Lab02: Viva la Convolution

Team 03: Chase Cummins, Mark Cunningham, Lana Nguyen

***Abstract*—This lab report explores the use of parallelism in rearranging loops and determining data layouts. Our results find that parallelism and SIMD instructions yield a formidable return on organizing and processing data. Through creating multiple variants that incorporate several of these requirements, we are able to compare and optimize the performance of different operations.**

I. Loop Transformations

A. Loop Unswitching/Index Set Splitting

To remove the modulo operator, loop unswitching and index set splitting were used to rearrange the loop. In our SIMD variant, we were able to split the computation into two segments with loop unswitching, the first loop handling parallelization using SIMD instructions, and the other handling sequential processing on remaining elements where SIMD instructions may be less efficient (typically when element size is small enough). Index set splitting handles overflow elements where 'i' is greater than the threshold. Both techniques ensure that the SIMD instructions are executed where they are most effective, otherwise using sequential processing where SIMD parallelism is not feasible or efficient. Additionally, we created a variant demonstrating loop fission and loop interchange to see how it would affect the performance of the code. Loop interchange can be applied to change the order of nested loops for better cache performance or parallelization.

B. Loop Unrolling

Loop unrolling reduces the loop overhead and allows for better use of instruction level parallelism through multiple loop iterations. UNROLL_FACTOR specifies how many loop iterations are processed in each unrolled loop body and increases the instruction-level parallelism since the compiler can optimize the code for a specific number of iterations. In our implementation, we have unrolled the inner loop to process a variable amount of elements at a time. The performance of the variant at different loop unrolling factor values is pictured under Figure 1.

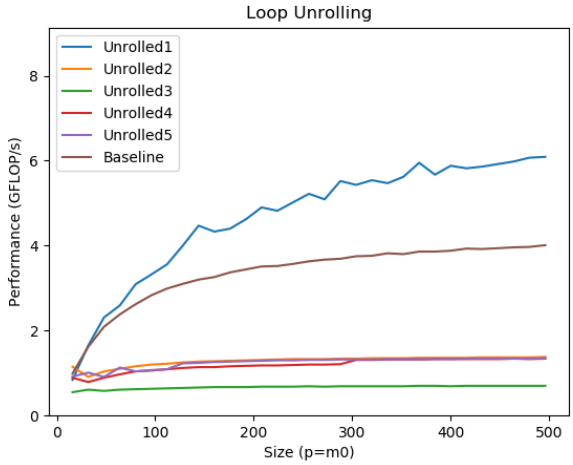

Fig. 1. Performance plot for varying unrolling factor size

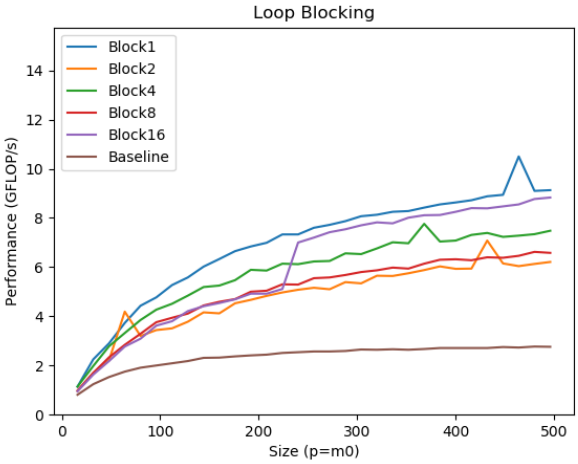

Fig. 2. Performance plot for varying block sizes

C. Loop Blocking

We created a variant to divide the main loop into blocks, breaking down the iteration space. Data accessed within a block is able to be reused, therefore reducing the number of cache misses. The computation is organized into an outer loop over i0, iterating over the input and output vectors in blocks of specified (block_size). Additionally in this variant, the circular boundary condition (to avoid array index out-of-bounds) has logic to adjust the input_idx greater than the array bounds. If the index exceeds m0 - 1, it wraps around to 0, to make sure the stencil operation considers the circular boundary properly. Under Figure 2, we can see the performance at different block sizes, and the implementation seems to overall work best at block size = 1. Regardless of block size, there is a noticeable performance improvement compared to the baseline code.

II. Parallelism

A. Instruction Level Parallelism

Instruction level parallelism already exists within the code through the use of pipelining. Multiple instructions/operations are performed in a single cycle as the processor already

implicitly and automatically leverages pipeline stages (Fetch, Decode, Execute, Memory access, and Writeback).

B. SIMD

Single instruction multiple data (SIMD) allows the program to perform the same operation on multiple elements simultaneously. Since most of the stencil operation does not require wrapping elements due to overflow, SIMD can be applied. In our implementation, we used SIMD when we could guarantee that there would not be overflow. The threshold at which overflow would occur is the weights vector size subtracted from the total array size.

There are a few differences between our base implementation that removed the modulo operator and our SIMD implementation. Since AVX2 allows vectors with size of eight, it follows that we would be traversing the outer loop in a step size of eight. To prevent beginning an iteration where an overflow would occur or an out of bounds access was attempted, it was necessary to round down threshold value to the nearest multiple of eight. This means that when in the input size was less than eight, the program would fall back to using our base implementation.

The inner loop start and end was left the same, but we made use of the fused multiply and add function to update the value of ”res”. Since eight elements are handled a time, the weights vector also needed to be eight elements composed of the same value based on the current iteration of the inner for loop. We created an array of __mm256 weights before entering any for loops, where each element in the array contains the corresponding value from the input weights vector repeated eight times. This was done to avoid created temporary vectors in the innermost for loop that would only be created and destroyed again and again.

Any elements past the index threshold mentioned earlier are computed using our base implementation involving loop splitting and unswitching.

C. OpenMP

Adding the *parallel for* OpenMP directive to the loop splitting and unswitching implementation improved performance by a modest amount for thread counts one, two, and four, but eight threads seemed to be too much. There was an odd drop off that occurred at size 1500 when the thread count was two that we are unable to explain. Four threads seemed to have both the best performance and consistency.

D. MPI Send & Receive

We used MPI to achieve Distributed Memory Parallelism for the 1D stencil operation. MPI (message passing interface) allows a programmer to write code to be executed across multiple cores on a computer or multiple computers on a network. Each core or computer is called a node, and distributed memory parallelism works by running an instance of the program across each node in the cluster. The two most basic functions available to a programmer using MPI are *send* and *receive*. Since MPI is literally message passing,

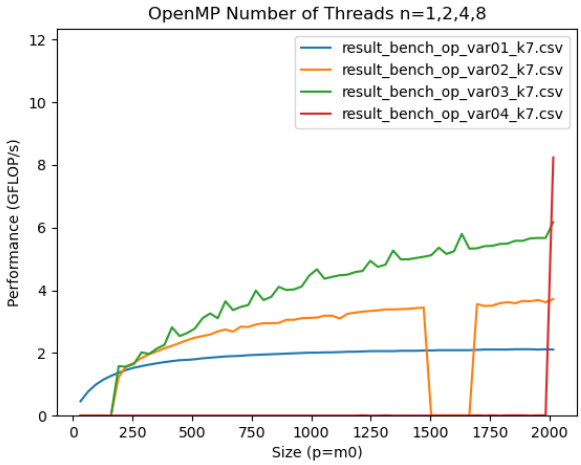

Fig. 3. Performance plot for OpenMP n threads

this makes sense. One node sends some type of message to another node which then receives the message. All additional functionality given by MPI is built upon these two functions. We used the send and receive functions to distribute the data over the nodes available on a given system, and each node would only compute the stencil operation on its portion of the input. This allows us to break up the amount of work w to be done across n nodes. In a perfect world, this would lead to a $\frac{w}{n}$ running time, but given that we need to distribute the data and gather it all back, we do not get that ideal performance boost. Additionally, we could not get our code running properly on the supercomputer so the performance plots shown are from the GPEL machines which are not as consistent from run to run. We did incorporate the compute code from our loop unswitching and index set splitting variant to give us some more performance. As you can see in Figure 4, the performance was similar across 2, 4, 8, and 16 nodes up until about 250 for the data size. Somewhat surprisingly, 4 nodes outperformed all the other sizes of nodes pretty consistently from 250 on. $n = 2$ peaked at around 5 GFLOP/s and $n = 4$ peaked around 7 GFLOP/s which is substantial increase. Disappointingly, $n = 8$ and $n = 16$ performed on average the same as $n = 2$ showing that the size of the array might have been two small to effectively split across nodes. If we had been able to run it effectively on the supercomputer, we might see a different outcome given that we would have exclusive access to the nodes as our program was running. We also were unable to scatter the input array effectively because of the overlap of data needed for a stencil operation, therefore we were stuck with sending the whole array to each node which decreased the performance.

E. MPI Collective Communications

The basis of MPI is the send and receive functions, but MPI also defines functions that make the programming easier such as broadcast and gather. These functions are not just convenient for a programmer achieving distributed memory

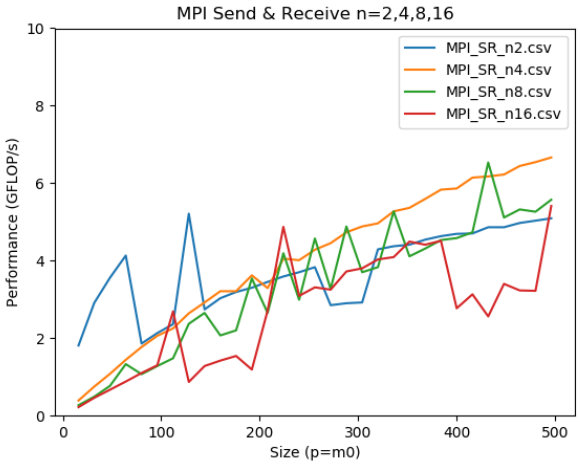

Fig. 4. Performance plot for MPI Send and Receive with n nodes

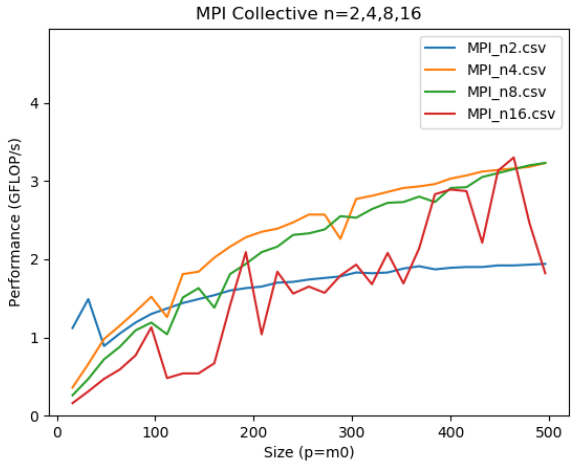

Fig. 5. Performance plot for MPI Collective Communications with n nodes

parallelism, but also utilize nice tricks such as tree-based communication for broadcast which distributes the data in much less time compared to one node sending the data to each other node. We generated two variants utilizing the MPI collective communications: one where the compute function was just a copy of the baseline, and another where we used our loop unswitching and index set splitting variant's compute function to speed up the computation. The plots are shown in Figure 5 and Figure 6 respectively. In these plots we can see that 4 nodes was generally the best overall still, although with v2 8 nodes took over at the end near size 450. There is again a lot of variance and spikes in the performance plots which we contribute to the inconsistency of resources on the GPEL machines. This may also account for 8 nodes taking over as fastest in v2. From these two plots though, we can see the increase in performance from the two loop transforms that were performed in v2 which led to about a doubling of performance. This is a far greater jump than adding additional nodes which leads us to believe that speeding up the basic computation itself is the most effective way to improve performance rather than just trying to parallelize slower code.

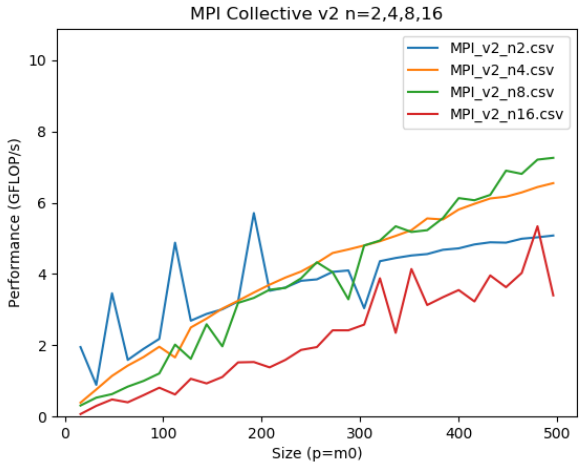

Fig. 6. Performance plot for MPI Collective Communications with n nodes combined with index set splitting and loop unswitching

III. Combining Multiple Forms of Parallelism

A. Two Forms

We used SIMD and MPI Collective Communication to create our implementation of two forms of parallelism. Since our SIMD variant iterates in step sizes of eight, it made sense to split the array into multiples of eight amongst the processes. To do this, we created a function that returns the number of elements a given process should receive. The function fills each process with eight elements before moving on to the next one, repeating the process until the input size is empty.

The COMPUTE_NAME function is largely the same except that it uses offsets determined by the rank of the process and size of array the process received. The distribute data function was left unchanged from our MPI Collective variant. The distributed allocate function was modified so that non-root processes would only allocate memory for the output_distributed based on the number it received from the function mentioned earlier. The collect data function was modified to use MPI_GatherV instead of MPI_Gather. This allowed us to specify the array size the root process would receive from all other processes, along with the offset within the array for each process.

As shown in Figure 7, our Two-Form variant completely dominated all other variants in this lab.

B. Three Forms

We added OpenMP to our Two Form implementation to achieve three forms of parallelism. However, the performance is worse when compared to the Two Form. In fact, using one thread achieved the best performance, further demonstrated

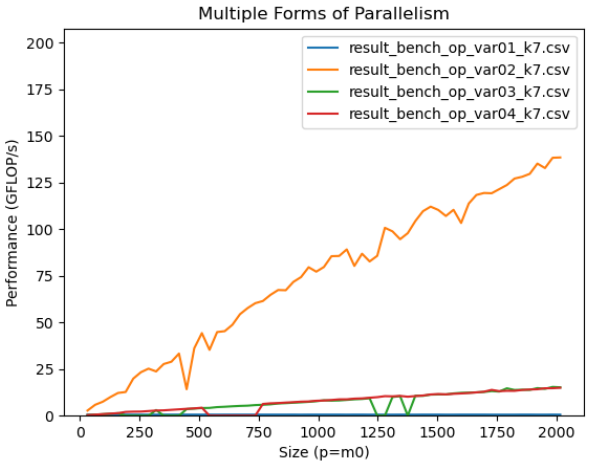

Fig. 7. Performance plot for Multiple Forms of Parallelism

how using shared memory parallelism was was unable to contribute any meaningful performance benefit.

The additional overhead of creating, managing, and destroying threads simply out-weighted any of the benefits of shared memory parallelism. Using step sizes of eight likely contributed this, since the compiler directives are often very conservative and have to take the worst case approach, meaning that the larger step size limited the space in which OpenMP could work with.

C. *Four Forms*

The Four Form variant is the same as the Three Form variant since Instruction Level Parallelism happens without any of our doing. As long as sequential instructions are not dependent, the processor will use pipe-lining to execute multiple instructions simultaneously given that the instruction throughput is greater than 1. This will be true in the 1D stencil operation being computed since each output element is not dependent on any of the other output elements.

IV. Conclusion

The performance achieved by combining SIMD and MPI was unmatched by any other variant. It was unable to be improved upon by adding OpenMP, but took inspiration from the blocking variants by using a block size of eight and distributing elements to processes in blocks of eight when possible.