadcast block we only need to wait for the latency of a single fma since each fma can happen in parallel.

Now we need a way to move through the input and keep these nice charactistics. You combine mu broadcast blocks together to form a broadcast codelet. This means you read every kth input value. No overlap. This is shown below.

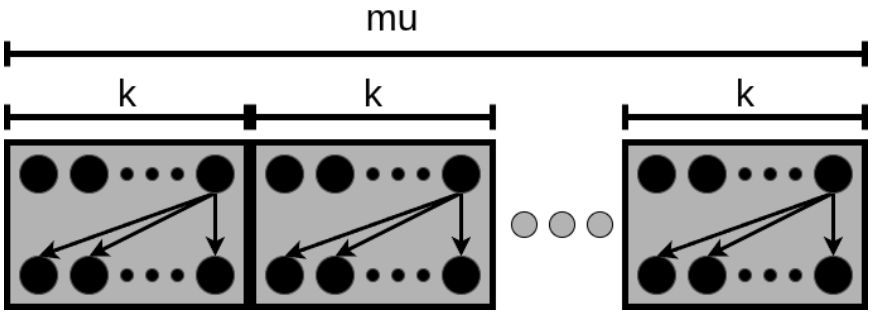

Figure 3.5: This is a broadcast codelet. It is formed by concatenating many broadcast blocks together. This keeps the nice feature of having non-overlapping computation.

This leaves many gaps since we skipped lots of input elements. So, we perform another broadcast codelet, but shifted over by one. We perform k broadcast codelets within a single microkernel iteration as shown below.

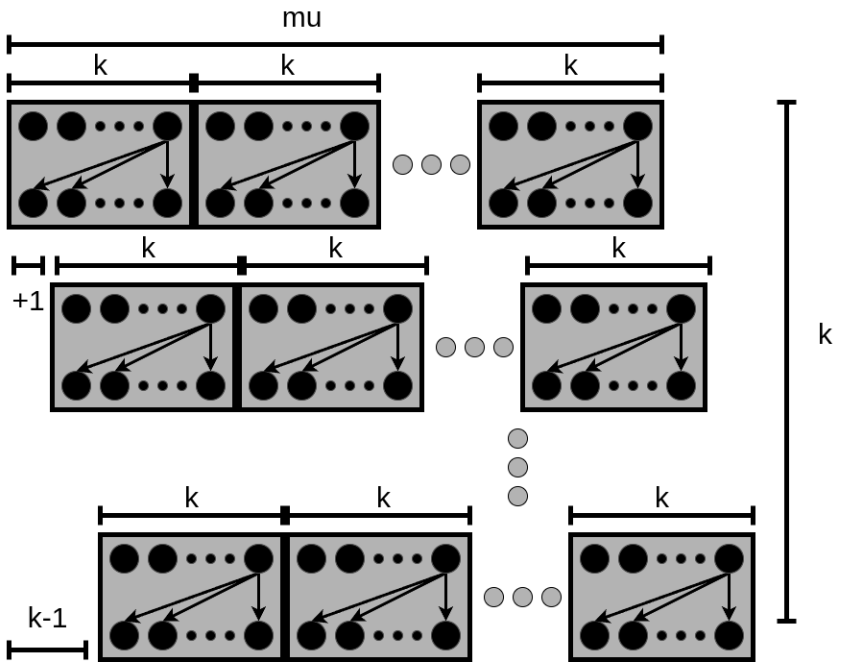

Figure 3.6: This is the micro kernel. It is formed by concatenating many broadcast codelets together, but with an offset. This way we don't skip any elements.

Each kernel iteration $mu * k + k - 1$ y_is are updated. The first k-1 come paritally complete by the previous kernel iteration. After a kernel iteration the first $mu * k$ y_is are written back out to memory and are not needed again. The last $k - 1$ y_is are used for the next iteration. The kernel has as many iterations as it needs to traverse the whole input. This is shown below.

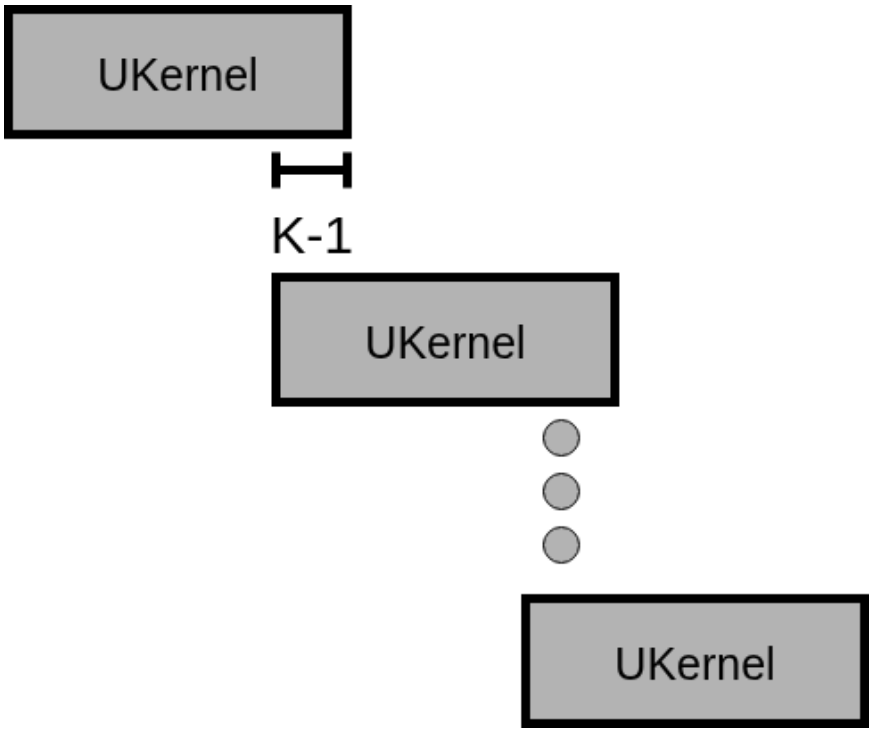

Figure 3.7: This is the kernel. It is formaed by concatenating many microkernels together, but leaving an overlap. This overlap is needed to finish the computation that the previous microkernel started.

All of this assumes that the input size is a multiple of $mu * k$. If not, it is handled in the fringe cases before and after the kernel. This fringe handler also makes sure that we can avoid modulus in the code by advancing the vector pointer enough so that we can't have wrap around.

3.3 Model

There are 3 constraints that determine the best shape of the kernel for any given hardware. The ratio of fused multiply add to memory operations. The ratio of the rate to latency of the fused multiply add units. The number of availible registers.

3.3.1 FMA to Memory Operations

$\frac{\#fma-in-\mu-kernel}{\#mem-in-\mu-kernel} >= \frac{r_{fma}}{r_{mem}}$

$\frac{mu*k^2}{3*mu*k} = \frac{k}{3} >= \frac{r_{fma}}{r_{mem}}$

This assumes that we are always accumulating to the output. Each fma needs 3 reads, read input, read output, perform the fma, write to output. In some μ-kernels this can be improved if we know that y is zero at the start of the μ-kernel. This constrainst ensures that for each memory operation, we can do a lot of useful computation. This ratio is improved by making k larger.

3.3.2 Rate to Latency Ratio

$(mu * k)/(r\_fma) >= (\lambda\_fma)$

but really you want

$(mu * k)/(r\_fma) >> (\lambda\_fma)$

This ensures that for the majority of the micro kernel there are the maximum amount of fmas queued. There is a startup and cooldown penalty of $\lambda/rate - 1$ until you either fill up or empty the queue.

3.3.3 Registers

$num\_reg \leq 1 + k + mu * k + k - 1$

The hardware will have a certain number of registers and we want to use as many of those as possible. We need 1 for the input, k for the weights, mu*k + k-1 for the

outputs. If we break this constraint, it will result in a lot of spilling and filling, so we have extra, unneeded memory operations.

3.3.4 Fraction of Peak

Working within these constraints we can calculate the fraction of peak we expect.

$fop = \frac{\#fma}{r_{fma}} / max(\frac{\#fma}{r_{fma}}, \frac{\#mem}{r_{mem}}, \frac{\#int}{r_{int}})$

This comes with some assumptions

1) There are little to no stalls (dependencies)

2) We have high instruction level parallelism

3) There is sufficient work (long steady state) to amortize the startup and cleanup of the fma queue

For some problems it is not possible to get 100% of peak in this formula because this assumes that the fma will be the bottleneck. If there is no way to make the fma your bottleneck then a different formula is needed to find your peak.

3.3.5 Throughput and Latencies for many different μ-archs

The following tables contain the throughput (Instructions per cycle) and latencies (cycles until complete) for fma, load, and store units across many different μ-architectures and both avx2 and avx512 (when applicable). These values are the driving factors of the model that help determine the optimal shape of the algorithm.

[11] [12] [13] [14]

uarch	Zen2	Zen3	Zen4	Haswell	Broadwell
fma latency (avx-512)	N/A	N/A	4	N/A	N/A
fma throughput (avx-512)	N/A	N/A	1	N/A	N/A
fma latency (avx-256)	5	4	4	5	4
fma throughput (avx-256)	2	2	2	2	2
load latency (avx-512)	N/A	N/A	11	N/A	N/A
load throughput (avx-512)	N/A	N/A	1	N/A	N/A
load latency (avx-256)	8	9	9	7	7
load throughput (avx-256)	2	2	2	2	2
store latency (avx-512)	N/A	N/A	11	N/A	N/A
store throughput (avx-512)	N/A	N/A	0.5	N/A	N/A
store latency (avx-256)	9	10	11	9	9
store throughput (avx-256)	1	1–2 (?)	1–2 (?)	1	1

Table 3.1: Latency/throughput across selected uarches (group 1).

uarch	Cascade Lake	Ice Lake	Sapphire Rapids
fma latency (avx-512)	5	4	4
fma throughput (avx-512)	2	1	2
fma latency (avx-256)	4	4	N/A
fma throughput (avx-256)	2	2	N/A
load latency (avx-512)	8	8	N/A
load throughput (avx-512)	2	2	1.51
load latency (avx-256)	8	8	N/A
load throughput (avx-256)	2	2	N/A
store latency (avx-512)	10	11	N/A
store throughput (avx-512)	1	1	1
store latency (avx-256)	10	11	N/A
store throughput (avx-256)	1	2	N/A

Table 3.2: Latency/throughput across selected uarches (group 2).

uarch	Emerald Rapids	Skylake	SkylakeX
fma latency (avx-512)	4	5	5
fma throughput (avx-512)	2	1	2
fma latency (avx-256)	4	5	N/A
fma throughput (avx-256)	2	2	N/A
load latency (avx-512)	8	N/A	8
load throughput (avx-512)	2	N/A	2
load latency (avx-256)	8	8	8
load throughput (avx-256)	3	2	2
store latency (avx-512)	12	N/A	10
store throughput (avx-512)	1	N/A	1
store latency (avx-256)	12	10	10
store throughput (avx-256)	2	1	1

Table 3.3: Latency/throughput across selected uarches (group 3).

3.3.6 Using These Features to Determine Fraction of Peak Performance

Using the various latencies and throughputs of the fma, load, and store units we can determine what fraction of peak a specific kernel will obtain. Currently the way this is done assumes that we are in the compute bound region of a roofline plot which is an issue because lots of smaller k values are in the memory bound region of the roofline plot. So some of the 1.0 fraction of peak should actually be less because it is impossible for them to hit compute fraction of peak. A new formula is needed to find the fraction of peak when the problem is memory bound. When a fraction of peak value is N/A it means that the ukernel has too few fma operations to hide the latency of the fma unit.

For more table examples, see appendix section A.7.

Table 3.4: Icelake AVX2 FOP k = 1

k	μ	#reg	ilp cyc	rker	fop
1	1	3	0.5	1.0	N/A
1	2	4	1.0	2.0	N/A
1	3	5	1.5	3.0	N/A
1	4	6	2.0	4.0	N/A
1	5	7	2.5	5.0	N/A

Table 3.5: Icelake AVX2 FOP k = 2

k	μ	#reg	ilp cyc	rker	fop
2	1	6	1.0	2.0	N/A
2	2	8	2.0	4.0	N/A
2	3	10	3.0	6.0	N/A
2	4	12	4.0	8.0	N/A
2.0	5.0	14.0	5.0	10.0	1.0

Table 3.6: Icelake AVX2 FOP k = 3

k	μ	#reg	ilp cyc	rker	fop
3	1	9	1.5	4.5	N/A
3	2	12	3.0	9.0	N/A
3.0	3.0	15.0	4.5	13.5	1.0
3.0	4.0	18.0	6.0	18.0	1.0
3.0	5.0	21.0	7.5	22.5	1.0

Table 3.7: Icelake AVX2 FOP k = 5

k	μ	#reg	ilp cyc	rker	fop
5	1	15	2.5	12.5	N/A
5.0	2.0	20.0	5.0	25.0	1.0
5.0	3.0	25.0	7.5	37.5	1.0
5.0	4.0	30.0	10.0	50.0	1.0
5.0	5.0	35.0	12.5	62.5	1.0

Table 3.8: Icelake AVX2 FOP k = 10

k	μ	#reg	ilp cyc	rker	fop
10.0	1.0	30.0	5.0	50.0	1.0
10.0	2.0	40.0	10.0	100.0	1.0
10.0	3.0	50.0	15.0	150.0	1.0
10.0	4.0	60.0	20.0	200.0	1.0
10.0	5.0	70.0	25.0	250.0	1.0

Table 3.9: Icelake AVX2 FOP k = 15

k	μ	#reg	ilp cyc	rker	fop
15.0	1.0	45.0	7.5	112.5	1.0
15.0	2.0	60.0	15.0	225.0	1.0
15.0	3.0	75.0	22.5	337.5	1.0
15.0	4.0	90.0	30.0	450.0	1.0
15.0	5.0	105.0	37.5	562.5	1.0

3.4 Multithreading

To make these kernels multithreaded you have to add a couple extra loops to chunk up the input, so each thread has its own section to work on. This cannot be done with a single for loop though. Because of the nature of this operation, each μ-kernel has a bit of overlap with the previous and next iteration of the μ-kernel. To fix this we break the problem into multiple phases. Each thread will go through PH phases. Each phase is an offset for the thread block.

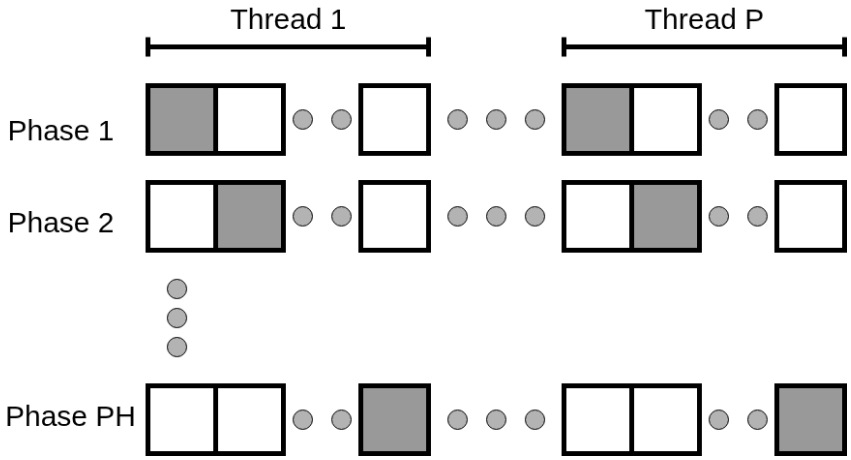

Figure 3.8: This is the multithreaded phase diagram. The idea here is that each thread will have work that is non-overlapping with other threads. Each grey block is the section that a thread is working on. The threads use the original single core kernel.

Chapter 4

Mechanism

The mechanisms are what actually make the theory work. They also handle edge cases that the theory purposfully ignores.

4.1 DLT

Our goal of this data layout transformation is to arrange the data such that vectors can be loaded instead of just single elements. This way we can take advantage of vector intrinsics to operate on many floats at once.

We do this by treating the 1D input as 2D where the number of rows is simd_len and the number of columns is $\frac{input\_len}{simd\_len}$
Then you perform a transpose of this matrix to make each row simd_len long. This way each element is just a vector load of the corresponding row.
There are also $k-1$ ghost rows added to the end of the matrix. These handle the cases where two elements were adjacent, but after the DLT they are no longer adjancent. This adds a tiny bit of overhead to the DLT, but the advantage of enabling simd outweighs this bit of overhead.

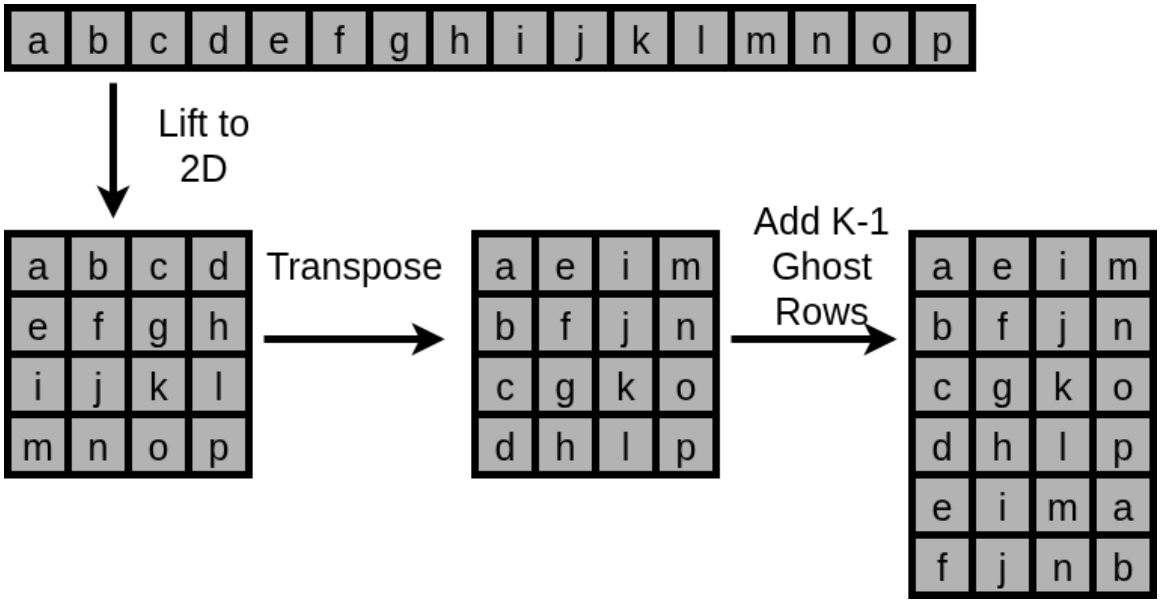

Figure 4.1: This is the data layout transformation for our 1D stencil kernels. This is done to allow for our algorithm to operate on vectors instead of scalars. We accomplish this by raising the input to 2D, performing a transpose, padding with some extra rows, then lowering back to 1D.

Because of the way that we arrange the data, the algorithm to compute the stencil doesn't change. What changes is that now you load elements with a stride of simd length and now you do vector math instead of scalar math.

4.2 Interface

Below is the interface for each level of the kernel without the DLT

Kernel(x, y, w, m0, k, x_stride, y_stride, offset, mu)

UKernel(x, y, w, m0, k, x_stride, y_stride, offset, mu, i0)

BCodelet(x, y, w, m0, k, x_stride, y_stride, offset, mu, p_off)

BB(x, y, w, m0, k, x_stride, y_stride, offset_out, offset_in)

The interface simplifies once you perform the DLT because the DLT handles some of the parameters like offset and strides. Want these removed so that all the data that you are retrieving is contiguous, this contiguous data is desirerable so that indexing can be kept to a minimum.

Kernel(x_dlt, y_dlt, w, m0, k, mu)

UKernel(x_dlt, y_dlt, w, m0, k, mu, i0)

BCodelet(x_dlt, y_dlt, w, m0, k, mu, p_off)

BB(x_dlt, y_dlt, w, m0, k)

Vector(x_dlt, y_dlt, w, y_idx, x_idx, w_idx, simd_len)

dlt_in(m0, k, offset, simd_len, x, x_dlt, y_dlt, x_stride)

dlt_out(m0, offset, simd_len, x_dlt, y, y_dlt, y_stride)

4.3 Code

4.3.1 No DLT

BB(x, y, w, m0, k, x_stride, y_stride, offset_out, offset_in)

```
for( p = 0; p < k; ++p ){
    y_idx = (offset_out - p + m0) % m0;
    x_idx = (offset_in + m0) % m0;
    w_idx = p;
    y[y_idx * y_stride] +=
        x[x_idx * x_stride] *
```

```
        w[w_idx];
}
```

BCodelet(x, y, w, m0, k, x_stride, y_stride, offset, mu, p_off)

```
for( iu = 0; iu < mu; ++iu){
    offset_in = iu*k + p_off;
    offset_out = iu*k + p_off - offset;
    BB(...)
}
```

UKernel(x, y, w, m0, k, x_stride, y_stride, offset, mu, i0)

```
for( p_off = i0; p_off < i0 + (k); ++p_off){
    BCodelet(...)
}
```

Kernel(x, y, w, m0, k, x_stride, y_stride, offset, mu)

```
for( i0 = 0; i0 < m0; i0+=mu*k ){
    Ukernel(...)
}
```

4.3.2 With DLT

To see the code that performs the dlt, see appendix section A.8

Vector(x_dlt, y_dlt, w, simd_len, y_idx, x_idx, w_idx)

```
for( int vid = 0; vid < simd_len; vid++ ){
    y_dlt[y_idx*simd_len + vid] +=
            x_dlt[x_idx*simd_len + vid]
            * w[w_idx];
}
```

BB(x_dlt, y_dlt, w, m0, k, simd_len, rows)

```
for( p = 0; p < k; ++p ){
    y_idx = (offset_out - p + rows) % rows;
    x_idx = (offset_in + rows) % rows;
    w_idx = p;
    Vector(...)
}
```

BCodelet(x_dlt, y_dlt, w, m0, k, mu, simd_len, rows, p_off, simd_len)

```
for( iu = 0; iu < mu; ++iu){
    offset_in = iu*k + p_off;
    offset_out = iu*k + p_off;
    BB(...)
}
```

UKernel(x_dlt, y_dlt, w, k, mu, simd_len, rows, i0)

```
for( p_off = i0; p_off < i0 + (k); ++p_off){
    BCodelet(...)
```

```
}
```

Kernel(x_dlt, y_dlt, w, m0, simd_len, k, mu)

```
int rows = m0/simd_len + (k-1);
for( i0 = 0; i0 < rows; i0+=mu*k ){
    Ukernel(...)
}
```

4.4 Extracting the Steady State

The steady state of the the stencil operation is when the kernel can operate without dealing with any edge cases such as wrap around and leftovers.

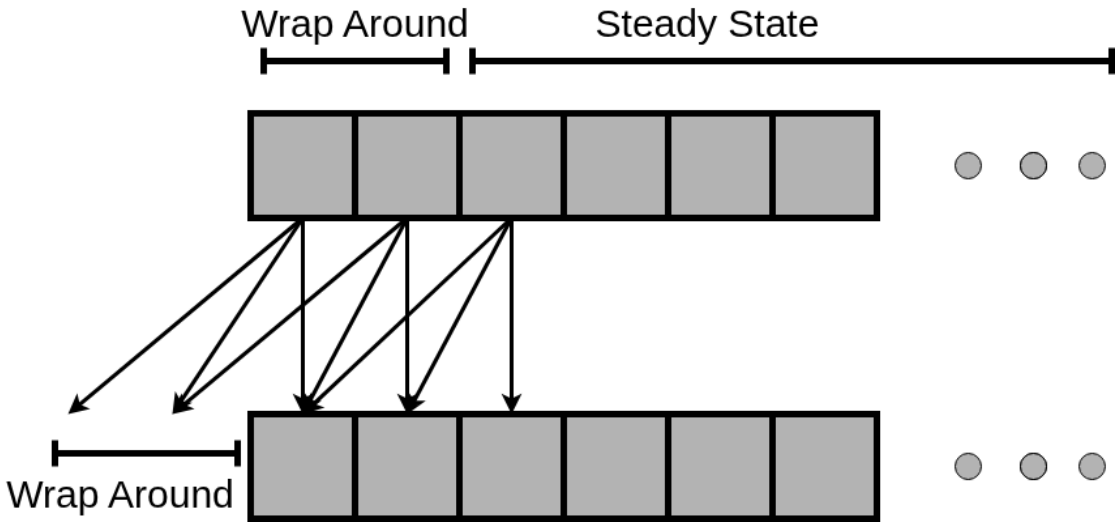

Figure 4.2: This is showing how we achieve a steady state for our kernel. We handle any cases where the input has to wrap around before we start our kernel. This makes it so our kernel does not have to worry about edge cases which would slow it down.

4.4.1 Removing the modulus

For the definintion of our stencil, there is wrap around in the computation. This means that at the beginning of the computation, the first k-1 input values are broadcasted to the end of the output array. If this was handled in the steady state we would have to have modulus in the index computation. This drastically slows down the kernel because there is more interger math happening and its especially slow because it is integer division. So instead we handle this bit outside of the kernel to ensure that no wrap around will be handled within the kernel which allows for the removal of the modulus.

4.4.2 Handling Leftovers

If the input size isn't a multiple of $mu * k$, we have to handle that remaining bit after the kernel because the μ-kernel can only handle $mu * k$ chunks of input. This is accomplished by having a simple reduction loop after the kernel.

4.5 Code Generator

For these fast kernels I created a code generator to remove the burden of hand writing each kernel especially since these kernels only have a few parameters that determine what it looks like.

- k (int)
- mu (int)
- shift (bool)
- threaded (bool)

- number of threads (int)
- number of phases (int)

The k is the only parameter that changes what computation occurs. The rest of the parameters just change how the computation will occur.

The mu (with the k) will determine how much computation is done within the μ-kernel.

Shift determines whether on each kernel iteration you load and store all y values being operated on or will only store the y's that are completed, shift over uncompleted y's, and zero out the y's that have been written.

Threaded will determine if you use multiple threads to perform the computation. If threaded is true, you cannot use the shift strategy, you need to read and write y's on each iteration. The number of threads and phases only matter if threaded is true.

There are a few things this code generator expects to exist. It expects the routines dlt_in, dlt_out, STARTUP, and COOLDOWN. These are non-unique routines that apply to every generated kernel. It also expects the macros SIMD_D, SIMD_LOAD_D, SIMD_LOAD, SIMD_STORE, SIMD_BROADCAST, SIMD_FMADD, ZERO. It uses these macros because the actual implementation of the SIMD operations are platform specific. This allows the code generator to work across many platforms and simd lengths without having to bother too much with the details. There is a small amount of work the developer would need to do to implement these macros.

4.6 Fast Kernel Example

This kernel is for a k=3, mu=2 stencil operation. The STARTUP and COOLDOWN macros handle the fringe cases at the beginning an end of the input. Having these

macros helps extract the steady state removing the need for modulous in the kernel. This example assumes that we have to load in the x and y inputs and write to the output y on each iteration. In some cases this is necessary, but in others it is not and results in a lot of extra load and store operations. The dlt_in and dlt_out macros handle the data layout transformation the allows the vector instructions. There is only one loop present because the μ-kernel is completely unrolled. The comments of iu=n are showing each μ-kernel iteration. Within each of those iterations there are multiple BBk blocks that make up a BCodelet. Then each of those BBK macros expand to load in the input x element and do each broadcast that is necessary for the broadcast block.

These are the building blocks of the fast kernel. The BROAD macro is just a mapping to a fused multiply add instruction. The BB3 is to perform a broadcast block with a k size of 3.

```
#define BROAD(y, p) SIMD_FMADD(yv##y, xv, wv##p, yv##y)


#define BB3(iu, p_off, y0, y1, y2)                          \
    {                                                       \
    int bcoff = (iu)*(3) + (p_off);                         \
    int x_idx = (i0)+(bcoff);                               \
    SIMD_LOAD_D(xv, x_dlt + (x_idx)*simd_len)               \
    BROAD(y0, 2)                                            \
    BROAD(y1, 1)                                            \
    BROAD(y2, 0)                                            \
   }
```

This is the prelude to the kernel. It performs the dlt of the input data, initializes registers, and advances the calculation to the steady state.

```
dlt_in(m0, k, offset, simd_len, x, &x_dlt, &y_dlt, x_stride);


SIMD_BROADCAST(wv0, w[0])
SIMD_BROADCAST(wv1, w[1])
SIMD_BROADCAST(wv2, w[2])
SIMD_D(xv)
SIMD_D(yv0)
...
SIMD_D(yv7)
int rows = m0/simd_len + (k-1);
int start_of_ss = k-1;
STARTUP(y_dlt, x_dlt, w, rows, k, simd_len, start_of_ss)


int i0 = start_of_ss;
```

This is the kernel where the inner microkernel is fully unrolled.

```
for( ; i0 < rows - mu*k; i0+=mu*k ) {
    SIMD_LOAD_D(yv0, y_dlt + (i0 - 2)*simd_len)
    ...
    SIMD_LOAD_D(yv7, y_dlt + (i0 - -5)*simd_len)

    // iu = 0 =============
    BB3(0, 0,
```

```
        0, 1, 2)
    BB3(1, 0,
        3, 4, 5)

    ...

    // iu = 2 =============
    BB3(0, 2,
        2, 3, 4)
    BB3(1, 2,
        5, 6, 7)

    SIMD_STORE(y_dlt + (i0+(-2))*simd_len, yv0)
    ...
    SIMD_STORE(y_dlt + (i0+(5))*simd_len, yv7)
}
```

This is the epilouge of the kernel. This deals with the bit at the end of the kernel that couldn't fit in a microkernel iteration and performs the dlt of the output to put it in the expected format.

```
COOLDOWN(i0, y_dlt, x_dlt, w, rows, k, simd_len, start_of_ss)
dlt_out(m0, offset, simd_len, x_dlt, y, y_dlt, y_stride);
```

For a full code example, it is in the appendix section A.4.

4.7 Fast Shift Kernel Example

In the previous section I showed an example of a fast kernel. Here shows if we can assume that we don't always need to read in y and write y on each iteration. This uses the same kernel parameters of k=3 and mu=2. In the kernel loop we can see now with this method there are no loads of y at the beginning of the loop, only before the kernel starts. Now on each iteration we only have to load in the x input value. At the end of the kernel we have less stores and now instead shift some of the values to put them in the correct register instead of doing a needless store and load causing more memory traffic. We also zero out most of the y values because we know that loading it in from memory would result in the same thing.

These are the building blocks of the fast shift kernel. They are the exact same as in the original fast kernel.

```
#define BROAD(y, p) SIMD_FMADD(yv##y, xv, wv##p, yv##y)


#define BB3(iu, p_off, y0, y1, y2)                          \
    {                                                       \
    int bcoff = (iu)*(3) + (p_off);                         \
    int x_idx = (i0)+(bcoff);                               \
    SIMD_LOAD_D(xv, x_dlt + (x_idx)*simd_len)               \
    BROAD(y0, 2)                                            \
    BROAD(y1, 1)                                            \
    BROAD(y2, 0)                                            \
   }
```

This is the prelude to the shift kernel. The difference here is that some elements need to be loaded in before the loop. These are the elements that were only partially computed in the STARTUP block.

```
dlt_in(m0, k, offset, simd_len, x, &x_dlt, &y_dlt, x_stride);


SIMD_BROADCAST(wv0, w[0])
SIMD_BROADCAST(wv1, w[1])
SIMD_BROADCAST(wv2, w[2])
SIMD_D(xv)
SIMD_D(yv0)
...
SIMD_D(yv7)


int rows = m0/simd_len + (k-1);
int start_of_ss = k-1;
STARTUP(y_dlt, x_dlt, w, rows, k, simd_len, start_of_ss)
int i0 = start_of_ss;


SIMD_LOAD_D(yv0, y_dlt + (i0 - 2)*simd_len)
SIMD_LOAD_D(yv1, y_dlt + (i0 - 1)*simd_len)
```

This is the kernel where the inner microkernel is fully unrolled. This version doesn't require any elements to be loaded at the beginning of the microkernel. This version also only writes completely computed y's back to memory. The uncompleted ones are shifted into the lower registers for use on the next iteration. There is also the

zeroing out of the rest of the y registers. This is needed because unlike the original fast kernel, this one does not get a fresh set of y's at the beginning of each loop.

```
for( ; i0 < rows - mu*k; i0+=mu*k ) {
    // iu = 0 =============
    BB3(0, 0,
        0, 1, 2)
    BB3(1, 0,
        3, 4, 5)

    ...

    // iu = 2 =============
    BB3(0, 2,
        2, 3, 4)
    BB3(1, 2,
        5, 6, 7)

    SIMD_STORE(y_dlt + (i0+(-2))*simd_len, yv0)
    ...
    SIMD_STORE(y_dlt + (i0+(3))*simd_len, yv5)
    yv0 = yv6;
    yv1 = yv7;
    ZERO(yv2)
    ...
    ZERO(yv7)
```

```
}
```

This is the epilouge of the kernel. There are some extra stores here to deal with the elements that were not written at the end of the last iteration of the microkernel.

```
SIMD_STORE(y_dlt + (i0-2)*simd_len, yv0)
SIMD_STORE(y_dlt + (i0-1)*simd_len, yv1)
COOLDOWN(i0, y_dlt, x_dlt, w, rows, k, simd_len, start_of_ss)
dlt_out(m0, offset, simd_len, x_dlt, y, y_dlt, y_stride);
```

For a full code example, it is in the appendix section A.5.

4.8 Multithread Code

The multithreaded code uses the kernel but adds a couple extra loop layers and changes the outermost kernel loop. The kernel that is uses has to be the one without the shift.

The block loop replaces the normal kernel loop. Each block is a mu*k chunk of data for the μ-kernel. The number of blocks, B, depends on the input size, thread count, and phase count. It is obtained through the following formula: $(rows - start\_of\_ss - mu * k)/(mu * k * P * PH)$

```
for(int bid = 0; bid < B; ++bid) {
    // steady_state_offset +
    // thread_offset +
    // block_offset +
```

```
    // phase_offset
    int i0 = start_of_ss +
             (PH*mu*k*B*pid) +
             (mu*k*bid) +
             (phase*mu*k*B);
    UKernel(...)
}
```

The thread loop is where the parallelization happens. This loop chooses a thread and that thread id is used to calculate where the thread should start its computation.

```
#pragma omp parallel for num_threads(P)
for(int pid = 0; pid < P; ++pid) {
    BlockLoop(...)
}
```

The phase loop determines the offset for all the threads.

```
for(int phase = 0; phase < PH; ++phase){
    ThreadLoop(...)
}
```

For a full code example, it is in the appendix section A.6.

Chapter 5

Analysis

Table 5.1: Specifications of evaluated CPUs

Model Name	**MHz**	**L1d**	**L1i**	**L2**	**L3**	**μ-arch**
Intel(R) Xeon(R) Gold 6338	2000.0	48K	32K	1280K	49152K	Icelake
AMD Ryzen 9 7900X	5733.0	384K	384K	12M	64M	Zen4
Intel(R) Xeon(R) E5-2660	3000.0	32K	32K	256K	25600K	Haswell
AMD EPYC 7452	2350.0	32K	32K	512K	16384K	Zen2
Intel(R) Xeon(R) Gold 6430	3400.0	48K	32K	2048K	61440K	Sapphire Rapids

Table 5.2: Compilers and Their Flags

Compiler	**Thread**	**Flags**
GCC	false	`-O3 -std=c11 -mavx512f -mfma -march=(uarch) -mtune=(uarch)`
GCC	true	`-O3 -std=c11 (-mavx2 or -mavx512f) -mfma -march=(uarch) -mtune=(uarch) -fopenmp`
Clang	false	`-O3 -std=c11 -mavx512f -mfma -march=(uarch) -mtune=(uarch)`
Clang	true	`-O3 -std=c11 (-mavx2 or -mavx512f) -mfma -march=(uarch) -mtune=(uarch) -fopenmp`
Clang-Polly	false	`-O3 -std=c11 -mavx512f -mfma -march=(uarch) -mtune=(uarch) -mllvm -polly -mllvm -polly-vectorizer=stripmine`
Clang-Polly	true	`-O3 -std=c11 (-mavx2 or -mavx512f) -mfma -march=(uarch) -mtune=(uarch) -mllvm -polly -mllvm -polly-vectorizer=stripmine -fopenmp`

5.1 Compiler Comparison

Against the Polly compiler optimized implementation, our kernel obtains much higher performance. Even giving Polly all the building blocks to obtain high performance, our entire algorithm, it can't make those final steps needed to obtain high performance. This reenforces the claim that compilers lose lots of context about the original problem by the time they are optimizing code.

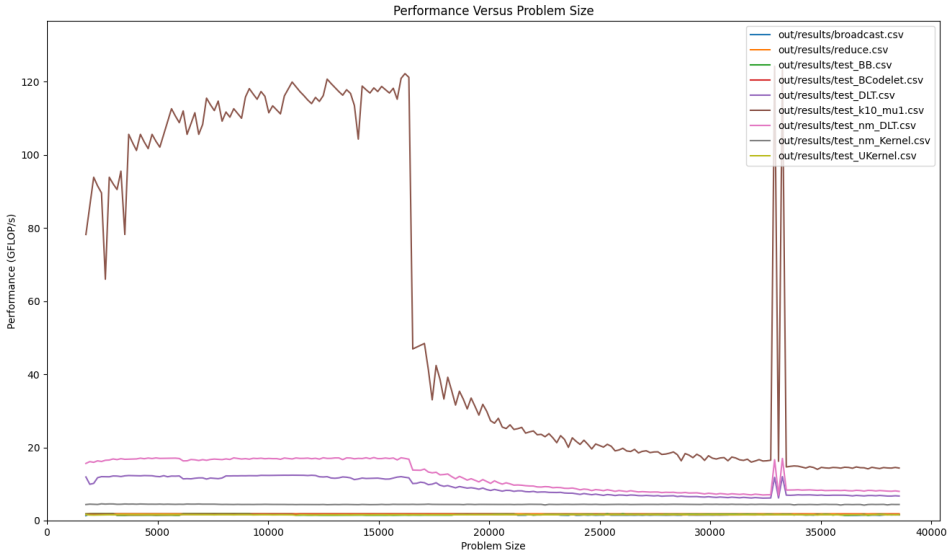

Figure 5.1: Run on an AMD Ryzen 9 7900X and compiled with Polly. The top line of this plot shows our implementation. All the other lines show various stages of our algorithm that Polly attempts to optimize. This shows that there is a large performance gap and validates the claim that Polly loses context at optimization time

5.2 Hand-Crafted Comparision

Against the hand-crafted Intel MKL implementation, our kernel obtains much higher performance. This shows that even on very tuned, hand-crafted implementations, there has been very little done to optimize the 1D stencil operation.

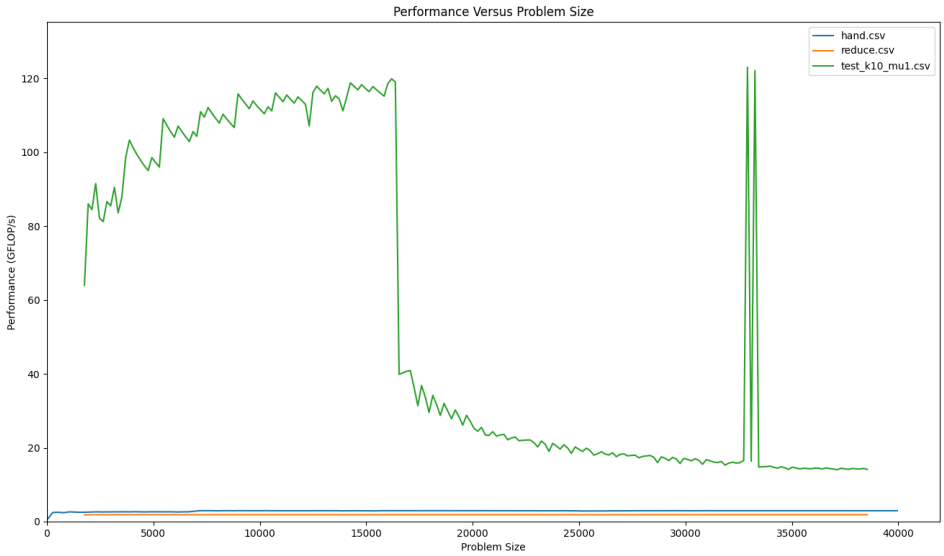

Figure 5.2: Run on an AMD Ryzen 9 7900X and compiled with GCC. The top line of this plot shows our implementation. The blue line is the Intel MKL implementation that is getting just above baseline performance. This shows the performance gap between our implementation and an optimized hand-crafted implementation.

5.3 Validating Across Microarchitectures

Across all the microarchitectures that we benchmarked, our kernel follows the same trend of performance. While we can't reach the same GFLOP/s on all of these, this is due to the CPUs having different clock rate and speeds of various components. However, on all microarchtitectures tested, our kernel obtains high performance.

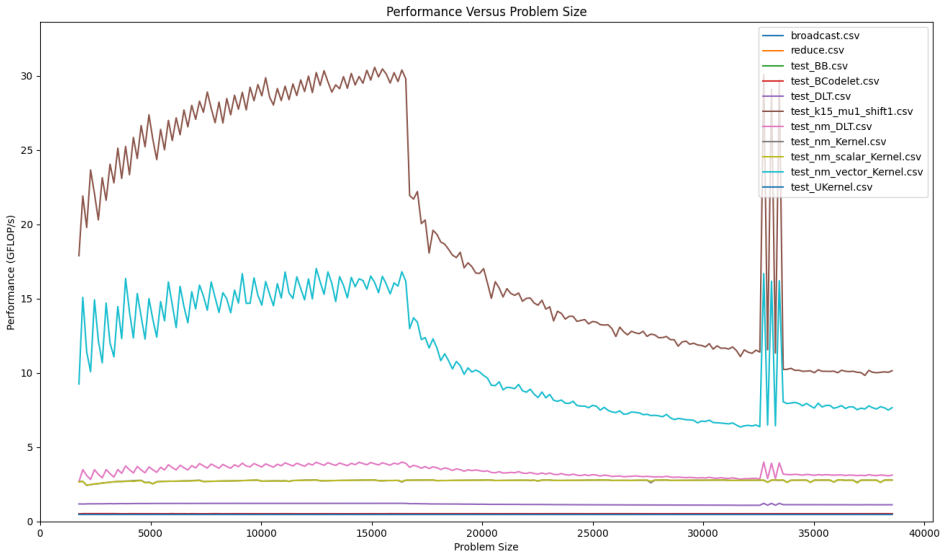

Figure 5.3: Run on an Intel(R) Xeon(R) E5-2660 and compiled with GCC. This is the Haswell microarchitecture.

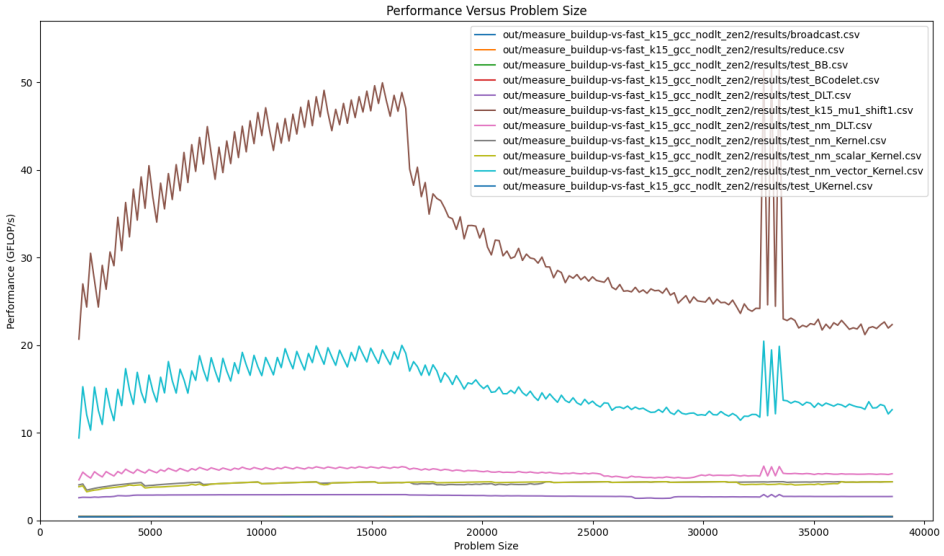

Figure 5.4: Run on an AMD EPYC 7452 and compiled with GCC. This is the Zen2 microarchitecture.

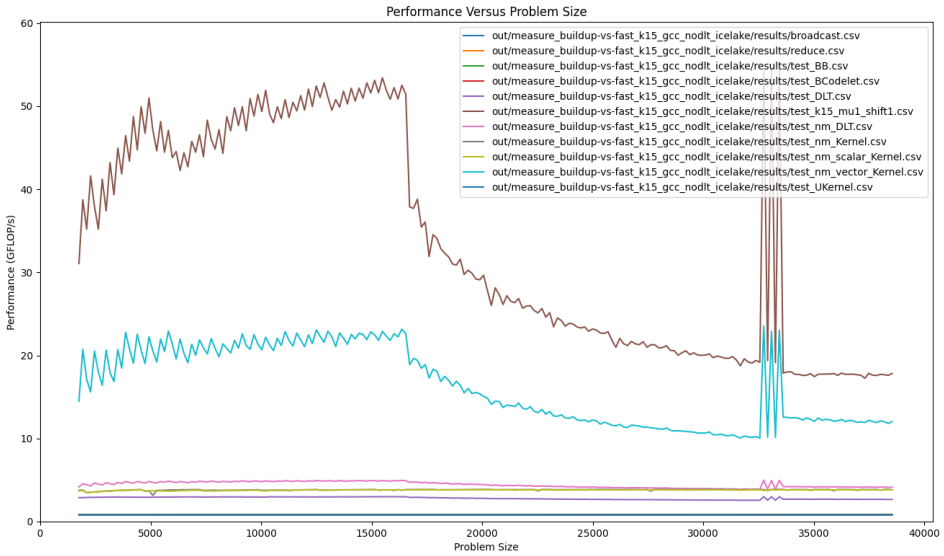

Figure 5.5: Run on an Intel(R) Xeon(R) Gold 6338 and compiled with GCC. This is the Icelake microarchitecture.

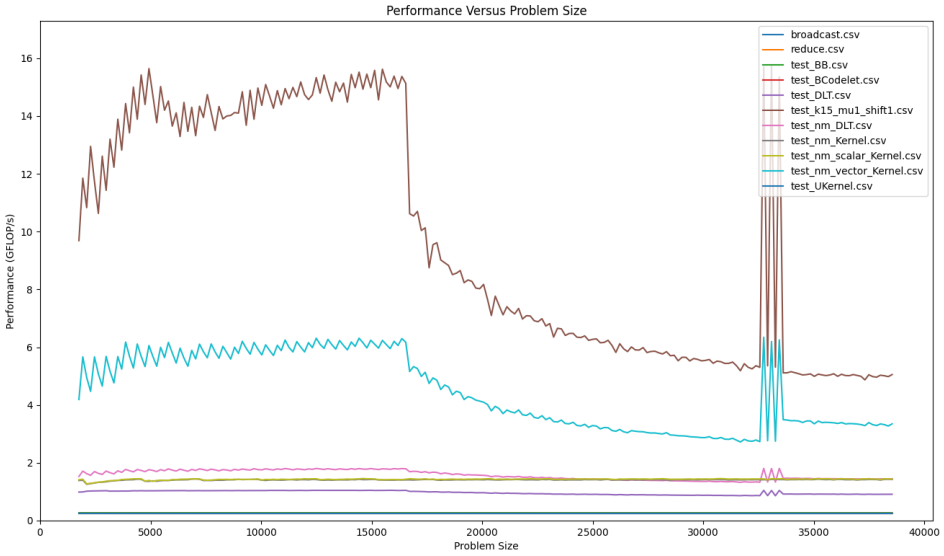

Figure 5.6: Run on an Intel(R) Xeon(R) Gold 6430 and compiled with GCC. This is the Sapphire Rapids microarchitecture.

5.4 Statistical Significance Across Microarchitectures

These next plots show a couple of things. One is the Welch's t-test value between my implementation and Polly's attempt to optimize my full algorithm. For each implementation we take 30 samples for each input size to do this t-test. Also on the graph we show the GFLOP/s of each implementation as well as the spread of GFLOP/s for each input size.

For every architecture we tested the p value is far below 0.05 which means that we reject the null hypothesis. We have enough evidence to conclude that there is a statistical difference between the GFLOP/s achieved by my implementation and Polly's.

An interesting thing to note is that while all microarchitectures have high performance for the first 16,000 input sizes, they follow different trends. This is due to the underlying differences in the microarchitectures. The icelake and sapphire rapids trend lines are very similar because these microarchitectures are similar. Haswell has some similarities to those, but it is less similar than those. Then looking at the AMD zen 2, it doesn't have the drop in performance within this region. It has a lot of up and down but generally increases in GFLOP/s until 16,000 input size.

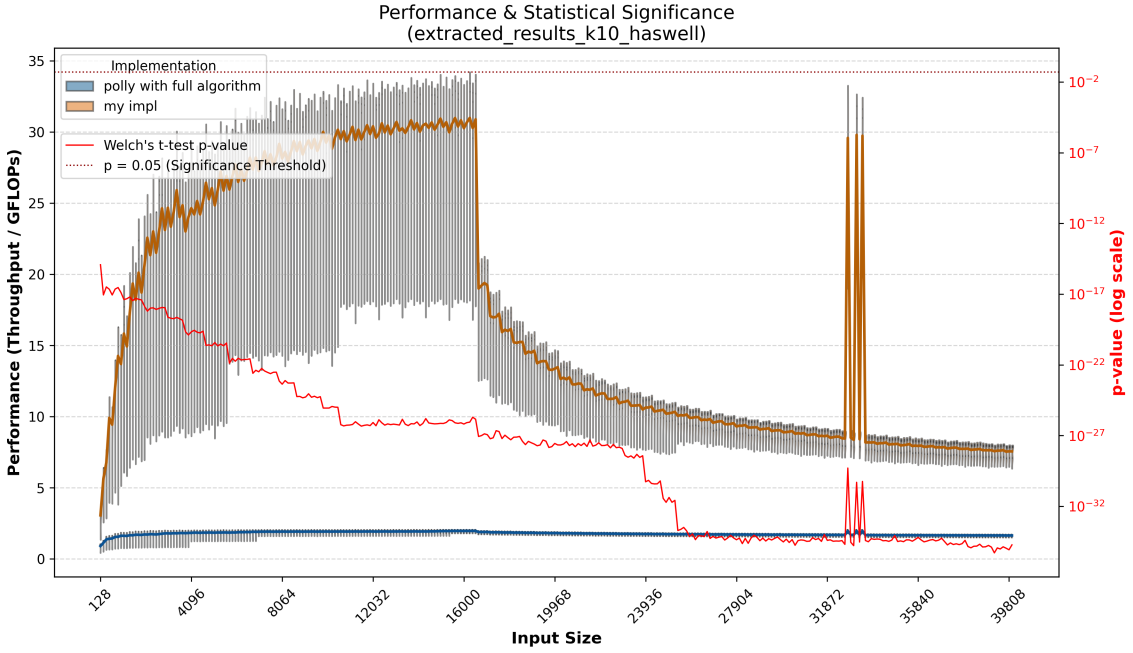

Figure 5.7: Run on an Intel(R) Xeon(R) E5-2660 and compiled with GCC. This is the Haswell microarchitecture. There is a lot of performance variation with this microarchitecture compared to the other ones tested.

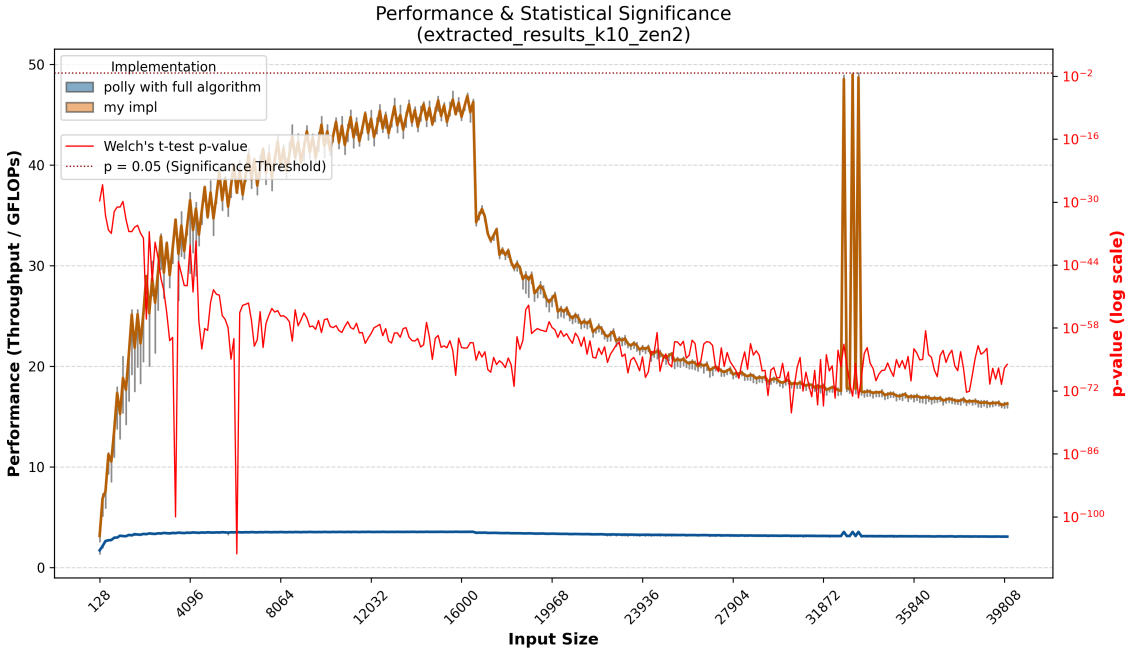

Figure 5.8: Run on an AMD EPYC 7452 and compiled with GCC. This is the Zen2 microarchitecture.

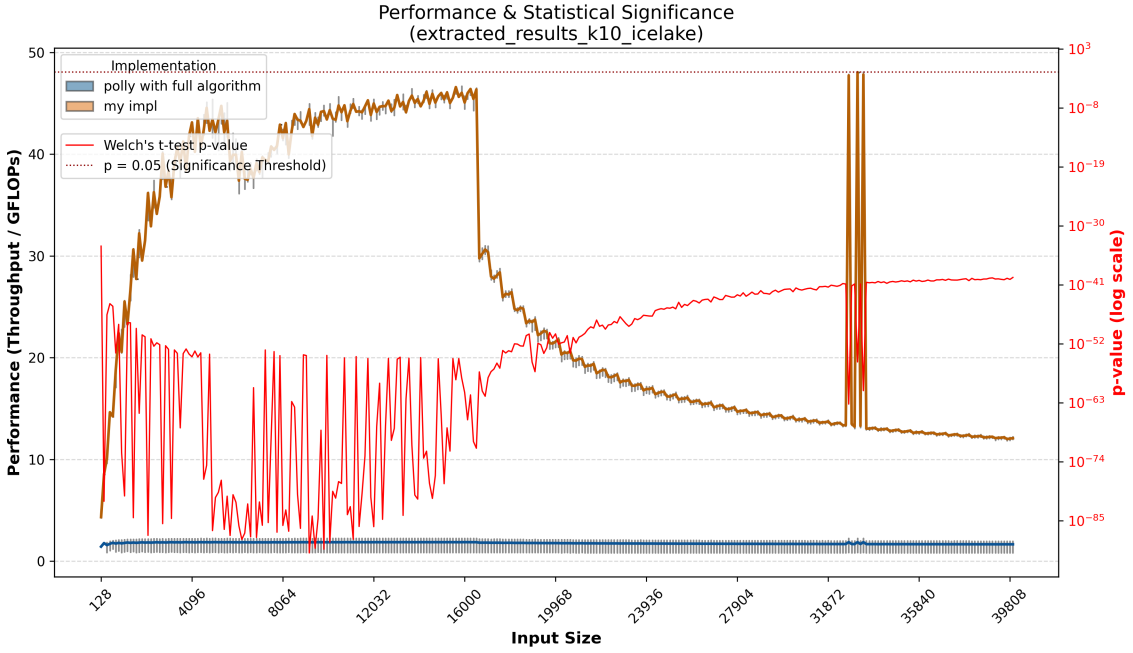

Figure 5.9: Run on an Intel(R) Xeon(R) Gold 6338 and compiled with GCC. This is the Icelake microarchitecture.

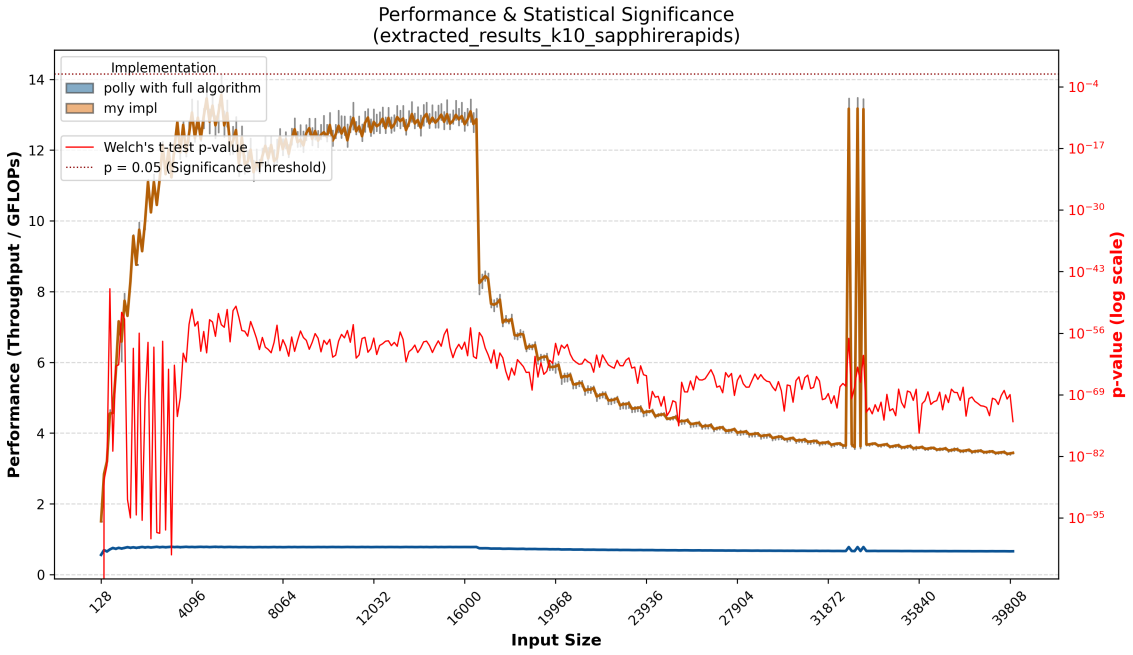

Figure 5.10: Run on an Intel(R) Xeon(R) Gold 6430 and compiled with GCC. This is the Sapphire Rapids microarchitecture.

For more plots of differing k values see appendix section A.1.1.

5.5 Varying the k and mu

For our kernel we have two parameters that determine the shape, the k and the mu. When we increase the k, we obtain higher GFLOP/s. This is because everytime we retrieve data from memory, we get to do a lot more fmas with it before having to fetch more data. When we vary the mu, what happens is also dependent on the k value. At lower k values varying the mu doesn't change the performance much. At higher k values, choosing the correct mu can massively increase your performance.

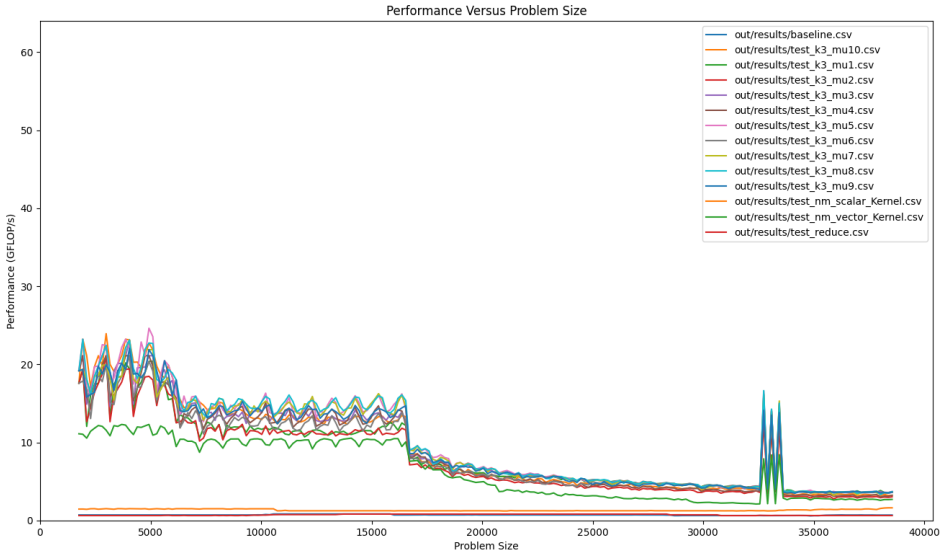

Figure 5.11: Run on an Intel(R) Xeon(R) Gold 6338 and compiled with GCC. This is when k=3. All the top lines on this plot are from varying the mu value of the kernel. Most of these obtain about the same performance because the mu matters much less when k is small.

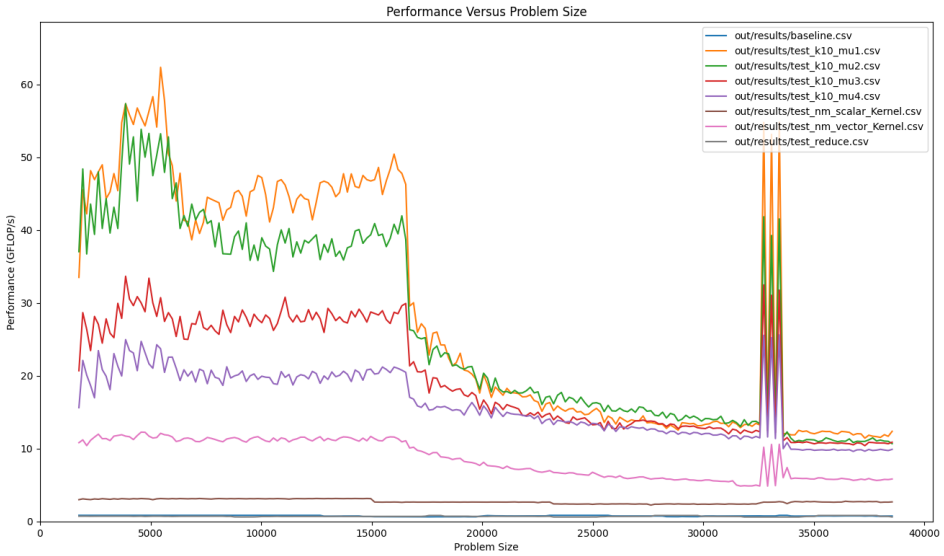

Figure 5.12: Run on an Intel(R) Xeon(R) Gold 6338 and compiled with GCC. This is when k=10. The top 4 lines of the plot are from varying the mu of the kernel. These obtain vary different performance results because the mu matter much more when k is large. In this case the higher mu values are getting lower performance because not all the data can fit in register, so there is spilling and filling.

5.6 Including the DLT

We created this algorithm with the assumption that these kernels will be part of a larger system. This allowed us to assume that we will recieve the data in the format that we expect it. We cannot always make this assumption though and wanted to see the performance when we don't recieve the data how we expect. Performing the DLT makes our kernel take a large performance hit, but it is still better to use our vector kernel than the scalar kernel even when we will have to do a lot of extra work up front to make the vector kernel work. Something to note is that this implementation of the DLT uses a naive transpose algorithm. With a more efficient transpose this performance would increase.

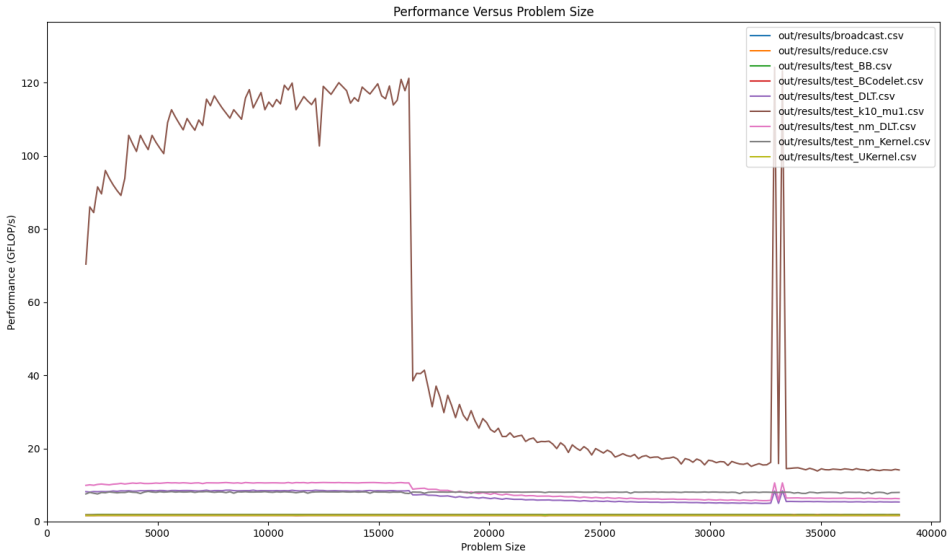

Figure 5.13: Run on an AMD Ryzen 9 7900X and compiled with GCC. These are our performance results when we recieve the data in the layout that the kernel expects. It can obtain very high performance because we don't need to do any repacking.

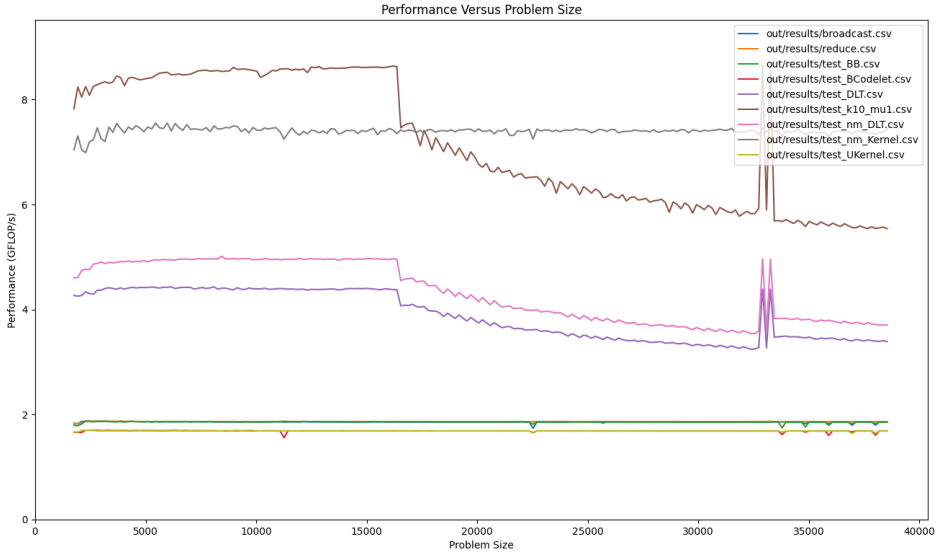

Figure 5.14: Run on an AMD Ryzen 9 7900X and compiled with GCC. These are our performance results when we have to perform the data layout transformation. There is a large performance hit because we have to repack the data before we can start any computation. Even with that large performance hit, our kernel still obtains better performance than the scalar kernel.

5.7 Shift vs No Shift

The difference between shift and no shift on gcc is small but there is still some performance increase by performing the shift instead of reading and writing the y values on each μ-kernel iteration. On clang however, there is a much larger and noticible difference which shows that the shift method should be used over no shift whenever possible.

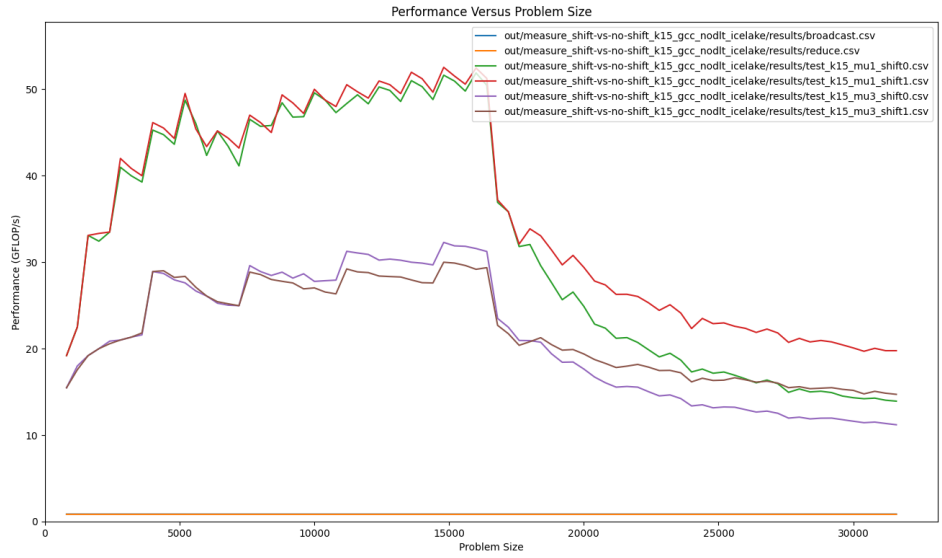

Figure 5.15: Run on an Intel Xeon Gold 6338 and compiled with GCC. This shows that there is not much difference between using the shift version of the kernel compared to noin-shift. The performance differences here are because of choosing different mu values.

For more plots of differing k values see appendix section A.1.3.

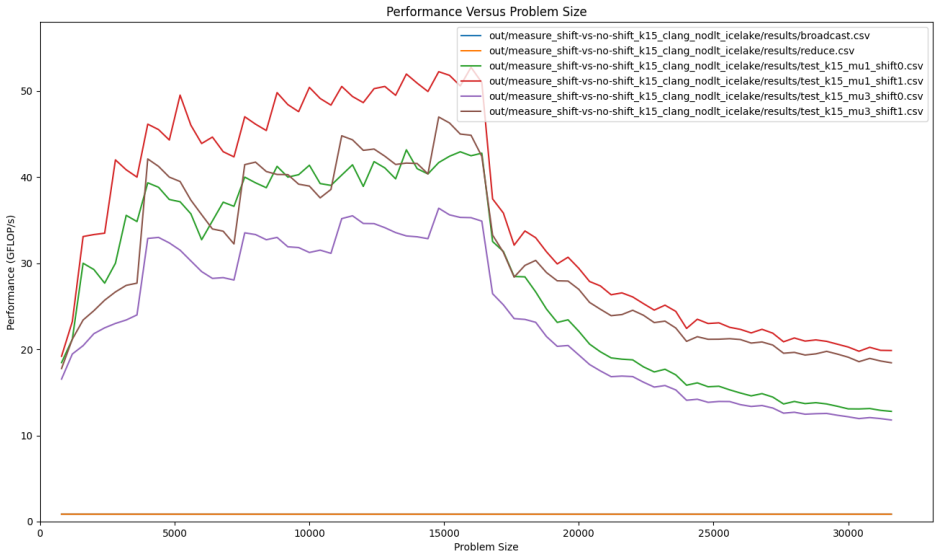

Figure 5.16: Run on an Intel Xeon Gold 6338 and compiled with Clang. With Clang there is a much bigger difference when using the shift method compared to no shift. Even with the worse mu value, the shift kernel gets higher performance than a no shift with the beter mu value.

5.8 AVX2 vs AVX512

AVX2 features 256 bit simd registers, these registers can hold 8 floats. AVX512 doubles the capacity making 512 bit registers that hold 16 floats. Another difference is that in AVX2 you only have access to 16 ymm registers, in AVX512 you have access to 32 zmm registers [15].

These next plots show the difference between various implementations when AVX2 or AVX512 is used. For both k=3 and k=15, using avx512 yields higher performance. For higher k values the benefit of switching to avx512 is higher. This is because at higher k values the bottleneck moves more towards computation, so having larger fma units is beneficial.

As expected, there is a speedup when using avx512 over avx2 since you can do

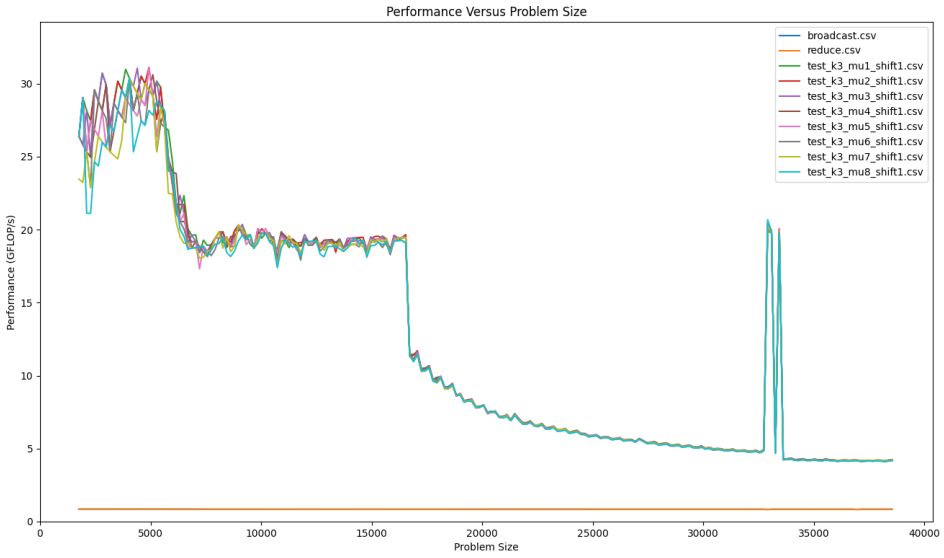

Figure 5.17: Run on an Intel Xeon Gold 6338 and compiled with GCC. This run uses avx2 and a k value of 3.

twice as many fmas at a time. However, the speedup is not as impressive as expected. For lower k values the FLOP/s is only around 10% and for higher k values it gets up to around 40%.

One of the main causes that keep this from being the 2x speedup expected is the memory system. Because for lower k values we are limited by the memory system and not computation speed, increasing the computing performance doesn't really do much. As the k value increases the bottleneck shifts more towards computation so the avx512 extension becomes more useful.

Another factor in some μ-architectures is that the fma support for AVX512 is just combining two avx2 fma units, so you end up not getting any extra computing power. [16]

A reason that the larger k sizes get a larger increase is because of the access to higher amounts of registers. Those larger k values require more data to be held in

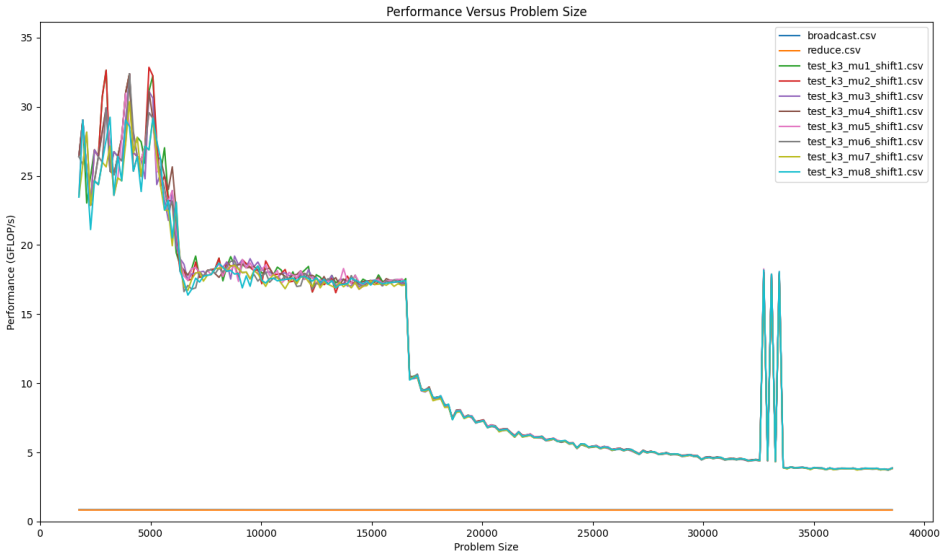

Figure 5.18: Run on an Intel Xeon Gold 6338 and compiled with GCC. This run uses avx512 and a k value of 3.

register, so having more availible registers removes the need to do a lot of moving in and out of registers.

For more plots see appendix section A.1.4.

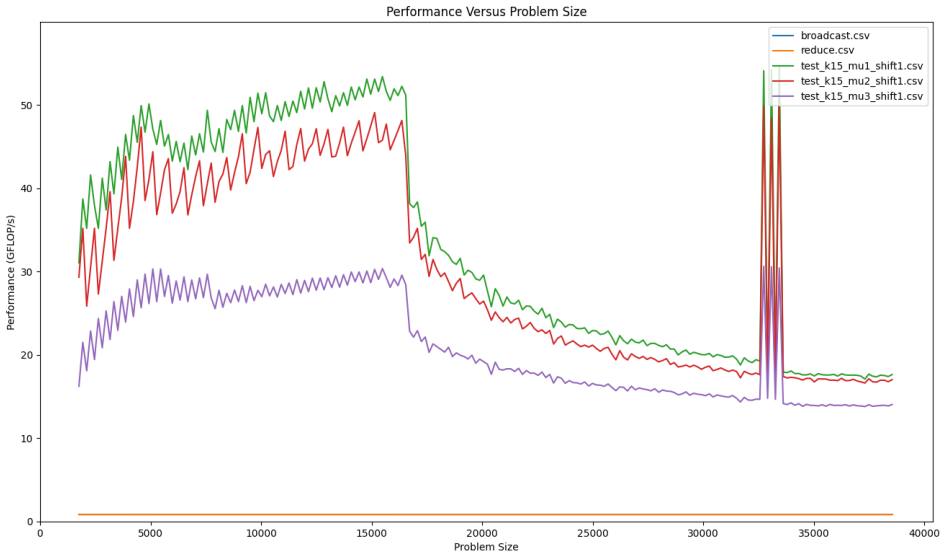

Figure 5.19: Run on an Intel Xeon Gold 6338 and compiled with GCC. This run uses avx2 and a k value of 15.

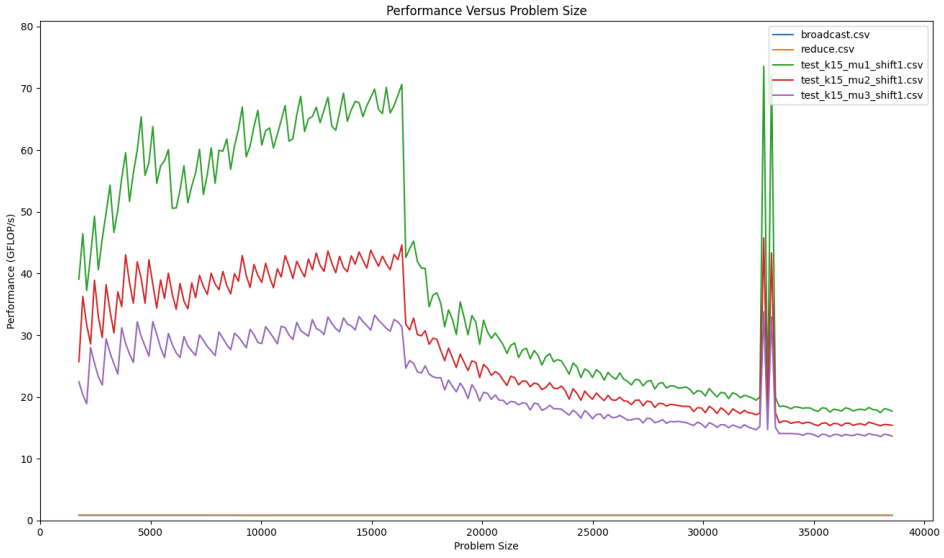

Figure 5.20: Run on an Intel Xeon Gold 6338 and compiled with GCC. This run uses avx512 and a k value of 15.

5.9 Multithread Implementations

There are two important things to figure out about multithreaded implementations. First, when does the performance of multithreaded implementations become better than the single thread implementation. Second, how do the parameters of the multithreaded implementations affect performance.

5.9.1 Where Multithreading Becomes Useful

For small input sizes the fast sequential kernel is the better implementation to use. Once the input size passes a certain point, the multithreaded kernel is better. This is because this problem requires a lot of memory retrieval and when many threads are trying to get memory, the CPU can't bring in the data fast enough. This causes threads to idle while waiting for memory. Another memory issues at low input sizes is that most of the data can fit into a single cache. This causes more waiting as the threads are trying to access the same cache for the needed data.

The ceratin poitn where thereading becomes better depends heavily on the CPU. For CPUs with larger caches this will be a much bigger input size to reach higher levels of caches or main memory.

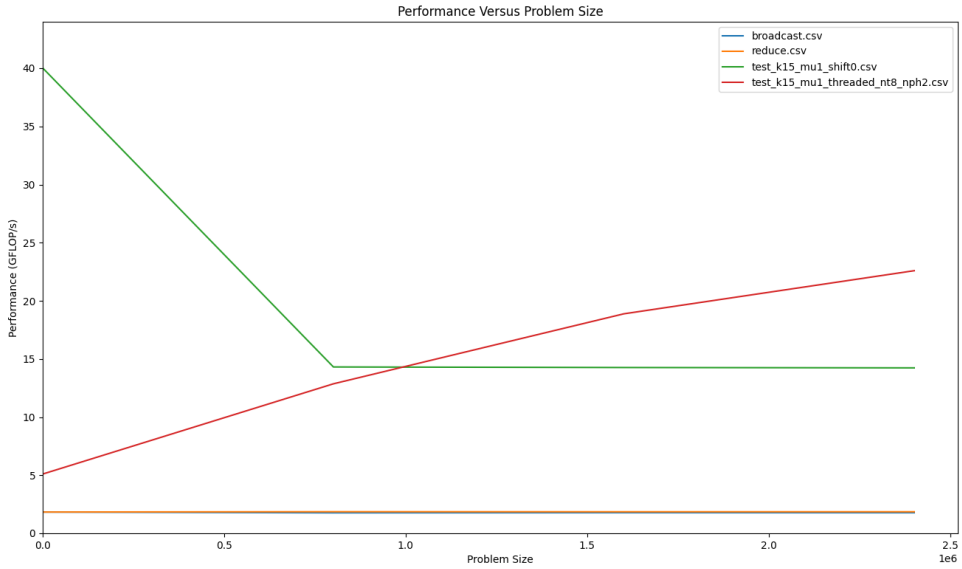

Figure 5.21: GCC with threading, Run on an AMD Ryzen 9 7900X and compiled with GCC. This plot is used to find the crossover point where multithreading becomes faster than the sequential version. This is for a k=15 kernel, the crossover point may be different for different k values. This crossover point is also CPU dependent.

5.9.2 Comparing Multithreaded Implementations

For the multithreading implementations there are two extra knobs to turn to get more performance, number of threads and number of phases.

For the first experiment size, which is small, the threaded version using only one thread is the most performant. This is because the overhead of the threads is more expensive than just doing the computation.

The performance lines generally follow what is expected, more threads, higher performance. Except for one and two threads, the single thread is more performant. With only two threads the computation speed boost is not enough to hide the overhead of managing the threads.

The performance increase from more threads is not as great as expected. 64

threads is only about 2x as performant as 4 threads, when ideally you'd want a 16x increase in performance.

Also, these plots do not test a large enough input size to see where the performance starts to drop off, if it does.

For more plots see appendix section A.1.5.

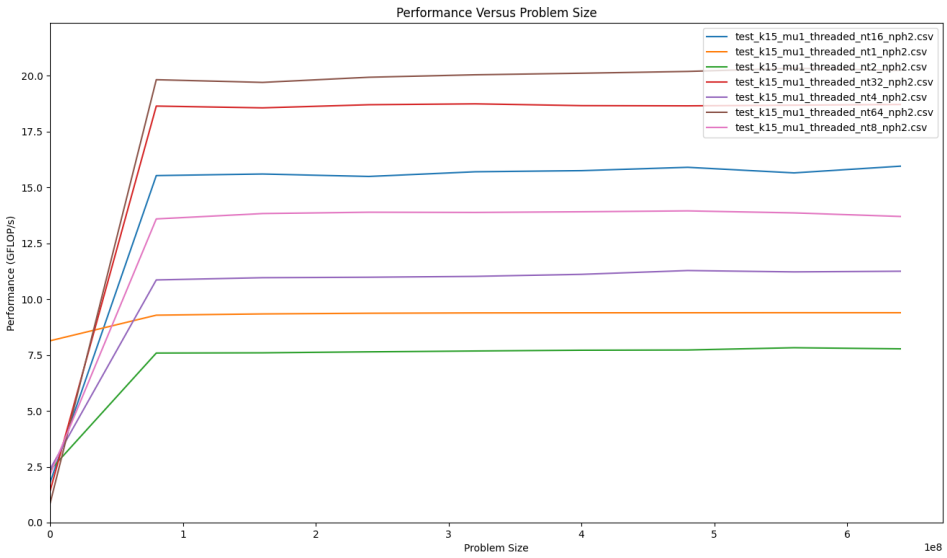

Figure 5.22: Run on an Intel(R) Xeon(R) Gold 6338 and compiled with GCC. This shows the performance when varying the thread count with two phases. Generally there is a performance increase when using more threads. Except for when we use two threads. This is becasue the two threads live on the same core and do not get access to more L1 caches.

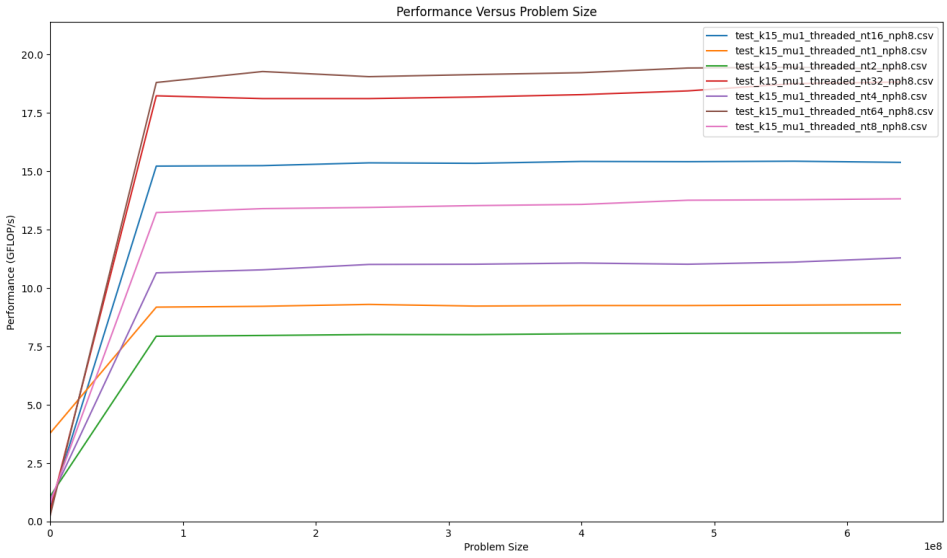

Figure 5.23: Run on an Intel(R) Xeon(R) Gold 6338 and compiled with GCC. This shows the performance when varying the thread count with eight phases. Generally there is a performance increase when using more threads. Except for when we use two threads. This is becasue the two threads live on the same core and do not get access to more L1 caches.

5.10 Spike Analysis

On all of our results plots, they follow the same trend. We get very high performance for input sizes from 0-16,000. Then we get a sudden drop followed by a slow exponential decay. Then we get a sudden spike of performance for input sizes of 32,000-33,000. Then it continues the exponential decay forever. At the 32,000 input size mark, this would be 128KB of memory. At that memory size on Linux, malloc starts to use the mmap syscall directly. To test if this is what is causing this spike, we re-run the experiment, but directly use mmap for all sizes to see if this spike disapears. This run does make the spike disapear, but now all the performance in the first 16,000 input sizes is lost.

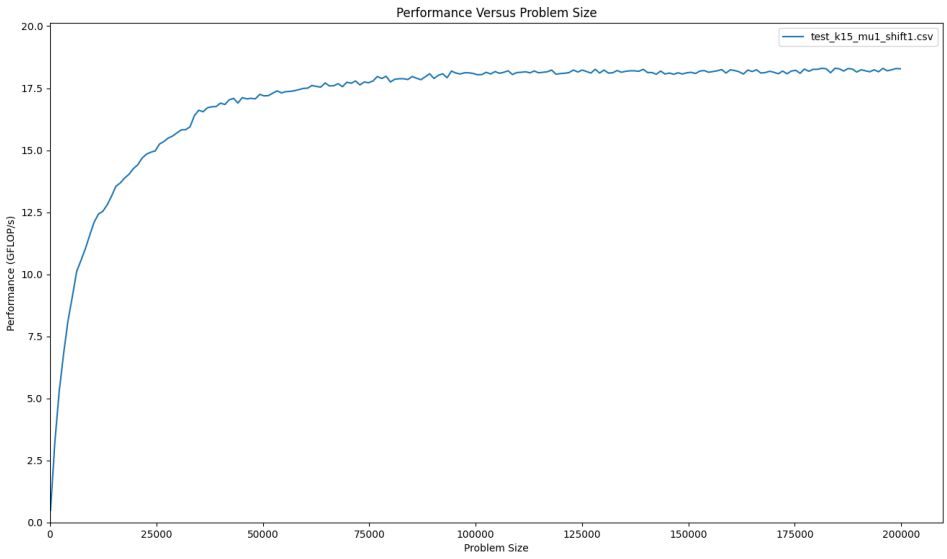

Figure 5.24: Run on an AMD Ryzen 9 7900X and compiled with GCC. This is a run where instead of using malloc for x_dlt and y_dlt arrays we use mmap. This switch removes the spike, but now all the performance in the first 16,000 input sizes is lost.

Chapter 6

Limitations and Future Work

There are a few limitations of this solution.

- We don't use explicit register renaming. Register renaming would allow for more registers to be in use in the μ-kernel and eliminate some more memory operations.
- This is only for a CPU. The idea is still applicable to GPUs and other hardware. It just wasn't applied here.
- The kernels that are generated expect the data to already be layed out. When including the data layout in the performance runs, the kernels take a huge hit in performance.

I plan to extend this work in a few ways

- Handle higher dimension kernels that are built up from a combination of data layout trtansformations and 1D stencil kernels.
- Extend solution to run on accelerators (GPUs, FPGAs).
- Extend our performance model to find peak GFLOP/s when the problem is memory bound.

Chapter 7

Summary

For the class of stencil computations, we can rearrange the order of computation to maximize instruction level parallelism and minimize register pressure to achieve a high performance kernel. Through performance comparisions we show that our solution is superior to what the Polly compiler can optimize and what hand-crafted implementations can achieve.

This analysis has been done across many different *mu*-architectures showing that the algorithm and model is generalized to work across many architectures if a few parameters of the architecture are known.

Bibliography

[1] University of California, Berkeley. "Cs267 lecture 13." UC Berkeley, accessed online. (), [Online]. Available: https://people.eecs.berkeley.edu/~demmel/cs267/lecture17/lecture17.html (visited on 04/21/2026).

[2] F. G. Van Zee and R. A. van de Geijn, "Blis: A framework for rapidly instantiating blas functionality," *ACM Trans. Math. Softw.*, vol. 41, no. 3, Jun. 2015, ISSN: 0098-3500. DOI: 10.1145/2764454. [Online]. Available: https://doi.org/10.1145/2764454.

[3] R. Veras, "A systematic approach for obtaining performance on matrix-like operations over structured and real-world data," 2017. DOI: 10.1184/R1/6714416.

[4] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan, "Data layout transformation for stencil computations on short-vector simd architectures," CC'11/ETAPS'11, pp. 225–245, 2011.

[5] K. Stock, M. Kong, T. Grosser, *et al.*, "A framework for enhancing data reuse via associative reordering," PLDI '14, pp. 65–76, 2014. DOI: 10.1145/2594291.2594342. [Online]. Available: https://doi.org/10.1145/2594291.2594342.

[6] A. Yasin, "A top-down method for performance analysis and counters architecture," pp. 35–44, 2014. DOI: 10.1109/ISPASS.2014.6844459.

[7] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan, "When polyhedral transformations meet simd code generation," *SIGPLAN Not.*, vol. 48, no. 6, pp. 127–138, Jun. 2013, ISSN: 0362-1340. DOI: 10.1145/2499370.2462187. [Online]. Available: https://doi.org/10.1145/2499370.2462187.

[8] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan, "A stencil compiler for short-vector simd architectures," ICS '13, pp. 13–24, 2013. DOI: 10.1145/2464996.2467268. [Online]. Available: https://doi.org/10.1145/2464996.2467268.

[9] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for cuda," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, Jan. 2013, ISSN: 1544-3566. DOI: 10.1145/2400682.2400713. [Online]. Available: https://doi.org/10.1145/2400682.2400713.

[10] J. Zhang, F. Franchetti, and T. Low, "High performance zero-memory overhead direct convolutions," Sep. 2018. DOI: 10.48550/arXiv.1809.10170.

[11] uops.info contributors. "Uops.info." (), [Online]. Available: https://www.uops.info/ (visited on 04/21/2026).

[12] Chips and Cheese. "Skylake: Intel's longest serving architecture." (), [Online]. Available: https://chipsandcheese.com/p/skylake-intels-longest-serving-architecture (visited on 04/21/2026).

[13] Chips and Cheese. "Zen3.drawio.png." Image file. (), [Online]. Available: https://i0.wp.com/old.chipsandcheese.com/wp-content/uploads/2022/10/Zen3.drawio.png?ssl=1 (visited on 04/21/2026).

[14] Chips and Cheese. "The nerfed fpu in ps5's zen 2 cores." (), [Online]. Available: https://chipsandcheese.com/p/the-nerfed-fpu-in-ps5s-zen-2-cores (visited on 04/21/2026).

[15] W. contributors, *Avx-512 — Wikipedia, the free encyclopedia*, [Online; accessed 21-April-2026], 2026. [Online]. Available: https://en.wikipedia.org/w/index.php?title=AVX-512&oldid=1345710086.

[16] T. P. Morgan, *Deep dive into intel's "ice lake" xeon sp architecture*, 2021. [Online]. Available: https://www.nextplatform.com/compute/2021/04/19/deep-dive-into-intels-ice-lake-xeon-sp-architecture/1633019.

Appendix A

Additional Figures

A.1 Results

A.1.1 Statistical Significance Across Microarchitectures

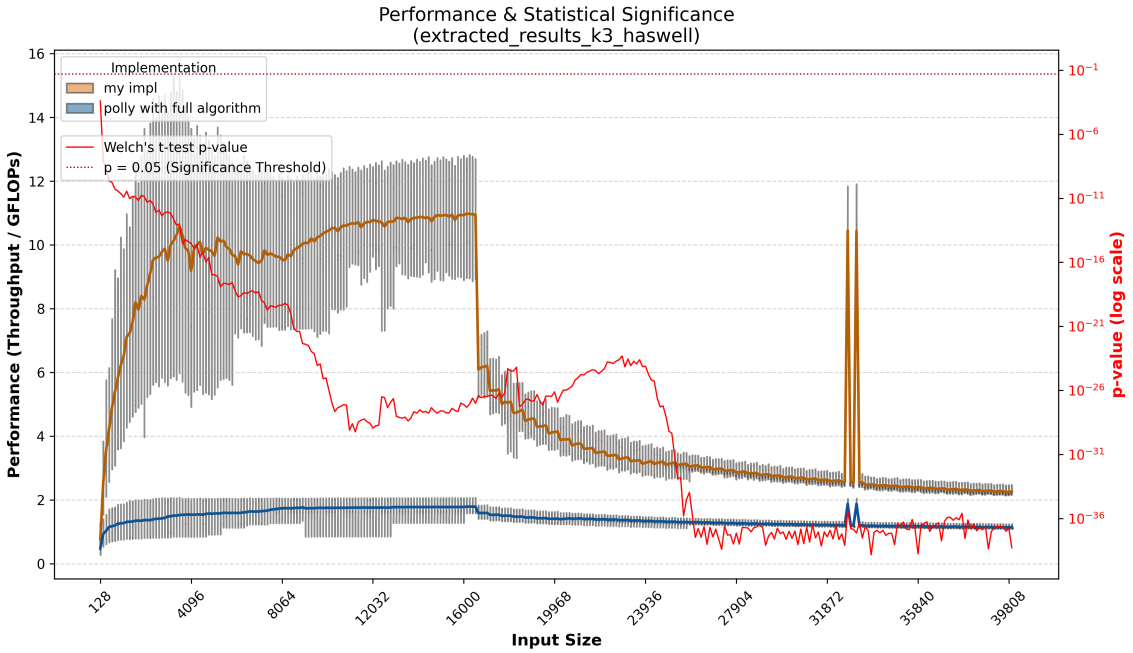

Figure A.1: Run on an Intel(R) Xeon(R) E5-2660 and compiled with GCC. This is the Haswell microarchitecture.

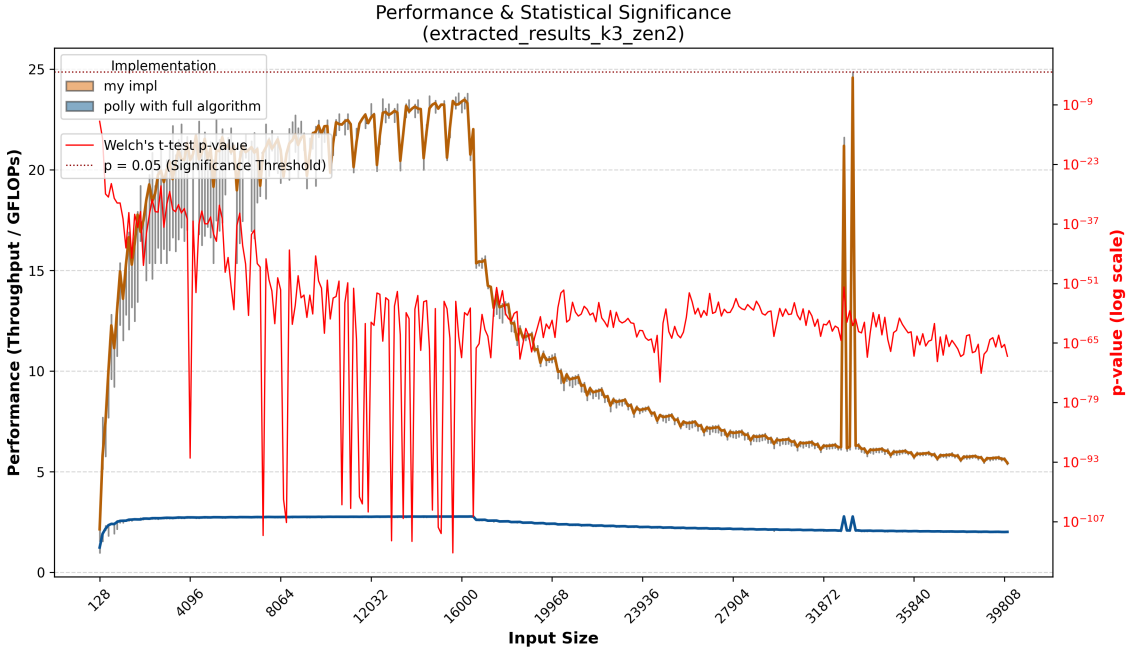

Figure A.2: Run on an AMD EPYC 7452 and compiled with GCC. This is the Zen2 microarchitecture.

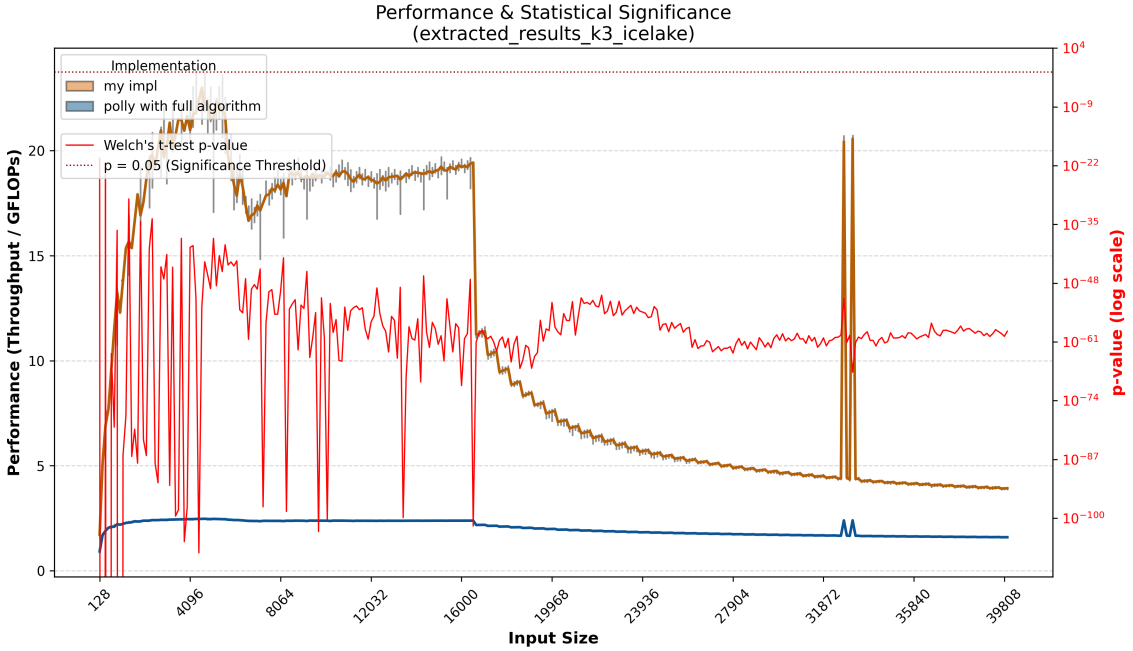

Figure A.3: Run on an Intel(R) Xeon(R) Gold 6338 and compiled with GCC. This is the Icelake microarchitecture.

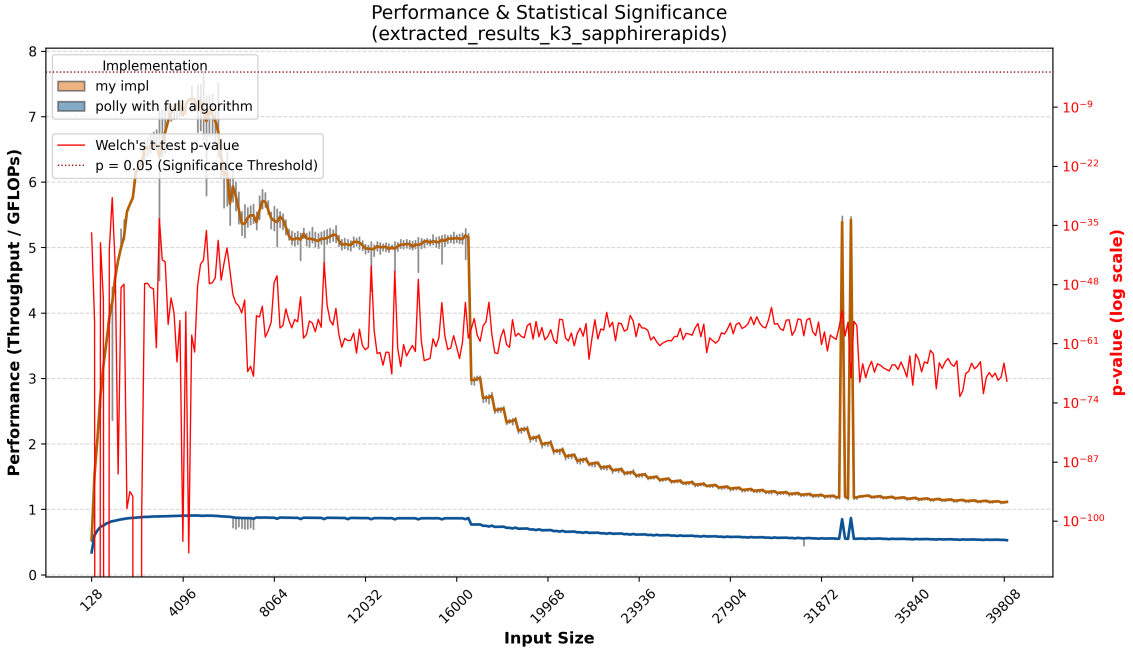

Figure A.4: Run on an Intel(R) Xeon(R) Gold 6430 and compiled with GCC. This is the Sapphire Rapids microarchitecture.

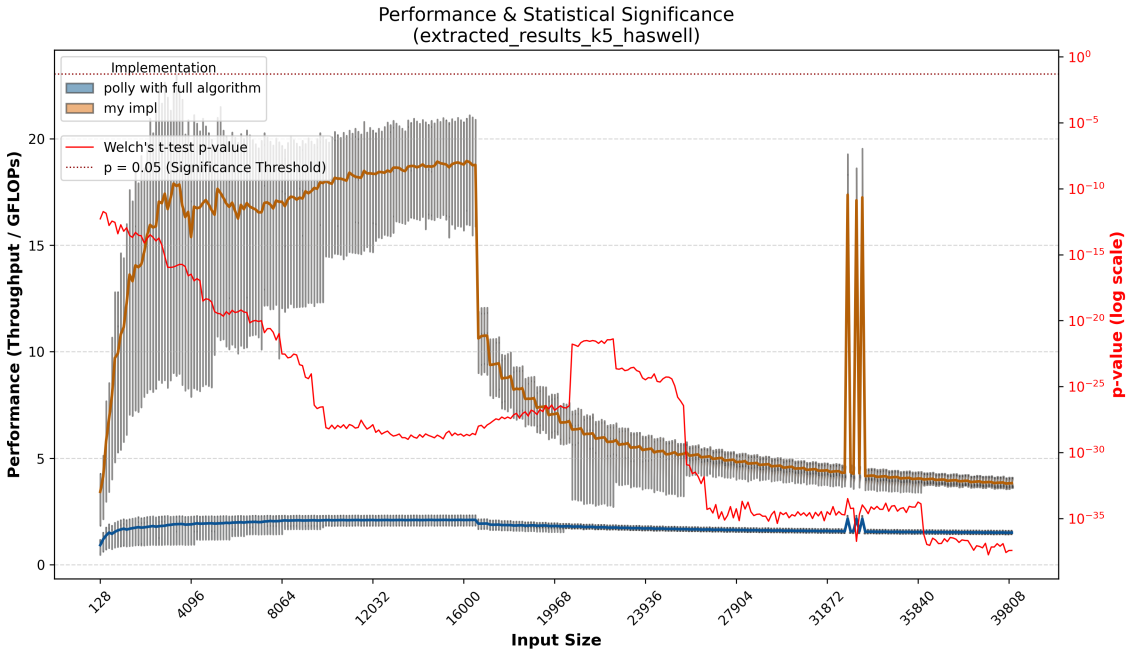

Figure A.5: Run on an Intel(R) Xeon(R) E5-2660 and compiled with GCC. This is the Haswell microarchitecture.

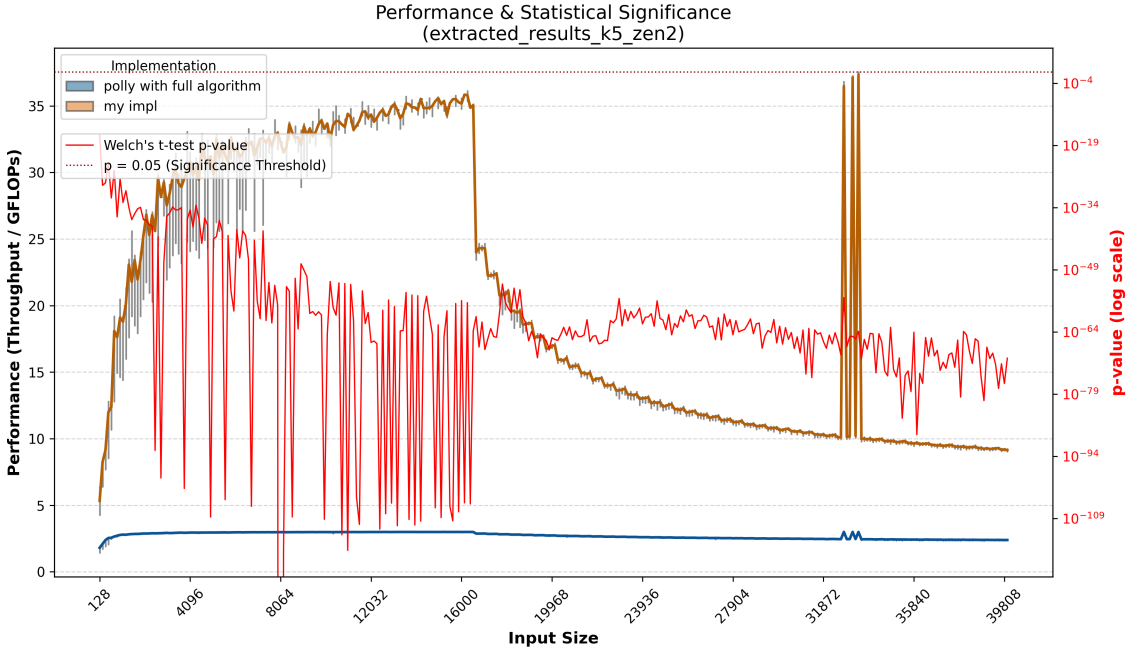

Figure A.6: Run on an AMD EPYC 7452 and compiled with GCC. This is the Zen2 microarchitecture.

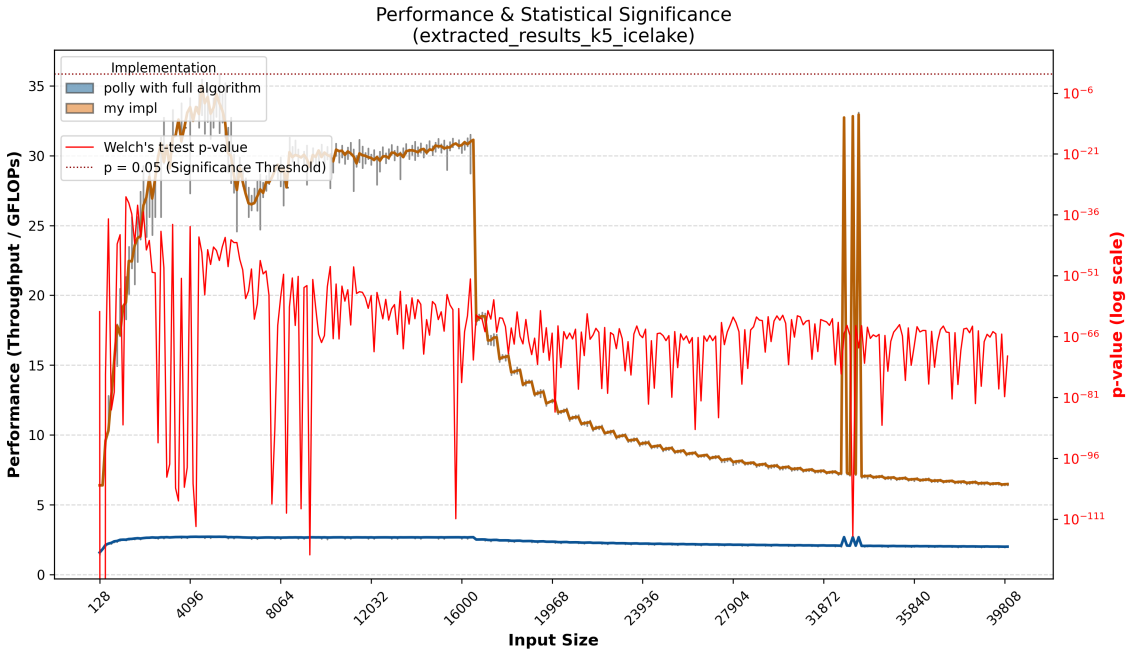

Figure A.7: Run on an Intel(R) Xeon(R) Gold 6338 and compiled with GCC. This is the Icelake microarchitecture.

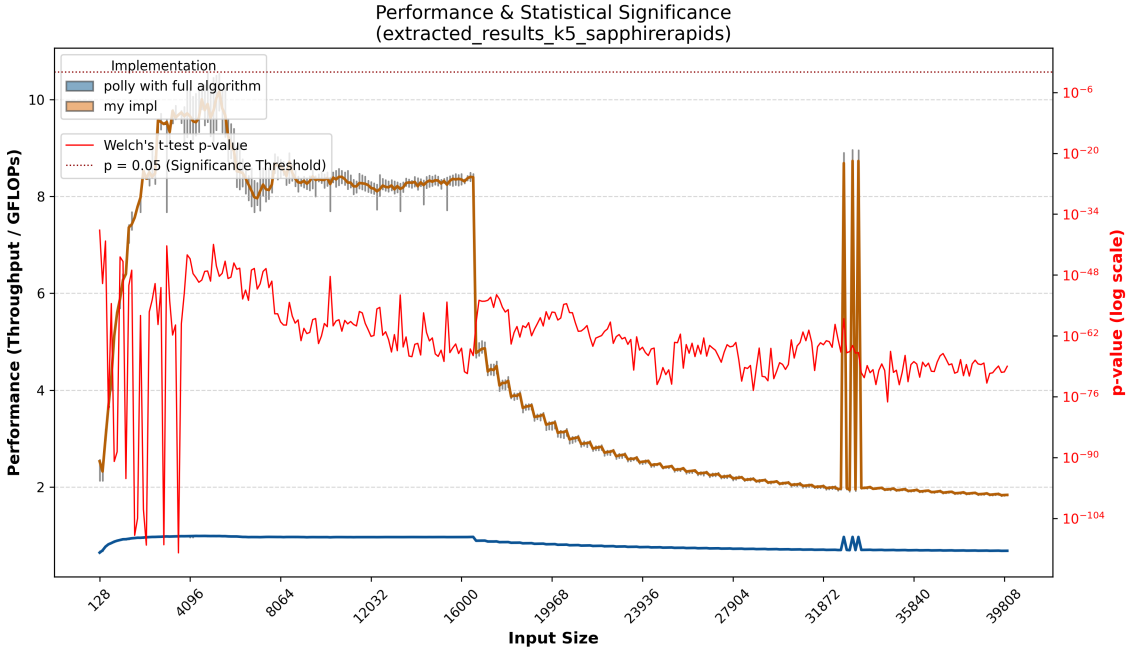

Figure A.8: Run on an Intel(R) Xeon(R) Gold 6430 and compiled with GCC. This is the Sapphire Rapids microarchitecture.

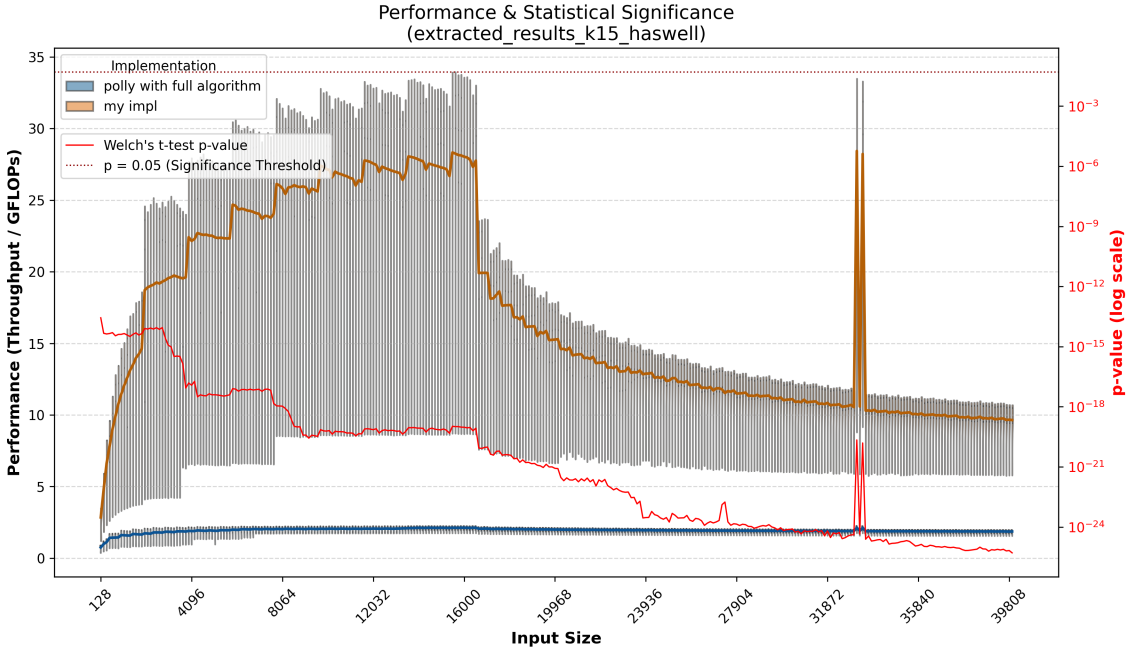

Figure A.9: Run on an Intel(R) Xeon(R) E5-2660 and compiled with GCC. This is the Haswell microarchitecture.

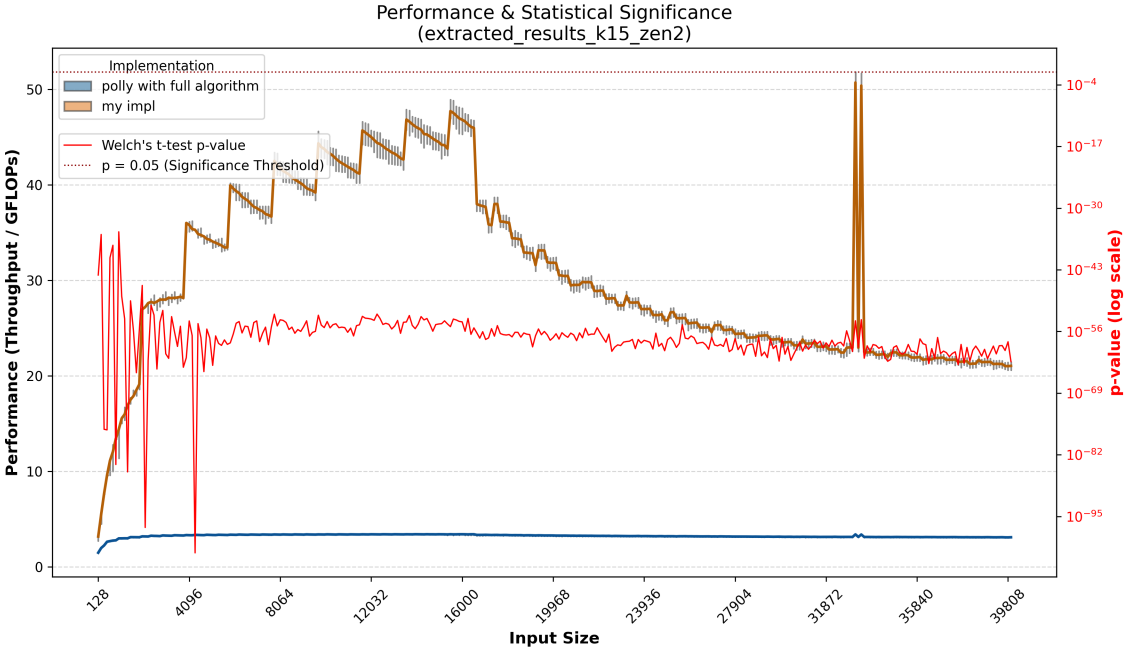

Figure A.10: Run on an AMD EPYC 7452 and compiled with GCC. This is the Zen2 microarchitecture.

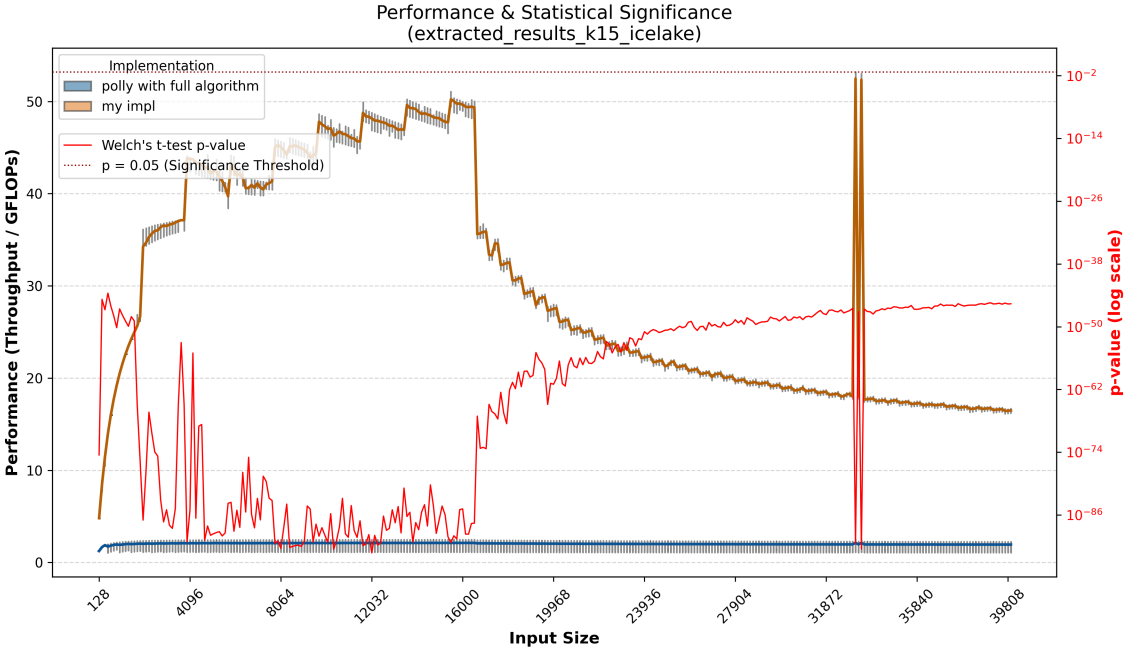

Figure A.11: Run on an Intel(R) Xeon(R) Gold 6338 and compiled with GCC. This is the Icelake microarchitecture.

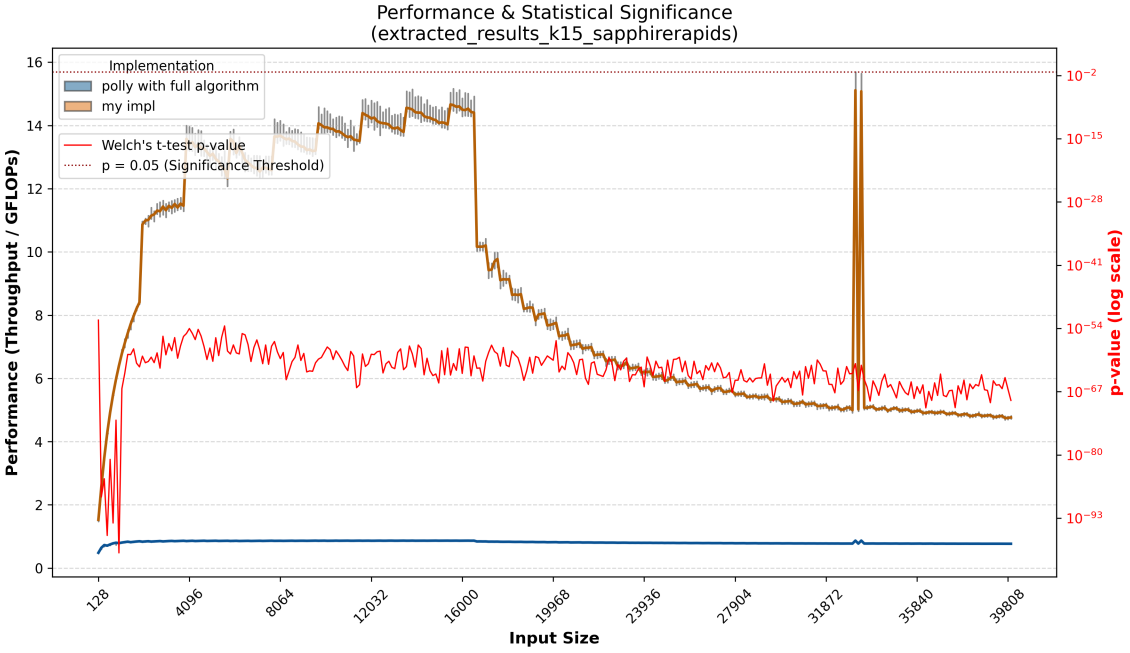

Figure A.12: Run on an Intel(R) Xeon(R) Gold 6430 and compiled with GCC. This is the Sapphire Rapids microarchitecture.

A.1.2 DLT vs No DLT

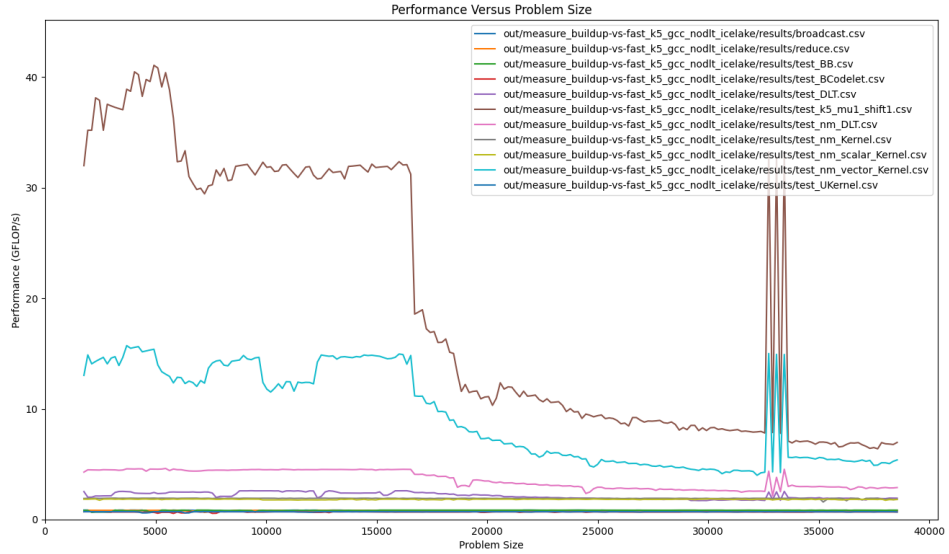

Figure A.13: GCC no threading, Intel Xeon Gold 6338, k=5. No dlt executed, but kernel still works on that data layout

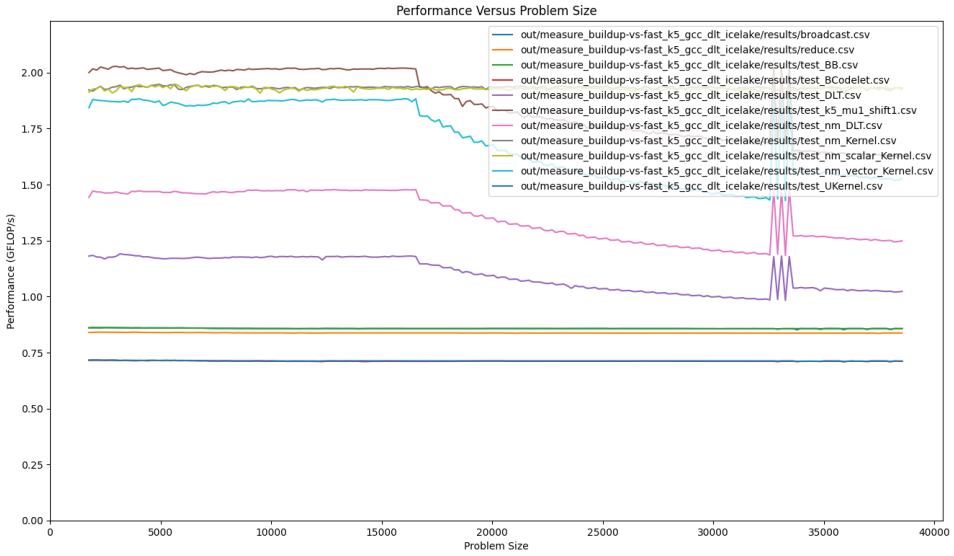

Figure A.14: GCC no threading, Intel Xeon Gold 6338, k=5. dlt executed

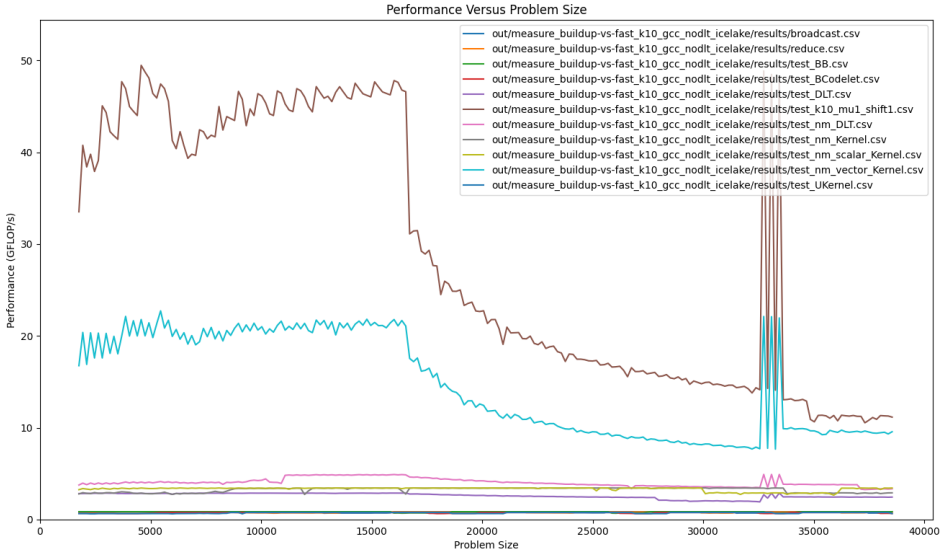

Figure A.15: GCC no threading, Intel Xeon Gold 6338, k=10. No dlt executed, but kernel still works on that data layout

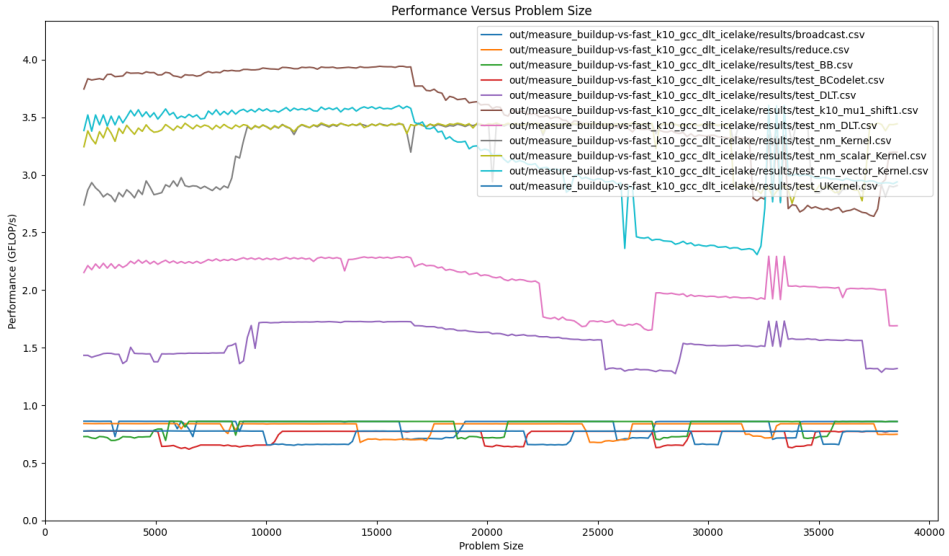

Figure A.16: GCC no threading, Intel Xeon Gold 6338, k=10. dlt executed

A.1.3 Shift vs No Shift

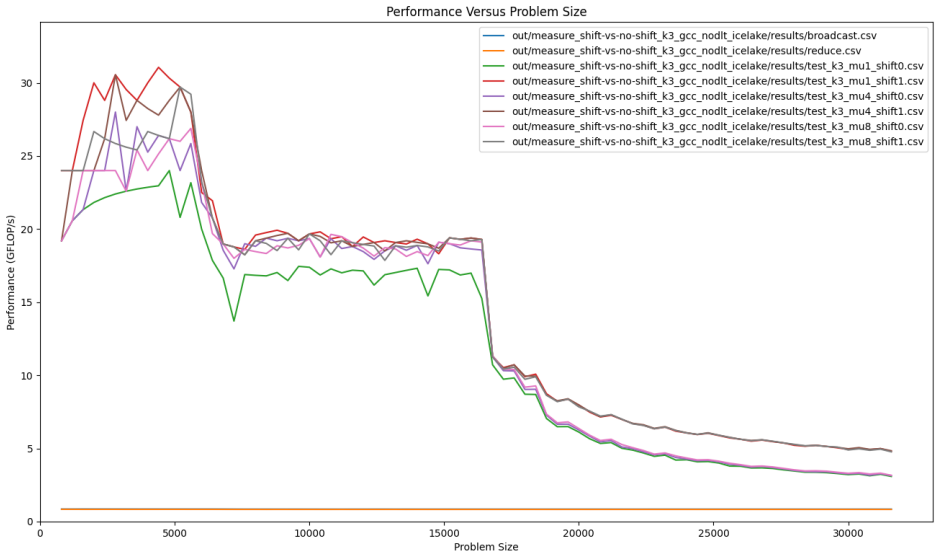

Figure A.17: GCC no threading, Intel Xeon Gold 6338, k=3, shift vs noshift

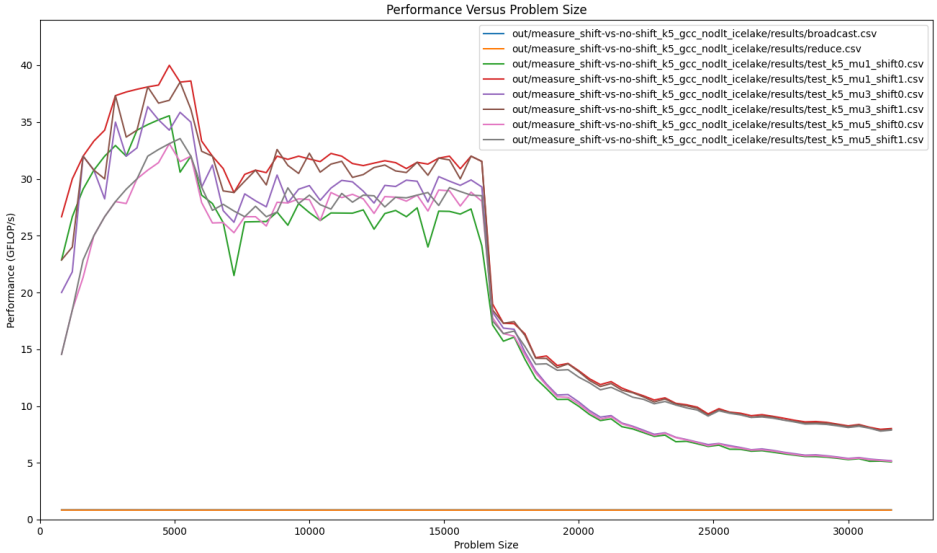

Figure A.18: GCC no threading, Intel Xeon Gold 6338, k=5, shift vs noshift

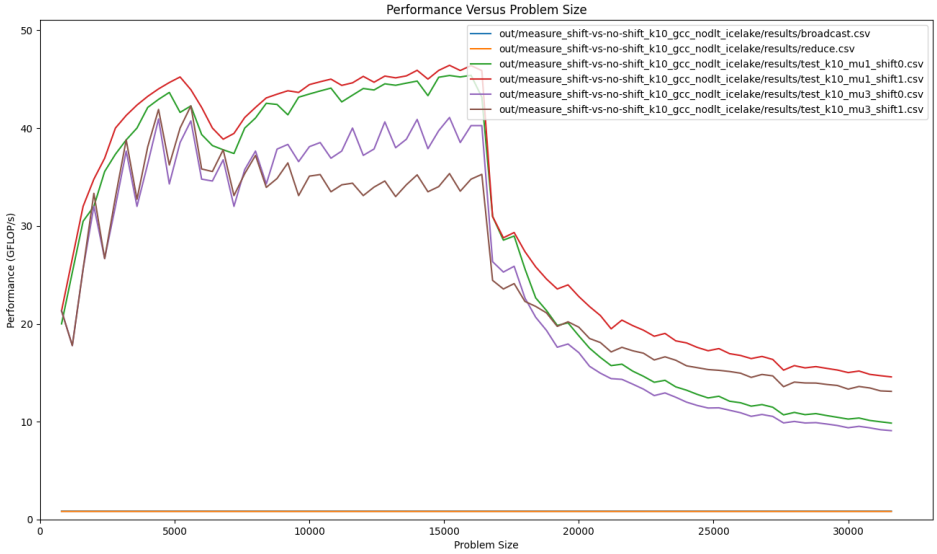

Figure A.19: GCC no threading, Intel Xeon Gold 6338, k=10, shift vs noshift

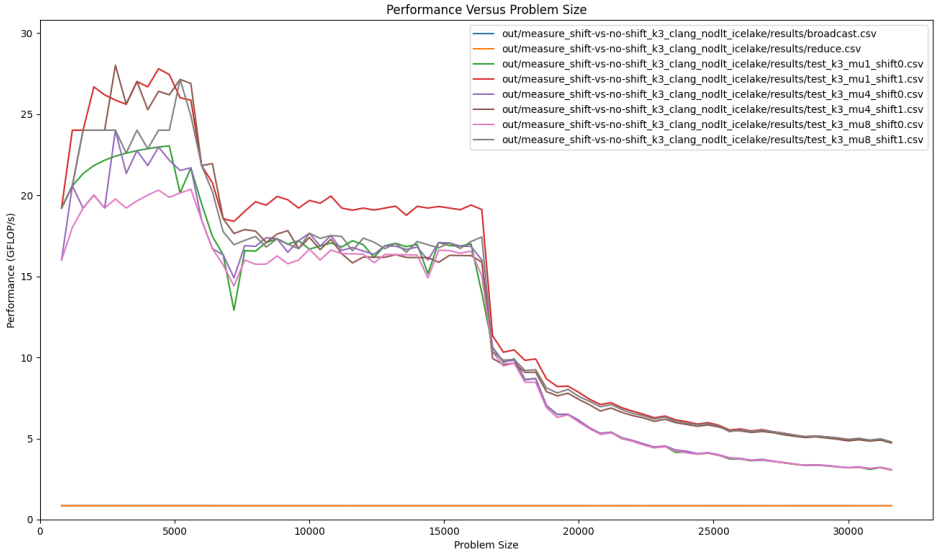

Figure A.20: Clang no threading, Intel Xeon Gold 6338, k=3, shift vs noshift

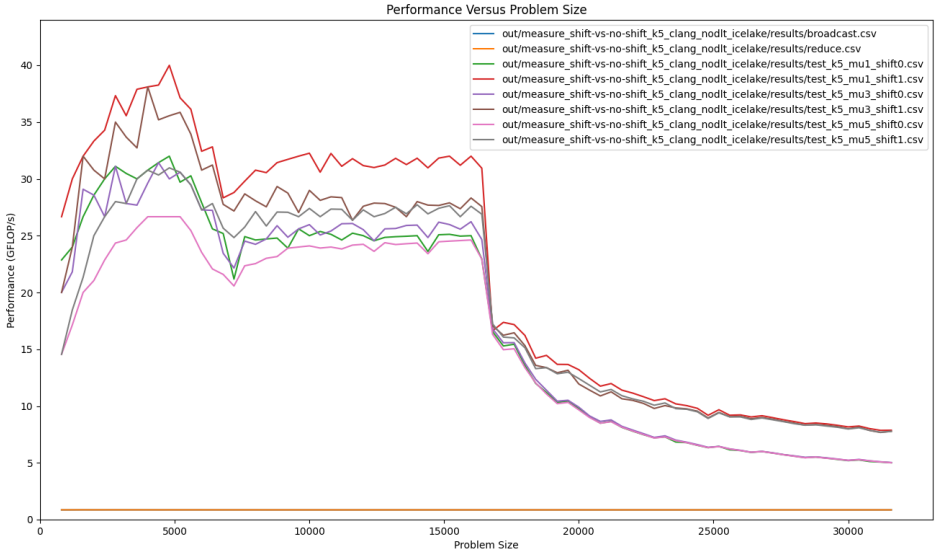

Figure A.21: Clang no threading, Intel Xeon Gold 6338, k=5, shift vs noshift

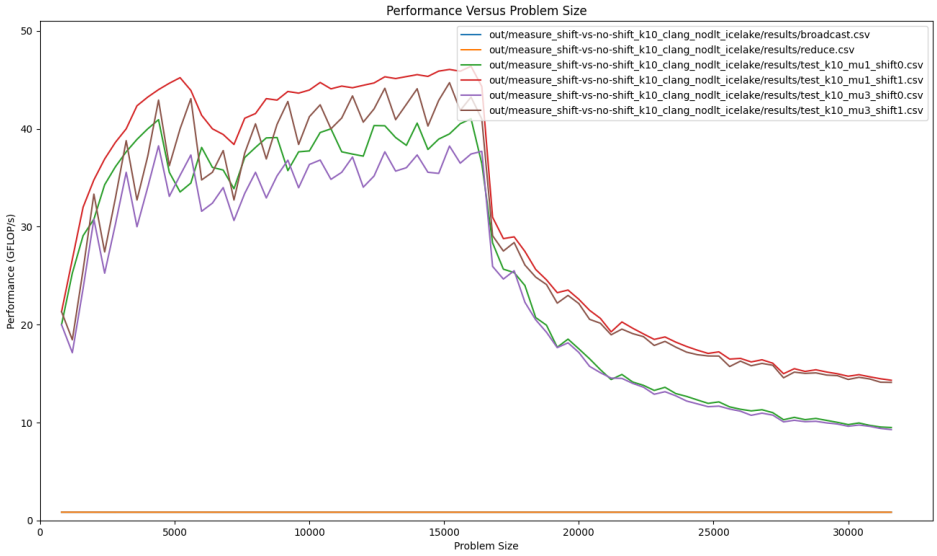

Figure A.22: Clang no threading, Intel Xeon Gold 6338, k=10, shift vs noshift

A.1.4 AVX2 vs AVX512

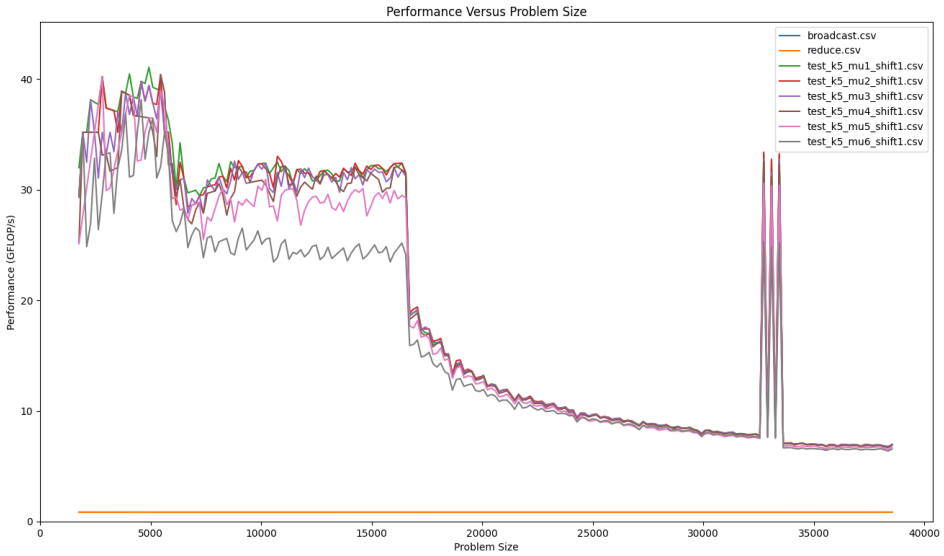

Figure A.23: GCC no threading, Intel Xeon Gold 6338, k=5, avx2

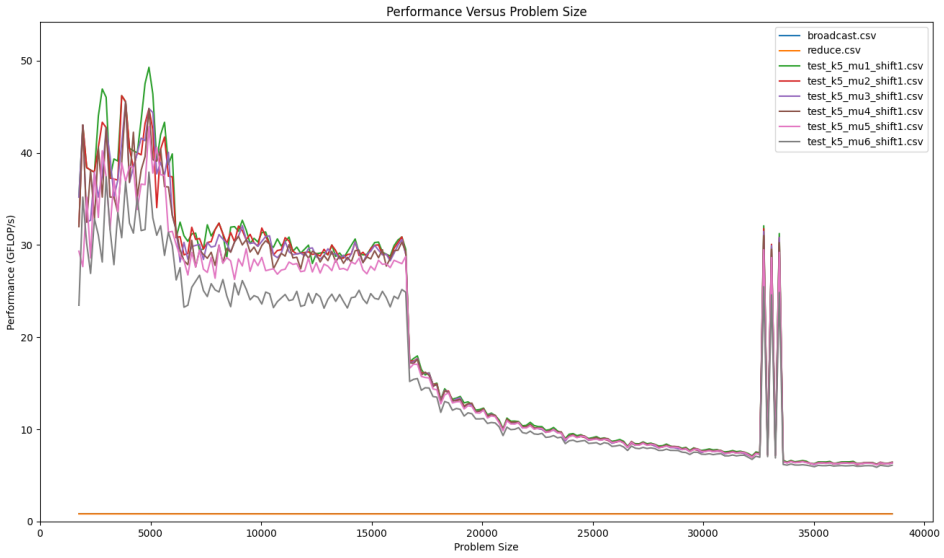

Figure A.24: GCC no threading, Intel Xeon Gold 6338, k=5, avx512

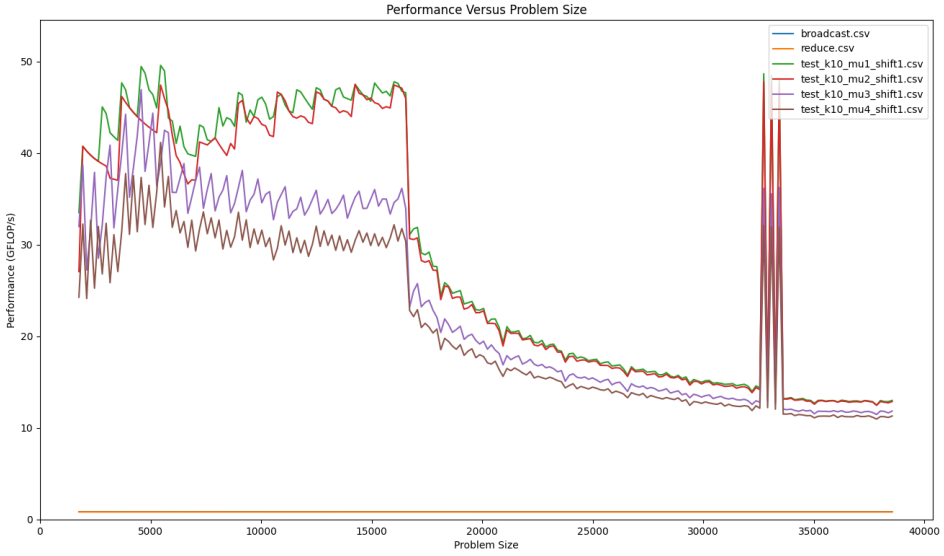

Figure A.25: GCC no threading, Intel Xeon Gold 6338, k=10, avx2

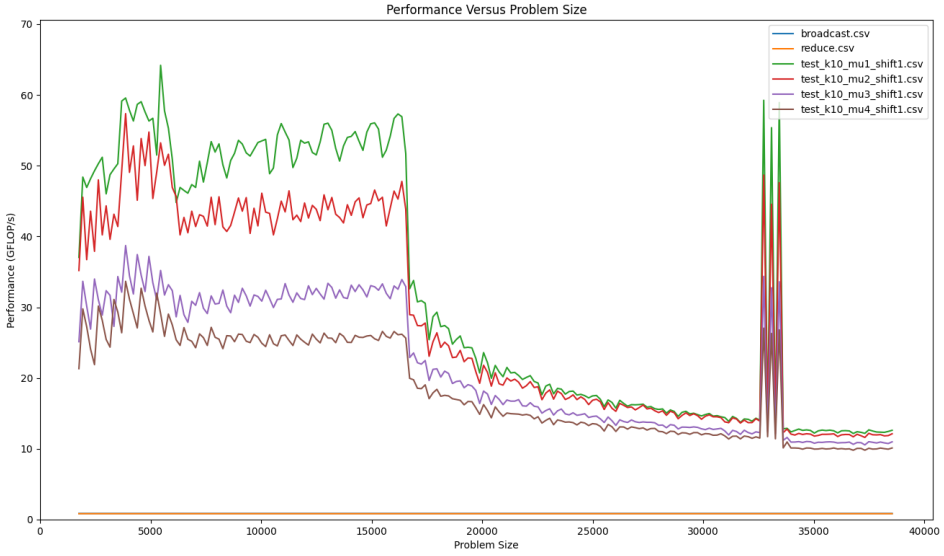

Figure A.26: GCC no threading, Intel Xeon Gold 6338, k=10, avx512

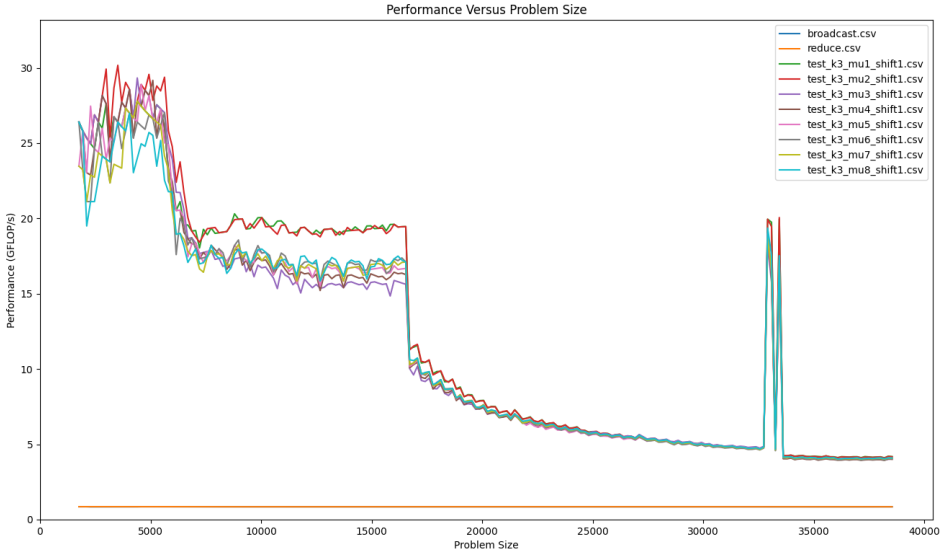

Figure A.27: Clang-Polly no threading, Intel Xeon Gold 6338, k=3, avx2

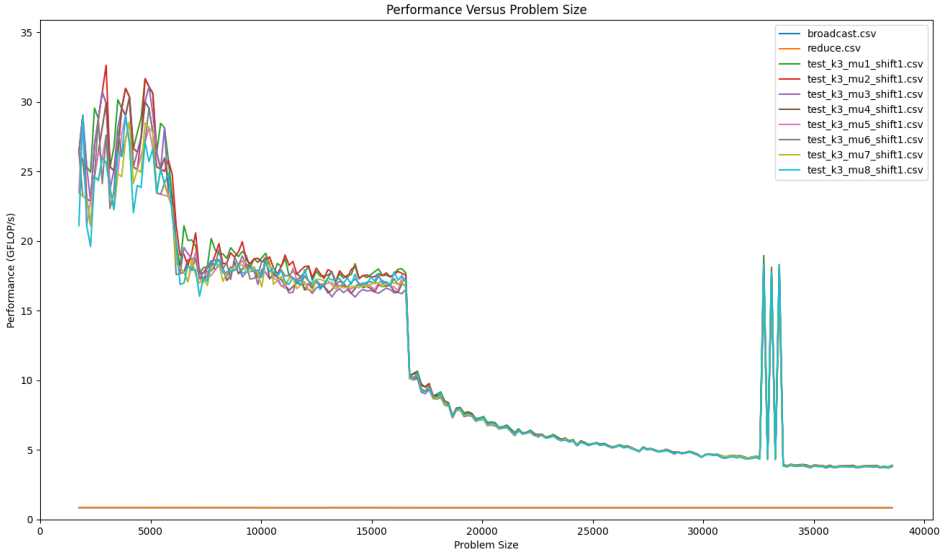

Figure A.28: Clang-Polly no threading, Intel Xeon Gold 6338, k=3, avx512

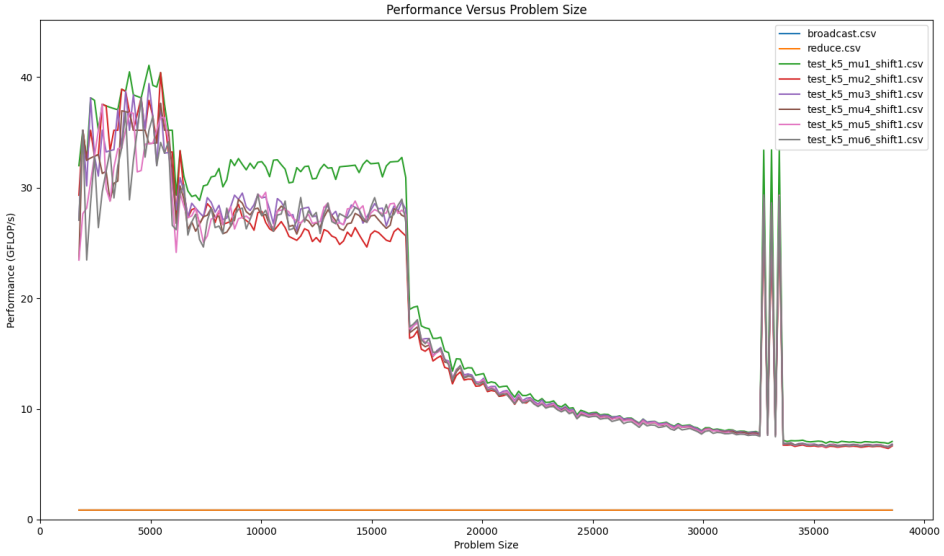

Figure A.29: Clang-Polly no threading, Intel Xeon Gold 6338, k=5, avx2

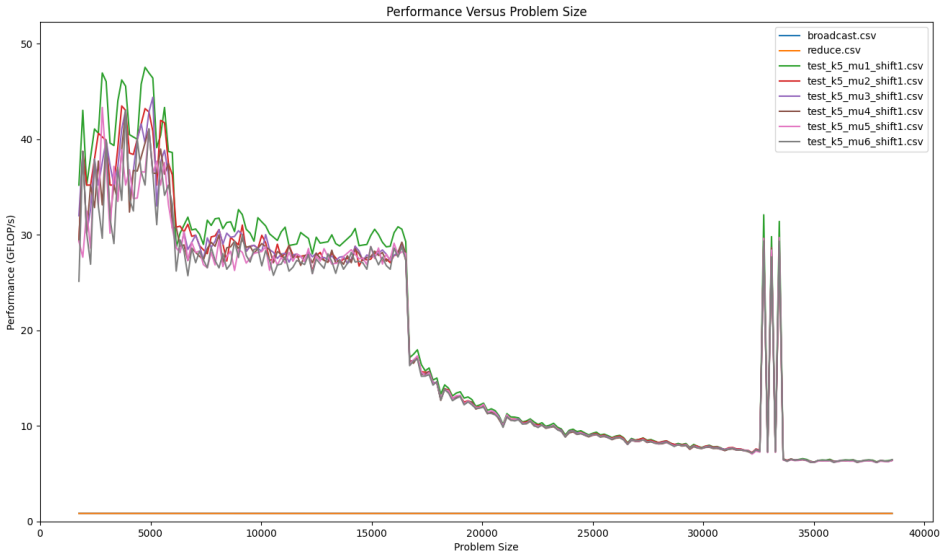

Figure A.30: Clang-Polly no threading, Intel Xeon Gold 6338, k=5, avx512

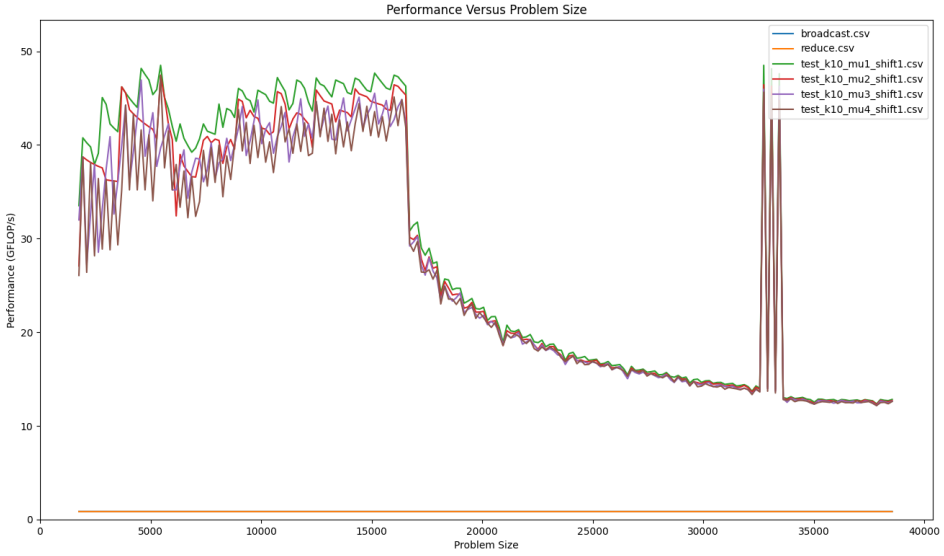

Figure A.31: Clang-Polly no threading, Intel Xeon Gold 6338, k=10, avx2

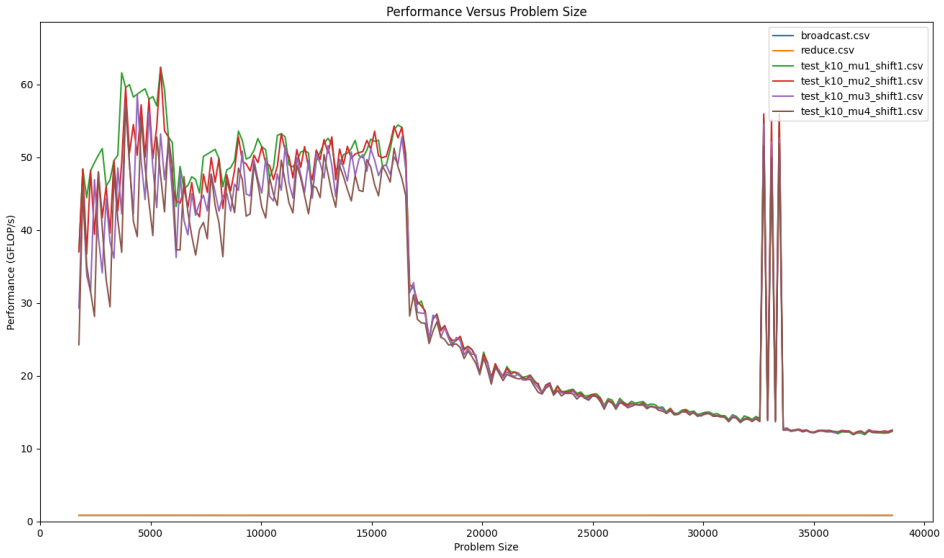

Figure A.32: Clang-Polly no threading, Intel Xeon Gold 6338, k=10, avx512

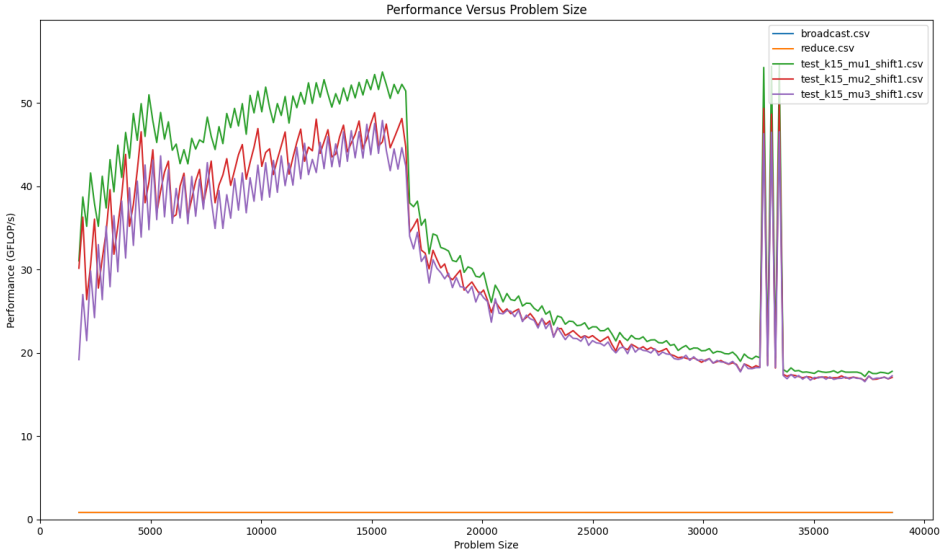

Figure A.33: Clang-Polly no threading, Intel Xeon Gold 6338, k=15, avx2

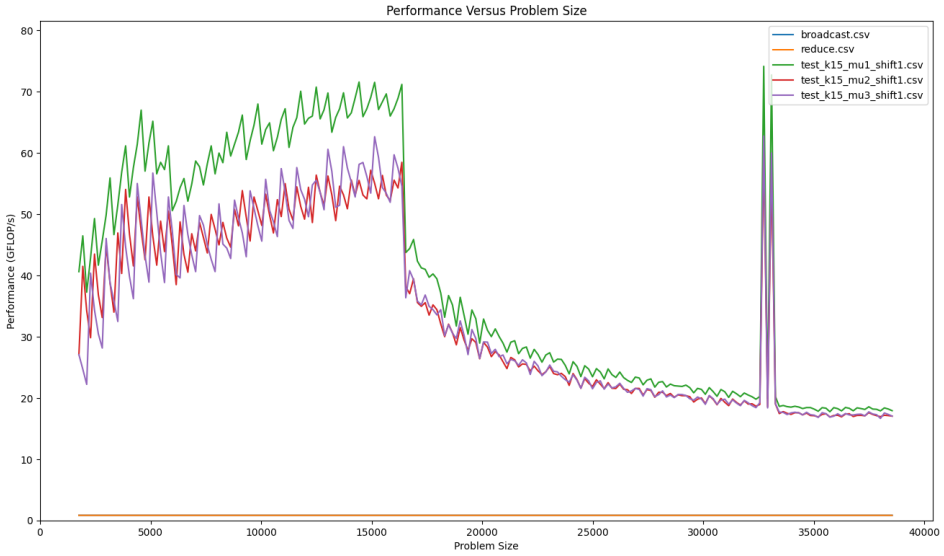

Figure A.34: Clang-Polly no threading, Intel Xeon Gold 6338, k=15, avx512

A.1.5 Threaded

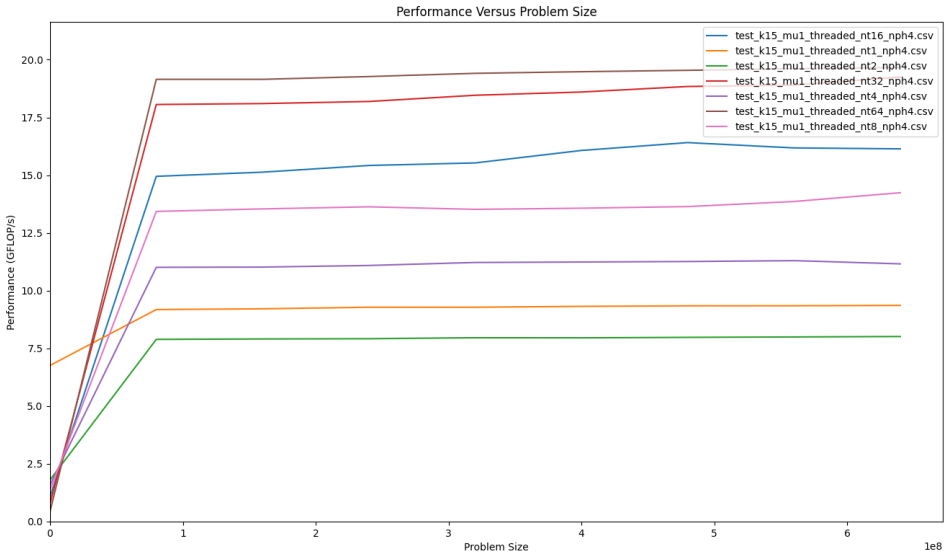

Figure A.35: GCC with threading, Intel Xeon Gold 6338, k=15

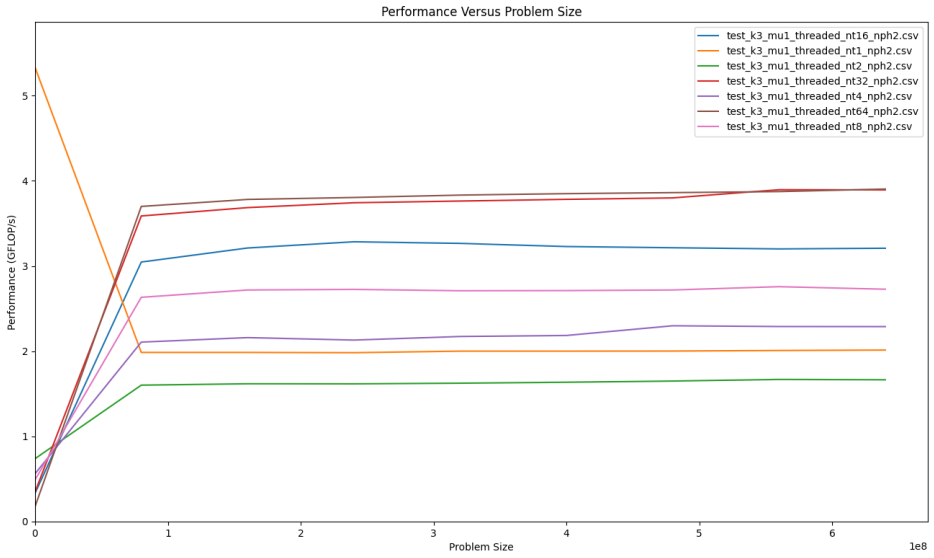

Figure A.36: GCC with threading, Intel Xeon Gold 6338, k=3

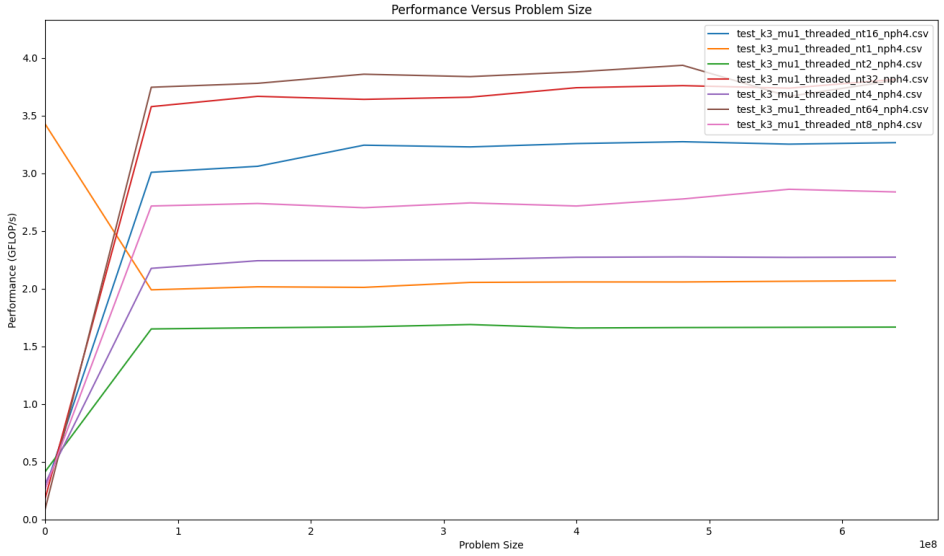

Figure A.37: GCC with threading, Intel Xeon Gold 6338, k=3

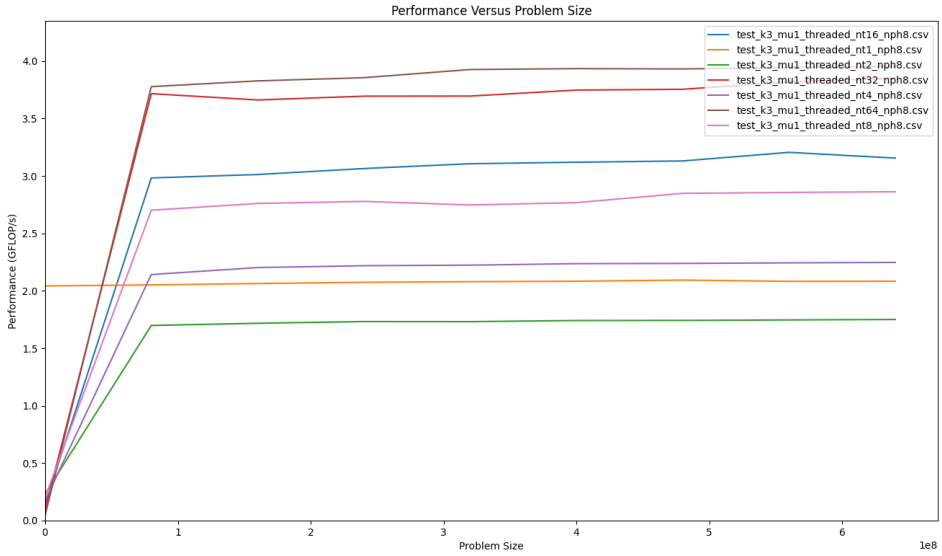

Figure A.38: GCC with threading, Intel Xeon Gold 6338, k=3

A.2 BLIS Full Code Example

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

#include "instruments.h"

#include "COMPUTE.h"

#ifndef COMPUTE_NAME
#define COMPUTE_NAME BLIS
#endif

#ifndef COMPUTE_MODEL_NAME
#define COMPUTE_MODEL_NAME BLIS_model
#endif

#include <immintrin.h>
#include <stdlib.h>

void pack_B(float* B, int rs, int cs, float* B_pack,
            int NC, int KC, int NR, int j, int p){
    int row_offset = p;
    int col_offset = j;
    for(int i = 0; i < NC*KC; i++){
        int block = i / (NR*KC);
```

```
        int row   = (i / NR) % KC + row_offset;
        int col   = i % NR + col_offset + block*NR;

        B_pack[i] = B[row*rs + col*cs];
    }
}

void pack_A(float* A, int rs, int cs, float* A_pack,
            int KC, int MC, int MR, int p, int i){
    int row_offset = i;
    int col_offset = p;
    for(int i = 0; i < KC*MC; i++){
        int block = i / (MR*KC);
        int row   = i % MR;
        int col   = (i / MR) % KC;
        A_pack[i] = A[(row_offset + block*MR + row)*rs +
            (col_offset + col)*cs];
    }
}

void kernel_16x8_crow(float* A_pack, float* B_pack, float* C,
                      int rs_c, int cs_c, int KC){
    int cldx = rs_c;
    __m512 c0 = _mm512_loadu_ps(C + 0*cldx);
    __m512 c1 = _mm512_loadu_ps(C + 1*cldx);
    __m512 c2 = _mm512_loadu_ps(C + 2*cldx);
```

```
__m512 c3 = _mm512_loadu_ps(C + 3*cldx);
__m512 c4 = _mm512_loadu_ps(C + 4*cldx);
__m512 c5 = _mm512_loadu_ps(C + 5*cldx);
__m512 c6 = _mm512_loadu_ps(C + 6*cldx);
__m512 c7 = _mm512_loadu_ps(C + 7*cldx);



for(int k = 0; k < KC; k++){
    __m512 b = _mm512_loadu_ps(B_pack + 16*k);

    __m512 a0 = _mm512_set1_ps(A_pack[k*8 + 0]);
    __m512 a1 = _mm512_set1_ps(A_pack[k*8 + 1]);
    __m512 a2 = _mm512_set1_ps(A_pack[k*8 + 2]);
    __m512 a3 = _mm512_set1_ps(A_pack[k*8 + 3]);
    __m512 a4 = _mm512_set1_ps(A_pack[k*8 + 4]);
    __m512 a5 = _mm512_set1_ps(A_pack[k*8 + 5]);
    __m512 a6 = _mm512_set1_ps(A_pack[k*8 + 6]);
    __m512 a7 = _mm512_set1_ps(A_pack[k*8 + 7]);

    c0 = _mm512_fmadd_ps(b, a0, c0);
    c1 = _mm512_fmadd_ps(b, a1, c1);
    c2 = _mm512_fmadd_ps(b, a2, c2);
    c3 = _mm512_fmadd_ps(b, a3, c3);
    c4 = _mm512_fmadd_ps(b, a4, c4);
    c5 = _mm512_fmadd_ps(b, a5, c5);
```

```
        c6 = _mm512_fmadd_ps(b, a6, c6);
        c7 = _mm512_fmadd_ps(b, a7, c7);
    }

    _mm512_storeu_ps(C + 0*cldx, c0);
    _mm512_storeu_ps(C + 1*cldx, c1);
    _mm512_storeu_ps(C + 2*cldx, c2);
    _mm512_storeu_ps(C + 3*cldx, c3);
    _mm512_storeu_ps(C + 4*cldx, c4);
    _mm512_storeu_ps(C + 5*cldx, c5);
    _mm512_storeu_ps(C + 6*cldx, c6);
    _mm512_storeu_ps(C + 7*cldx, c7);

}


void COMPUTE_NAME( op_params_t  *op_params,
           op_inputs_t  *inputs,
           op_outputs_t *outputs,
           op_inouts_t  *inouts,
           hwctx_t   *hwctx )

{
    // dimensions
    const int m0 = op_params->m0;
    const int n0 = op_params->n0;
```

```
const int k0 = op_params->k0;

// strides
const int rs_a = op_params->rs_a;
const int cs_a = op_params->cs_a;
const int rs_b = op_params->rs_b;
const int cs_b = op_params->cs_b;
const int rs_c = op_params->rs_c;
const int cs_c = op_params->cs_c;

// buffers
float *A = inputs->A_mat;
float *B = inputs->B_mat;
float *C = inouts->C_mat;

int M = m0;
int K = k0;
int N = n0;

int MC = 2048, KC = 2048, NC = 8192;
if(M < MC) MC = M;
if(K < KC) KC = K;
if(N < NC) NC = N;

int MR = 8, NR = 16;
```

```
float* B_pack=(float*)aligned_alloc(4096,
                                    sizeof(float)*KC*NC);
float* A_pack=(float*)aligned_alloc(4096,
                                    sizeof(float)*MC*KC);
// Block C, B by cols
for(int j = 0; j < N; j+=NC){
  // Block A by cols, B by rows
  for(int p = 0; p < K; p+=KC){
    // PACK B
    // fit in L3 cache
    // block dims (rows: KC, cols: NC)
    pack_B(B, rs_b, cs_b, B_pack, NC, KC, NR, j, p);

    for(int i = 0; i < M; i+=MC){ // Block C, A by rows
      // PACK A
      // fit into L2 cache
      // block dims (rows: MC, cols: KC)
      pack_A(A, rs_a, cs_a, A_pack, KC, MC, MR, p, i);

      // stride within packed B
      for(int jr = 0; jr < NC; jr+=NR){
        // stride within packed A
        for(int ir = 0; ir < MC; ir+=MR){
          // micro kernel
          // MR x NR submatrix
          kernel_16x8_crow(&A_pack[ir*KC],
```

```
                                &B_pack[jr*KC],
                                &C[(i+ir)*rs_c + (j + jr)*cs_c],
                                rs_c, cs_c, KC);

            }
          }
        }
      }
    }
}
```

A.3 BLIS Diagrams

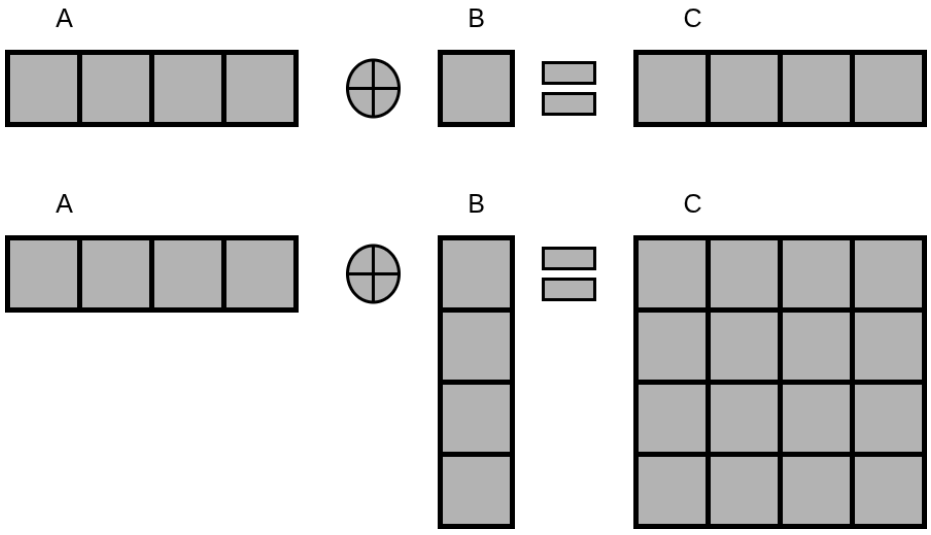

Figure A.39: Outer Product for BLIS algorithm

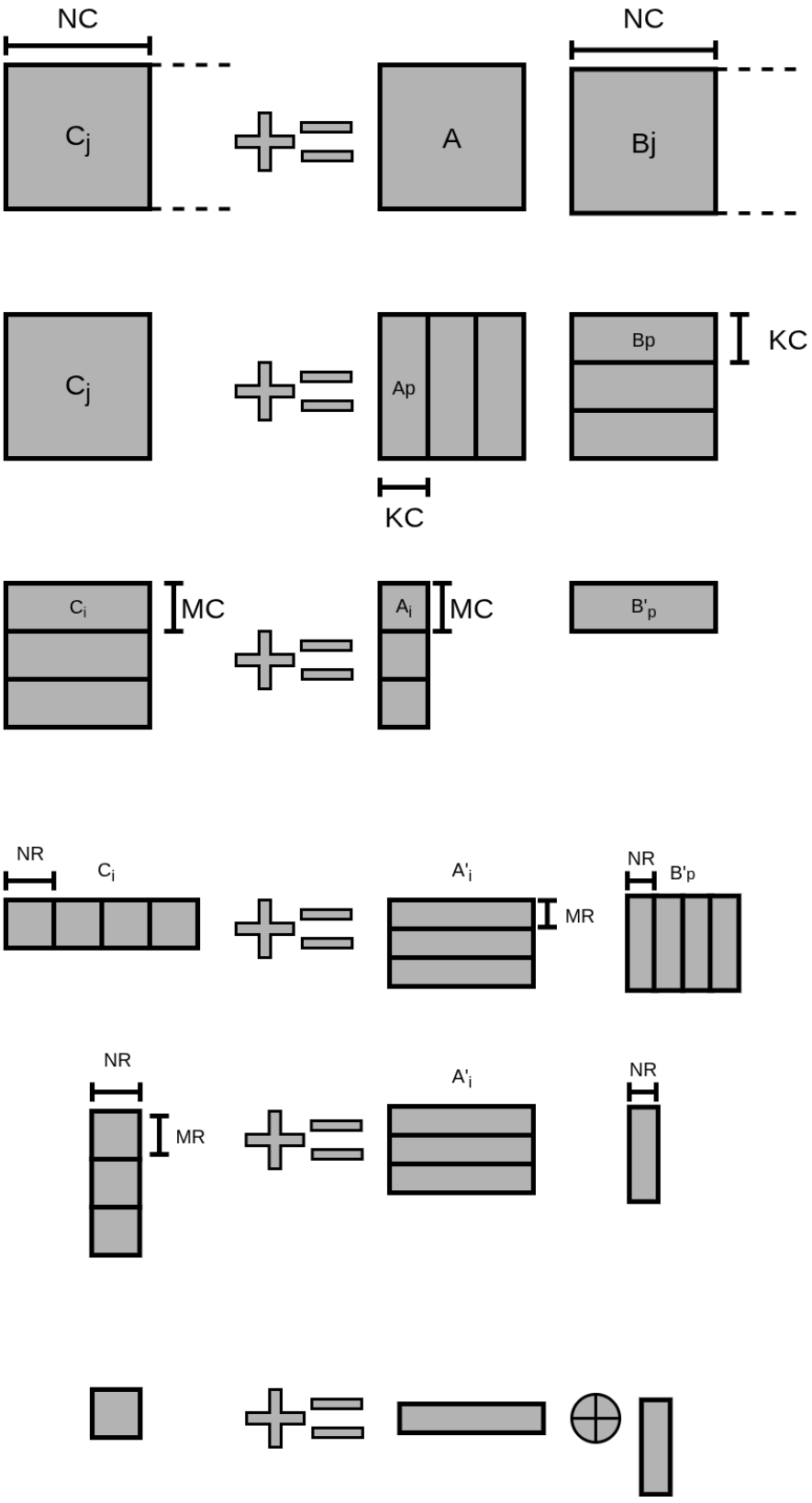
NC
NC
Cj
A
Bj
Cj
Ap
Bp
KC
KC
Ci
MC
Ai
MC
B'p
NR
Ci
A'i
NR
B'p
MR
NR
MR
A'i
NR

Figure A.40: BLIS

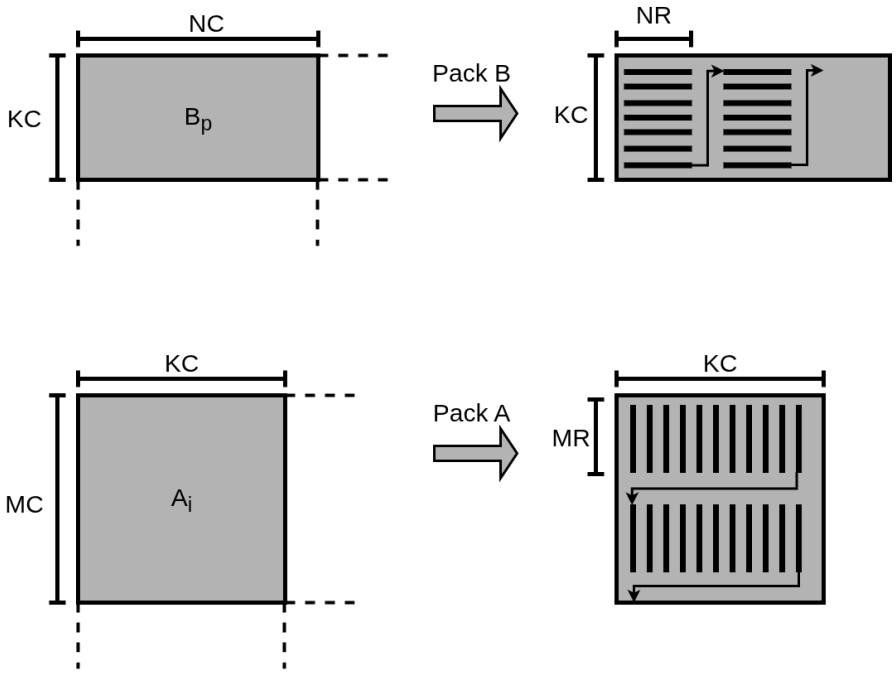

Figure A.41: BLIS Packing

A.4 Fast Kernel Full Code Example

```
#define BROAD(y, p) SIMD_FMADD(yv##y, xv, wv##p, yv##y)


#define BB3(iu, p_off, y0, y1, y2)                          \
    {                                                       \
    int bcoff = (iu)*(3) + (p_off);                         \
    int x_idx = (i0)+(bcoff);                               \
    SIMD_LOAD_D(xv, x_dlt + (x_idx)*simd_len)               \
    BROAD(y0, 2)                                            \
    BROAD(y1, 1)                                            \
    BROAD(y2, 0)                                            \
```

```
   }

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

#include "instruments.h"

#include "COMPUTE.h"

#ifndef COMPUTE_NAME
#define COMPUTE_NAME k3
#endif

#ifndef COMPUTE_MODEL_NAME
#define COMPUTE_MODEL_NAME k3_model
#endif

#include "kernel_helpers.h"

void COMPUTE_NAME( op_params_t  *op_params,
           op_inputs_t  *inputs,
           op_outputs_t *outputs,
           op_inouts_t  *inouts,
           hwctx_t   *hwctx )
{
```

```
int m0 = op_params->m0;
int k = op_params->k;
int offset = op_params->offset;
int x_stride = op_params->x_stride;
int y_stride = op_params->y_stride;
int x_len = op_params->x_len;
int y_len = op_params->y_len;

assert(x_len == m0*x_stride);
assert(y_len == m0*y_stride);

float *x = inputs->x;
float *y = outputs->y;

float *w = inputs->w;

int mu = 2;
int simd_len = SIMD_LEN;
float *x_dlt;
float *y_dlt;



dlt_in(m0, k, offset, simd_len, x, &x_dlt, &y_dlt, x_stride);

SIMD_BROADCAST(wv0, w[0])
SIMD_BROADCAST(wv1, w[1])
```

```
SIMD_BROADCAST(wv2, w[2])
SIMD_D(xv)
SIMD_D(yv0)
SIMD_D(yv1)
SIMD_D(yv2)
SIMD_D(yv3)
SIMD_D(yv4)
SIMD_D(yv5)
SIMD_D(yv6)
SIMD_D(yv7)
int rows = m0/simd_len + (k-1);
int start_of_ss = k-1;
STARTUP(y_dlt, x_dlt, w, rows, k, simd_len, start_of_ss)

int i0 = start_of_ss;

for( ; i0 < rows - mu*k; i0+=mu*k ) {
    BEGIN_INSTRUMENTATION;

    SIMD_LOAD_D(yv0, y_dlt + (i0 - 2)*simd_len)
    SIMD_LOAD_D(yv1, y_dlt + (i0 - 1)*simd_len)
    SIMD_LOAD_D(yv2, y_dlt + (i0 - 0)*simd_len)
    SIMD_LOAD_D(yv3, y_dlt + (i0 - -1)*simd_len)
    SIMD_LOAD_D(yv4, y_dlt + (i0 - -2)*simd_len)
    SIMD_LOAD_D(yv5, y_dlt + (i0 - -3)*simd_len)
    SIMD_LOAD_D(yv6, y_dlt + (i0 - -4)*simd_len)
```

```
SIMD_LOAD_D(yv7, y_dlt + (i0 - -5)*simd_len)
// iu = 0 =============

BB3(0, 0,
    0, 1, 2)
BB3(1, 0,
    3, 4, 5)

// iu = 1 =============

BB3(0, 1,
    1, 2, 3)
BB3(1, 1,
    4, 5, 6)

// iu = 2 =============

BB3(0, 2,
    2, 3, 4)
BB3(1, 2,
    5, 6, 7)

SIMD_STORE(y_dlt + (i0+(-2))*simd_len, yv0)
SIMD_STORE(y_dlt + (i0+(-1))*simd_len, yv1)
SIMD_STORE(y_dlt + (i0+(0))*simd_len, yv2)
SIMD_STORE(y_dlt + (i0+(1))*simd_len, yv3)
```

```
        SIMD_STORE(y_dlt + (i0+(2))*simd_len, yv4)
        SIMD_STORE(y_dlt + (i0+(3))*simd_len, yv5)
        SIMD_STORE(y_dlt + (i0+(4))*simd_len, yv6)
        SIMD_STORE(y_dlt + (i0+(5))*simd_len, yv7)
        END_INSTRUMENTATION;
    }

    COOLDOWN(i0, y_dlt, x_dlt, w, rows, k, simd_len, start_of_ss)

    dlt_out(m0, offset, simd_len, x_dlt, y, y_dlt, y_stride);
}
```

A.5 Fast Kernel with Shift Full Code Example

```
#define BROAD(y, p) SIMD_FMADD(yv##y, xv, wv##p, yv##y)

#define BB3(iu, p_off, y0, y1, y2)              \
    {                                           \
    int bcoff = (iu)*(3) + (p_off);             \
    int x_idx = (i0)+(bcoff);                   \
    SIMD_LOAD_D(xv, x_dlt + (x_idx)*simd_len)   \
    BROAD(y0, 2)                                \
    BROAD(y1, 1)                                \
    BROAD(y2, 0)                                \
```

```
   }

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

#include "instruments.h"

#include "COMPUTE.h"

#ifndef COMPUTE_NAME
#define COMPUTE_NAME k3
#endif

#ifndef COMPUTE_MODEL_NAME
#define COMPUTE_MODEL_NAME k3_model
#endif

#include "kernel_helpers.h"

void COMPUTE_NAME( op_params_t  *op_params,
           op_inputs_t  *inputs,
           op_outputs_t *outputs,
           op_inouts_t  *inouts,
           hwctx_t   *hwctx )
{
```

```
int m0 = op_params->m0;
int k = op_params->k;
int offset = op_params->offset;
int x_stride = op_params->x_stride;
int y_stride = op_params->y_stride;
int x_len = op_params->x_len;
int y_len = op_params->y_len;

assert(x_len == m0*x_stride);
assert(y_len == m0*y_stride);

float *x = inputs->x;
float *y = outputs->y;

float *w = inputs->w;

int mu = 2;
int simd_len = SIMD_LEN;
float *x_dlt;
float *y_dlt;

dlt_in(m0, k, offset, simd_len, x, &x_dlt, &y_dlt, x_stride);

SIMD_BROADCAST(wv0, w[0])
SIMD_BROADCAST(wv1, w[1])
SIMD_BROADCAST(wv2, w[2])
```

```
SIMD_D(xv)
SIMD_D(yv0)
SIMD_D(yv1)
SIMD_D(yv2)
SIMD_D(yv3)
SIMD_D(yv4)
SIMD_D(yv5)
SIMD_D(yv6)
SIMD_D(yv7)
int rows = m0/simd_len + (k-1);
int start_of_ss = k-1;
STARTUP(y_dlt, x_dlt, w, rows, k, simd_len, start_of_ss)

int i0 = start_of_ss;

SIMD_LOAD_D(yv0, y_dlt + (i0 - 2)*simd_len)
SIMD_LOAD_D(yv1, y_dlt + (i0 - 1)*simd_len)
for( ; i0 < rows - mu*k; i0+=mu*k ) {
    BEGIN_INSTRUMENTATION;

    // iu = 0 =============

    BB3(0, 0,
        0, 1, 2)
    BB3(1, 0,
        3, 4, 5)
```

```
// iu = 1 =============

BB3(0, 1,
    1, 2, 3)
BB3(1, 1,
    4, 5, 6)

// iu = 2 =============

BB3(0, 2,
    2, 3, 4)
BB3(1, 2,
    5, 6, 7)

SIMD_STORE(y_dlt + (i0+(-2))*simd_len, yv0)
SIMD_STORE(y_dlt + (i0+(-1))*simd_len, yv1)
SIMD_STORE(y_dlt + (i0+(0))*simd_len, yv2)
SIMD_STORE(y_dlt + (i0+(1))*simd_len, yv3)
SIMD_STORE(y_dlt + (i0+(2))*simd_len, yv4)
SIMD_STORE(y_dlt + (i0+(3))*simd_len, yv5)
yv0 = yv6;
yv1 = yv7;
ZERO(yv2)
ZERO(yv3)
ZERO(yv4)
```

```
        ZERO(yv5)
        ZERO(yv6)
        ZERO(yv7)
        END_INSTRUMENTATION;
    }

    SIMD_STORE(y_dlt + (i0-2)*simd_len, yv0)
    SIMD_STORE(y_dlt + (i0-1)*simd_len, yv1)
    COOLDOWN(i0, y_dlt, x_dlt, w, rows, k, simd_len, start_of_ss)

    dlt_out(m0, offset, simd_len, x_dlt, y, y_dlt, y_stride);
}
```

A.6 Multithreaded Kernel Full Code Example

```
#define BROAD(y, p) SIMD_FMADD(yv##y, xv, wv##p, yv##y)

#define BB3(iu, p_off, y0, y1, y2)              \
    {                                           \
    int bcoff = (iu)*(3) + (p_off);             \
    int x_idx = (i0)+(bcoff);                   \
    SIMD_LOAD_D(xv, x_dlt + (x_idx)*simd_len)   \
    BROAD(y0, 2)                                \
    BROAD(y1, 1)                                \
```

```
    BROAD(y2, 0)                                    \
   }

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

#include "instruments.h"

#include "COMPUTE.h"

#ifndef COMPUTE_NAME
#define COMPUTE_NAME k3
#endif

#ifndef COMPUTE_MODEL_NAME
#define COMPUTE_MODEL_NAME k3_model
#endif

#include "kernel_helpers.h"

void COMPUTE_MODEL_NAME( op_model_t   *model,
             op_params_t  *op_params,
             op_inputs_t  *inputs,
             op_outputs_t *outputs,
             op_inouts_t  *inouts,
```

```
            hwctx_t      *hwctx )
{
  int m0 = op_params->m0;
  int k = op_params->k;

  model->flops = 2*(k)*m0;

  model->bytes = ((3*k+1)*m0)*sizeof(float);
}

void COMPUTE_NAME( op_params_t  *op_params,
           op_inputs_t  *inputs,
           op_outputs_t *outputs,
           op_inouts_t  *inouts,
           hwctx_t   *hwctx )
{
    int m0 = op_params->m0;
    int k = op_params->k;
    int offset = op_params->offset;
    int x_stride = op_params->x_stride;
    int y_stride = op_params->y_stride;
    int x_len = op_params->x_len;
    int y_len = op_params->y_len;

    assert(x_len == m0*x_stride);
    assert(y_len == m0*y_stride);
```

```
float *x = inputs->x;
float *y = outputs->y;

float *w = inputs->w;

int mu = 1;
int simd_len = SIMD_LEN;
float *x_dlt;
float *y_dlt;

dlt_in(m0, k, offset, simd_len, x, &x_dlt, &y_dlt, x_stride);

SIMD_BROADCAST(wv0, w[0])
SIMD_BROADCAST(wv1, w[1])
SIMD_BROADCAST(wv2, w[2])
SIMD_D(xv)
SIMD_D(yv0)
SIMD_D(yv1)
SIMD_D(yv2)
SIMD_D(yv3)
SIMD_D(yv4)
int rows = m0/simd_len + (k-1);
int start_of_ss = k-1;
STARTUP(y_dlt, x_dlt, w, rows, k, simd_len, start_of_ss)
```

```
#define P 1
#define PH 2

    int B = (rows-start_of_ss-mu*k) / (mu*k*P*PH);

    for(int phase = 0; phase < PH; ++phase){
        #pragma omp parallel for num_threads(P)
        for(int pid = 0; pid < P; ++pid) {
            for(int bid = 0; bid < B; ++bid) {
                BEGIN_INSTRUMENTATION;
                //thread_offset+block_offset+phase_offset
                int i0 = start_of_ss +
                         (PH*mu*k*B*pid) +
                         (mu*k*bid) +
                         (phase*mu*k*B);

        SIMD_LOAD_D(yv0, y_dlt + (i0 - 2)*simd_len)
        SIMD_LOAD_D(yv1, y_dlt + (i0 - 1)*simd_len)
        SIMD_LOAD_D(yv2, y_dlt + (i0 - 0)*simd_len)
        SIMD_LOAD_D(yv3, y_dlt + (i0 - -1)*simd_len)
        SIMD_LOAD_D(yv4, y_dlt + (i0 - -2)*simd_len)
        // iu = 0 =============

        BB3(0, 0,
            0, 1, 2)
```

```
        // iu = 1 =============

        BB3(0, 1,
            1, 2, 3)

        // iu = 2 =============

        BB3(0, 2,
            2, 3, 4)

        SIMD_STORE(y_dlt + (i0+(-2))*simd_len, yv0)
        SIMD_STORE(y_dlt + (i0+(-1))*simd_len, yv1)
        SIMD_STORE(y_dlt + (i0+(0))*simd_len, yv2)
        SIMD_STORE(y_dlt + (i0+(1))*simd_len, yv3)
        SIMD_STORE(y_dlt + (i0+(2))*simd_len, yv4)
                END_INSTRUMENTATION;
            }
        }
    }

    int i0 = B*mu*k*P*PH + start_of_ss;

    COOLDOWN(i0, y_dlt, x_dlt, w, rows, k, simd_len, start_of_ss)

    dlt_out(m0, offset, simd_len, x_dlt, y, y_dlt, y_stride);
}
```

A.7 Fraction of Peak Tables

Table A.1: Icelake AVX512 FOP k = 1

k	μ	#reg	ilp cyc	rker	fop
1.0	1.0	3.0	1.0	1.0	N/A
1.0	2.0	4.0	2.0	2.0	N/A
1.0	3.0	5.0	3.0	3.0	N/A
1.0	4.0	6.0	4.0	4.0	N/A
1.0	5.0	7.0	5.0	5.0	1.0

Table A.2: Icelake AVX512 FOP k = 2

k	μ	#reg	ilp cyc	rker	fop
2	1	6	2.0	4.0	N/A
2	2	8	4.0	8.0	N/A
2.0	3.0	10.0	6.0	12.0	1.0
2.0	4.0	12.0	8.0	16.0	1.0
2.0	5.0	14.0	10.0	20.0	1.0

Table A.3: Icelake AVX512 FOP k = 3

k	μ	#reg	ilp cyc	rker	fop
3	1	9	3.0	9.0	N/A
3.0	2.0	12.0	6.0	18.0	1.0
3.0	3.0	15.0	9.0	27.0	1.0
3.0	4.0	18.0	12.0	36.0	1.0
3.0	5.0	21.0	15.0	45.0	1.0

Table A.4: Icelake AVX512 FOP k = 5

k	μ	#reg	ilp cyc	rker	fop
5.0	1.0	15.0	5.0	25.0	1.0
5.0	2.0	20.0	10.0	50.0	1.0
5.0	3.0	25.0	15.0	75.0	1.0
5.0	4.0	30.0	20.0	100.0	1.0
5.0	5.0	35.0	25.0	125.0	1.0

Table A.5: Icelake AVX512 FOP k = 10

k	μ	#reg	ilp cyc	rker	fop
10.0	1.0	30.0	10.0	100.0	1.0
10.0	2.0	40.0	20.0	200.0	1.0
10.0	3.0	50.0	30.0	300.0	1.0
10.0	4.0	60.0	40.0	400.0	1.0
10.0	5.0	70.0	50.0	500.0	1.0

Table A.6: Icelake AVX512 FOP k = 15

k	μ	#reg	ilp cyc	rker	fop
15.0	1.0	45.0	15.0	225.0	1.0
15.0	2.0	60.0	30.0	450.0	1.0
15.0	3.0	75.0	45.0	675.0	1.0
15.0	4.0	90.0	60.0	900.0	1.0
15.0	5.0	105.0	75.0	1125.0	1.0

Table A.7: Haswell AVX2 FOP k = 1

k	μ	#reg	ilp cyc	rker	fop
1	1	3	0.5	1.0	N/A
1	2	4	1.0	2.0	N/A
1	3	5	1.5	3.0	N/A
1	4	6	2.0	4.0	N/A
1	5	7	2.5	5.0	N/A

Table A.8: Haswell AVX2 FOP k = 2

k	μ	#reg	ilp cyc	rker	fop
2	1	6	1.0	2.0	N/A
2	2	8	2.0	4.0	N/A
2	3	10	3.0	6.0	N/A
2	4	12	4.0	8.0	N/A
2	5	14	5.0	10.0	N/A

Table A.9: Haswell AVX2 FOP k = 3

k	μ	#reg	ilp cyc	rker	fop
3	1	9	1.5	4.5	N/A
3	2	12	3.0	9.0	N/A
3	3	15	4.5	13.5	N/A
3.0	4.0	18.0	6.0	18.0	1.0
3.0	5.0	21.0	7.5	22.5	1.0

Table A.10: Haswell AVX2 FOP k = 5

k	μ	#reg	ilp cyc	rker	fop
5	1	15	2.5	12.5	N/A
5	2	20	5.0	25.0	N/A
5.0	3.0	25.0	7.5	37.5	1.0
5.0	4.0	30.0	10.0	50.0	1.0
5.0	5.0	35.0	12.5	62.5	1.0

Table A.11: Haswell AVX2 FOP k = 10

k	μ	#reg	ilp cyc	rker	fop
10	1	30	5.0	50.0	N/A
10.0	2.0	40.0	10.0	100.0	1.0
10.0	3.0	50.0	15.0	150.0	1.0
10.0	4.0	60.0	20.0	200.0	1.0
10.0	5.0	70.0	25.0	250.0	1.0

Table A.12: Haswell AVX2 FOP k = 15

k	μ	#reg	ilp cyc	rker	fop
15.0	1.0	45.0	7.5	112.5	1.0
15.0	2.0	60.0	15.0	225.0	1.0
15.0	3.0	75.0	22.5	337.5	1.0
15.0	4.0	90.0	30.0	450.0	1.0
15.0	5.0	105.0	37.5	562.5	1.0

Table A.13: Zen2 AVX2 FOP k = 1

k	μ	#reg	ilp cyc	rker	fop
1	1	3	0.5	1.0	N/A
1	2	4	1.0	2.0	N/A
1	3	5	1.5	3.0	N/A
1	4	6	2.0	4.0	N/A
1	5	7	2.5	5.0	N/A

Table A.14: Zen2 AVX2 FOP k = 2

k	μ	#reg	ilp cyc	rker	fop
2	1	6	1.0	2.0	N/A
2	2	8	2.0	4.0	N/A
2	3	10	3.0	6.0	N/A
2	4	12	4.0	8.0	N/A
2	5	14	5.0	10.0	N/A

Table A.15: Zen2 AVX2 FOP k = 3

k	μ	#reg	ilp cyc	rker	fop
3	1	9	1.5	4.5	N/A
3	2	12	3.0	9.0	N/A
3	3	15	4.5	13.5	N/A
3.0	4.0	18.0	6.0	18.0	1.0
3.0	5.0	21.0	7.5	22.5	1.0

Table A.16: Zen2 AVX2 FOP k = 5

k	μ	#reg	ilp cyc	rker	fop
5	1	15	2.5	12.5	N/A
5	2	20	5.0	25.0	N/A
5.0	3.0	25.0	7.5	37.5	1.0
5.0	4.0	30.0	10.0	50.0	1.0
5.0	5.0	35.0	12.5	62.5	1.0

Table A.17: Zen2 AVX2 FOP k = 10

k	μ	#reg	ilp cyc	rker	fop
10	1	30	5.0	50.0	N/A
10.0	2.0	40.0	10.0	100.0	1.0
10.0	3.0	50.0	15.0	150.0	1.0
10.0	4.0	60.0	20.0	200.0	1.0
10.0	5.0	70.0	25.0	250.0	1.0

Table A.18: Zen2 AVX2 FOP k = 15

k	μ	#reg	ilp cyc	rker	fop
15.0	1.0	45.0	7.5	112.5	1.0
15.0	2.0	60.0	15.0	225.0	1.0
15.0	3.0	75.0	22.5	337.5	1.0
15.0	4.0	90.0	30.0	450.0	1.0
15.0	5.0	105.0	37.5	562.5	1.0

A.8 DLT Code

```
void transpose_in(int rows_x, int cols_x, int stb, int m0,
                  float* x, float* x_dlt, int x_stride){
    for( int i = 0; i < rows_x; i++){
        for( int j = 0; j < cols_x; j++){
            int dlt_idx = j*rows_x + i;
            int x_idx = (stb + i*cols_x + j + (m0)) % (m0);
            x_dlt[dlt_idx] = x[x_idx*(x_stride)];
        }
    }
}

void dlt_in(int m0, int k, int offset, int simd_len,
            float* x, float** x_dlt,
            float** y_dlt, int x_stride) {
    int ghost_rows = (k) - 1;
    int dlt_size = (m0) + (simd_len)*ghost_rows;
    *x_dlt = calloc(dlt_size, sizeof(float));
    *y_dlt = calloc(dlt_size, sizeof(float));
    int stb = (offset);
    int rows_x = (simd_len);
    int cols_x = (m0)/(simd_len);
    /*this performs a transpose of the main chunk of the buffer*/
    transpose_in(rows_x, cols_x, stb, m0, x, *x_dlt, x_stride);
```

```
    /*handles the shift of the ghost rows at the end */
    int sgr = m0;
    for( int i = 0; i < ghost_rows; i++ ){
        for( int j = 0; j < (simd_len) - 1; j++ ){
            int dlt_idx = sgr + i*(simd_len) + j;
            int x_idx = (stb + (j+1)*cols_x + i + (m0)) % (m0);
            (*x_dlt)[dlt_idx] = x[x_idx*(x_stride)];
        }
    }
    /*places the remaining elements in the spots shifted*/
    for( int i = 0; i < ghost_rows; i++ ){
        int dlt_idx = dlt_size +
                      (-ghost_rows + i + 1)*(simd_len) -
                      1;
        int x_idx = (stb + i + (m0)) % (m0);
        (*x_dlt)[dlt_idx] = x[x_idx * (x_stride)];
    }
}

void transpose_out(int rows_x, int cols_x, int offset, int m0,
                   float* y, float* y_dlt, int y_stride){
    for( int i = 0; i < rows_x; i++){
        for( int j = 0; j < cols_x; j++){
            int dlt_idx = j*rows_x + i;
            int y_idx = (offset + i*cols_x + j + m0) % m0;
            y[y_idx*y_stride] = y_dlt[dlt_idx];
```

```
        }
    }
}

void dlt_out(int m0, int offset, int simd_len,
             float* x_dlt, float* y,
             float* y_dlt, int y_stride) {
    int rows_x = (simd_len);
    int cols_x = (m0)/(simd_len);
    transpose_out(rows_x, cols_x, offset, m0, y, y_dlt, y_stride);

    free(x_dlt);
    free(y_dlt);
}
```