Memory Hierarchy and Loop Transformations

Hadi Fawad

Abstract—This laboratory provides an initial exploration into the realm of memory hierarchy, data layout, and loop transformations, delving into their intricate connections and its impact on code efficiency.

1 Overview

1.1 What is the problem?

In this lab, we were presented with the challenge of optimizing a given set of baseline computations to improve their performance using what we've learned in class. The primary focus was on data layout, loop transformations, and memory hierarchy. We were asked to apply multiple loop transformations, such as Unswitching, Splitting, Fission, and Interchange. Beyond this, we also used various different forms of parallelism such as SIMD, OpenMP, MPI, and Instruction Level Parallelism.

The overarching problem in this lab is multifaceted, asking us how we can most effectively implement these strategies and provide the reasoning and approach within this document.

1.2 What are the pain points in this problem?

There were a multitude of possible and encountered problems in this lab, some of the more technically related problems everyone faced are listed below:

Loop Transformations: Understanding how to implement many different forms of loop transformations can prove to be an issue as each one has their own set of rules and interact differently with the given code. Ensuring that they're tailored to each specific case given by the instruction document is an issue.

Schooner Usage: Navigating the usage of the Super-Computer was definitely an issue in this lab, there was a learning curve that went beyond the conventional computing (local/gpel) environments that everyone was accustomed to. Coordinating what tasks were running on gpel vs. schooner required constant communication. Queue wait times on the schooner has also been unreliable with some queues taking much longer than others even when they're executed on the same code multiple times.

Tuning: Tuning some of the loops such as unrolling or blocking require simple trial and error rather than any other approach, this proved to be time consuming. After a certain point it would also be hard to gauge whether or not the returns were diminishing or not. Attempting this on schooner would prove to be even more demanding.

Other: There were also issues surrounding the MPI Learning curve, Parallelism, and general understanding of the project to ensure that each subtask was correctly implemented and was a more efficient approach.

1.3 How do we plan to solve this?

The plan is to utilize both lecture slides and online resources to cover our knowledge gaps and complete the assignment.

For the implementations of the 1D stencil, are going to use multiple different forms of parallelism to improve the efficiency of the operations.

1.4 How does this relate to material we covered in class?

This relates to the material that was covered in class because the correct implementations of these transformations require a detailed knowledge of how data is accessed in memory. We need to understand and be able to manipulate the C code to effectively access data in a way that will perform the best on the hardware that is available, whether that be on the supercomputer or the gpel machines.

2 Loop Transformations

2.1 Loop Unswitching

Loop switching is a programming technique that allows for different sections of code to be executed based on specific conditions. In the context of a 1D stencil operation, loop switching is often used to implement boundary conditions or handle edge cases efficiently. This technique optimizes code execution depending on the position of the loop's iterator.

Our code iterates through the i0 loop, representing positions in a one-dimensional array. The inner p0 loop processes neighboring elements to compute a result res. Loop switching occurs within the conditional statement, where it checks whether p0 + i0 is within the bounds of the array input_distributed. If it is within bounds, the code performs a standard operation by multiplying elements from input_distributed and weights_distributed, accumulating the result in res. However, if p0 + i0 goes beyond the array bounds, it switches to a wrap-around operation, ensuring that the code correctly handles boundary conditions by referencing elements at the beginning of the array.

2.2 Index Set Splitting

Index set splitting is a technique commonly used in parallel computing to divide a problem into multiple subproblems or index sets that can be processed independently and concurrently by multiple threads or processes. Each subproblem corresponds to a subset of the original data, and by splitting the index sets, it's possible to perform computations in parallel, which can lead to improved performance in multi-core or distributed systems.

Our code processes an array using two separate loops, res1 and res2, with the goal of optimizing performance through parallel execution. The key concept here is the variable loopEnd, which represents the point in the array where the index set is split.

The first loop, controlled by p0, processes elements up to loopEnd, while the second loop, also controlled by p0, handles elements beyond loopEnd. This division effectively splits the index set into two parts, allowing for parallelization of resources to calculate res1 and res2 concurrently. Once both res1 and res2 are computed for each i0, the final result, output_distributed[i0], is obtained by summing the results from both loops.

2.3 Loop Fission

Loop fission is the process of splitting a loop into multiple loops over the same index range, but each performing part of the operations of the original loop. In the context of this assignment, we've applied loop fission to the *COMPUTE NAME* function.

In the original version, each iteration of the loop computes a weighted sum of the input elements by initializing a temporary variable, then applying a weighted sum. With fission, we have essentially split the loop into two parts. In the first part, we initialize an empty array of values that are filled in a second loop. It was hypothesized that separating initialization and writing will improve cache utilization over constantly flipping between the two.

As shown in the graph below, the performance boost below supports our hypothesis caused by more efficient cache utilization.

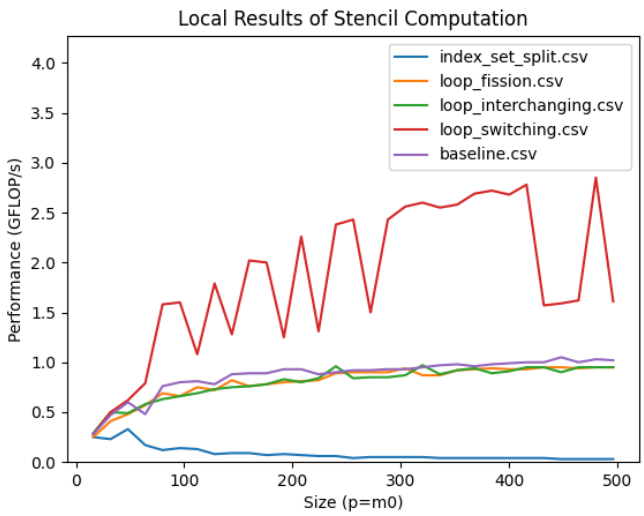
2.4 Loop Interchange

Loop interchange is an optimization technique that improves loop nesting. In the context of the stencil operation in our lab, we can apply loop interchange to the nested loops in the *COMPUTE NAME* function.

Loop interchange is an optimization technique that improves loop nesting. In the context of the stencil operation in our lab, one can apply loop interchange to the nested loops in the

In the event that $k0 \leq m0$, the initialization of *output distributed* and the summation operation will be consecutive in memory, so the inner loop's operations will benefit from better cache locality. This is supported by our graph, where each instance of loop interchanging is always at or below the performance of the baseline code.

2.5 Plot of Four Transformations



3 Code Generators

We used code generators to sweep through different factors for both Loop Unrolling and Loop Blocking. Below are the results.

3.1 Loop Unrolling

Loop unrolling is an optimization technique in computer programming and compiler design that aims to improve the execution speed of loops. It involves reducing the overhead of loop control and iteration by manually expanding (unrolling) the loop body. The goal of this lab was to create an unrolling variant that has no loops.

The code generator takes in the m0 and k0 values, and then creates a C file that is specific for both of those values. There is no "for" loops in the resulting file, but instead, the generator performs the loops and prints out the appropriate instruction, in order, to the generated file. When executed, this file will run in optimal time since there are no loops to go through, which can greatly enhance performance.

3.2 Loop Blocking

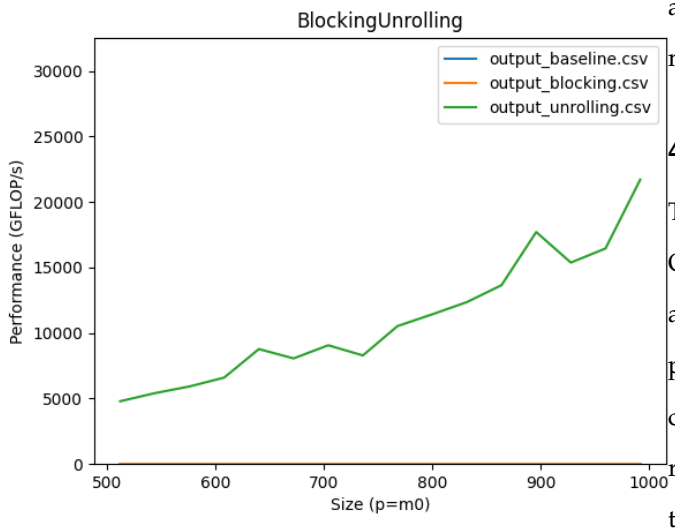
Loop blocking is a technique used to improve the performance of nested loops in computations. It involves dividing loops into smaller, cache-sized blocks, and then processing these blocks sequentially. By doing so, loop blocking enhances data locality, reducing memory access latency and improving cache efficiency.

The code generator can work with different sized blocks. The outermost loop, controlled by bi, processes these blocks iteratively, and each block encompasses a subset of the total iterations defined by block_size. I set block size to be 8 during testing, and a major limitation is that this block_size should be incremented in the same amount that m0 is. Within each block, the inner loop, controlled by i0, computes the result for individual loop iterations while considering boundary conditions through modulo arithmetic. Thus, we achieve loop blocking.

3.3 Code Generator Graph

The graph below shows the different performances of the unrolling and blocking as compared to the baseline. It is very clear that Loop Unrolling significantly outperforms the other variants; in this context, removing the for loops is a significant performance increase. I believe the reason for this is data locality; in a stencil, only the edges are not accessed

sequentially. The rest of the operations can be perfomed extremely fast without the need for a loop.



4 Parallelism

4.1 Instruction Level Parallelism

Within our code, ILP was used to perform operations in parallel. Inside the loop controlled by p0, the code performs ILP by simultaneously loading and processing four float values at a time. It does this by loading sets of four elements from input_distributed and weights_distributed, and then using SIMD instructions to perform multiplication and addition operations on these sets in parallel. This approach leverages ILP to effectively execute multiple instructions at the same time, significantly enhancing the computational efficiency by processing multiple data elements concurrently, and ultimately improving the performance of the code.

4.2 SIMD

The upgraded implementation uses SIMD operations to parallelize the computation, using AVX512 available on processors like Schooner. In the COMPUTE_NAME function, rather than computing the base operation element by element, our new version batches these calculations in groups of eight using 256-bit wide AVX2 intrinsics. This means that, the input data is loaded into AVX registers using mm256loadups, and weights are broadcasted using mm256broadcastss. These values are then multiplied and stored. The results are stored back to memory using mm256storeups.

4.3 OpenMP

The code uses MPI for distributed memory parallelism and OpenMP for shared memory parallelism to operate using a set of data. The program functions on arrays of floating-point values, specifically input, weights, and output. The computation takes place on the root node, denoted by the root_rid, which for the purpose of this code is set to 0. On this node, for each element in the input_distributed array, the code multiplies it with corresponding weights from the weights_distributed array and accumulates the results. The memory for these arrays is allocated and initialized on the root node, while other nodes are seemingly set up for potential future extensions or modifications. Memory deallocation is also handled on the root node once the computations are complete.

4.4 MPI Send & Receive

To parallelize the code without using the MPI collective library, data was manually sent to each rank.

First, the *DISTRIBUTED ALLOCATE NAME* and *DISTRIBUTED FREE NAME* functions remained mostly the same - they allocate and free the input and weights vectors as needed. We continue by dividing the input vector and distribute that and the full weighted vector to each rank.

The computation is then split among multiple MPI processes. The workload is divided evenly so that each process calculates a portion of the stencil operation. Halo cells are used for communication between processes to handle boundary conditions. We've done this because each process must have the necessary data to compute the stencil operation near the chunk boundaries, otherwise we'll get confusing segmentation faults. Once the computation is

complete, each rank sends its chunk of the output array back to the root rank, which assembles them back into the full output array.

Once the computation is complete, the root rank first copies its own chunk of the output data into the correct position in the output sequential array. Each rank sends its chunk of the output array back to the root rank. When this happens, the root rank enters a loop where it calls *MPI Recv* once for each of the other ranks. These calls are blocking and sequential, meaning the root rank waits for each *MPI Recv* to complete before proceeding to the next. This process is done one rank at a time.

At this point, while we have the framework for these operations, the test cases are not passing. This is because of a bug in our program that causes only the first 3 ranks to be accessed and written to, thus causing the fourth 'chunk' of the stencil operation to always be empty.

4.5 MPI Collective Communications

Like the above implementation, we used parallelism techniques from MPI. However, we did not limit ourselves to using only sends and receives. In the code, there are also references to . It was hypothesized that these libraries will provide a significant boost over our manual implementation in the previous section, since these libraries are carefully optimized.

We start by allocating memory to the input, output, and weights, just as we did in the above example. We use scatterv and broadcast to separate the data into blocks and distribute them from the rank node to the other nodes. From there, each process performs the computation (as seen in *COMPUTE NAME*) and gathers said computations using gatherv to then be combined. Unlike in our naive approach for gathering data in the above method, MPI's gathering algorithm will sometimes perform this in parallel.

Our program ran into a few problems, which we hypothesize are due to memory leaks. To debug this, the following Valgrind test were run:

mpiexec -n 4 valgrind –leak-check=full –track-origins=yes –trace-children=yes ./mpiCollective.x

(The full output can be found on our Gitlab submission. The file also includes the failing tests before memory leaks occured.)

Give the output from Valgrind, it was determined that there are a few memory leaks within the program. Almost all of them point to one of three areas - pthread creation, memory leaks from MPI libraries, and early deallocation in memory that point to MPI and MPIx libraries.

5 Combining Parallelism

We have implemented many different forms of parallelism in this lab, but up until now, they have been performed seperately. To achieve optimal results, combining different forms of parallelism to optimize the performance of a 1D stencil. The below sections describe how we implemented 2, 3, and 4 forms of parallelism together and how they performed in contrast to each other.

5.1 2 Forms

Our code leverages OpenMP parallelism with the

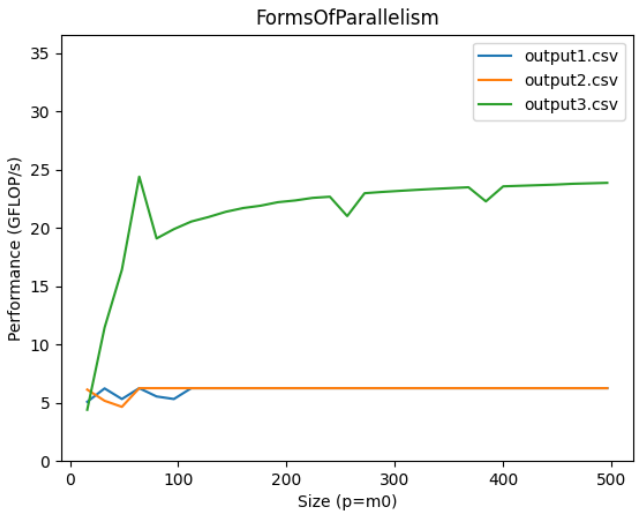
```
#pragma omp parallel
```

for directive to parallelize the outer loop, allowing multiple threads to concurrently process different iterations of the loop within a single node. This shared-memory parallelism optimizes the computational efficiency on a local level. Second, the code utilizes MPI for distributed-memory parallelism. While the OpenMP parallelism enhances performance within a single node, MPI ensures synchronization and coordination across all processes before continuing, making it suitable for distributed computing scenarios.

5.2 3 Forms

For three forms, leveraging shared memory parallelism through OpenMP by again using the parallel directive was

crucial. It also uses ILP is achieved within the inner loop by executing element-wise multiplications and additions simultaneously, capitalizing on the CPU's capacity for parallel instruction execution. Additionally, the code demonstrates data parallelism in the outer loop, processing various elements of the input and output arrays independently and in parallel. These combined forms of parallelism optimize the code's performance, particularly on multi-core architectures such as the supercomputer.



5.3 4 Forms

For our 4 forms of paralleism, employing SIMD parallelism by using SSE instructions to perform vectorized operations, Data Parallelism through concurrent processing of independent data elements within the outer loop, Message Passing Parallelism via MPI for distributing and aggregating results across multiple ranks or nodes, and implicit Loop Parallelism, where multiple iterations of the outer loop can be executed concurrently. These parallelism techniques collectively enhance computational efficiency, making efficient use of vectorized instructions, distributed computing resources, and concurrent loop processing.

5.4 Graph of Combining Parallelism

In the following graph, output1 is 2 forms of parallelism, output2 is 3 forms, and output3 is 4 forms. It can clearly be seen that combining all 4 forms of parallelism significantly outperform all others.