Raman Srivastava

04/23/2020

CS6376.501

Program 5: 1D Filtering Report

1.) Problem Statement

The problem was to filter a data file (data.bin) a collection of traces with a 1-Dimensional (1D) filter provided (filt.bin). The requirements were that the 1D filter would be 1024 elements in length and the data is a value of 16384 elements with 1024 data arrays, padded with 1024 elements per each 1024 array. After creating the filter, the task was to run the program with MPI and OpenACC, and compare the running time of the serial execution (1 MPI node and 1 OMP thread) on the Bridges server with GPU-accelerated parallel executions of 1 Nvidia P100 GPU and compare it to 2, 3, and 4 MPI nodes with the same OpenACC setup. Parallelism would be achieved by adding importing and implementing both MPI and OpenACC on the serial code, with all four environments using the same code, compiled with the <mpicc> compiler. The bonus, was to scale Open ACC on multiple GPUs, referred to as “Scalable Link Interface” for NVidia GPUs like the P100.

2.) Approach

Since nearly all of the code was already written in the previous assignments, the main task was to implement it in such a way to be able to test the performance in execution between serial code and GPU-accelerated parallelism speedup using MPI and OpenACC. The only relevant information that the program needed to output was elapsed execution time of the 1-D filtering kernel and the total, overall execution time of the entire program, so no user arguments were added. Execution time was obtained with the Wtime() function, provided, which computes the current time based on the CPU cycle and returns it as a double value. Thus, a Wtime() function is called before program starts and after the program ends in the main function to calculated the overall time. For computing elapsed time, a Wtime() is added before and after the 1-D filtering, represented by the triple loops. In order to achieve parallelism with OpenACC, a ‘#pragma acc kernels’ statement was placed before the triple (nested) loop that computes the output array from the data and filter arrays.. This #pragma statement is ignored when the C program is compiled without OpenACC, such as using gcc or pgcc, but when it is compiled with OpenACC (pgcc –acc), GPUs are available and used at the execution time on the #pragma acc loops. A ‘#pragma acc data copyin(filterArray, dataArray) copyout(outputArray)’ is used to tell the GPUs that the filter and data arrays are the copied inputs, and the output array should be copied back to the CPU. MPI nodes are specified at runtime, and each node is given its own copy of the filter array, as well as an equal portion of the Data array traces to be calculate the portion of the Output array independently of each other. These portions are then sent to back to the master node to be combined into one complete Output array, available to be used later if desired.

3.) Solution

Based on the 1D filtering code given, this is how the program executes:

0.) For SLI (multiple NVidia GPUs), check that there are NVidia GPUs detected. If there is at least 1 NVidia, assign each GPU to OpenACC via the MPI Nodes available. The formula is processID % number of NVidia GPUs to saturate all GPUs to at least 1 MPI node. If no NVidia GPUs are detected, assign the GPU detected (if one exists) to all MPI Node without SLI scaling.

1.) Using Wtime(), the start time of the program is obtained and stored.

2.) The filter array is created, allocating a space of 1024 (filter length)\*4 bytes per float. The data array is created, allocating a space of 1024 (trace count) \* 17408 (16384 for trace length + 1024 for the 0 pads) \* 4 bytes per float. The output array is created, allocating a space of 1024 (trace count) \* 16384 (trace length) \* 4 bytes per float.

3.) The 1D filter file (filt.bin) is read to the filter array. The data file (data.bin) is also fed into the data array with a loop that seeks the corresponding place in the file where the desired element is located, and is then placed into the correct element of the data array, after the 0 pads for each partition (if any) corresponding to the process. Similarly, the result file (rslt.bin) is fed into the result array with a loop for the partition (if any) corresponding to the process, except that it’s not 0 padded. The lower bound of data traces an MPI node handles is determined by 1024 / processID, while the upper bound is determined by 1024 / (processID + 1). For example, with 4 MPI nodes in place, MPI Node 0 (master node) handles traces 0 to 255, while MPI Node 1 handles traces 256 to 511 and so on and so forth.

4.) Then, the outputs of the 1D filter on the data array is created into the output array. This is also where the directive will operate on using the #pragma statements with OpenACC. This uses a triple (nested) loop that iterates k from 0 to the filter length of 1024, j from 0 to the trace length of 16384, and i from the lower bound trace count of the MPI Node computed earlier to the Upper Bound. In every iteration of the loop, the output multidimensional array at row j, column i gets incremented by the filter array at element k, and the data array at row j, column i plus k. Wtime() is recorded before and after this triple (nested) loops to obtain the Elapsed time.

5.) Using Wtime() again, the end time of the program is obtained and stored. Then, the master process waits to receive (MPI\_Recv) the other processes.

For example, with 4 MPI Nodes, the master process waits to receive (MPI\_Recv) the other quarters of the output array array (256-511, 512-767, and 768-1023 for trace count), and combines them all into one output array.

6.) The other processes each send their portion of the filtered output array computed from their portion of the data array using MPI\_Send(). For example, with 4 MPI Nodes, the other 3 processes each feed a quarter of the output array, with process 1 sending the quarter at trace count 256-511, process 2 sending at 512-767, and process 3 sending 768-1023. Each process computes their assigned quarter of the output array though the 1D filter on the data array, respectively. Once the filtering is complete, they send their portion with MPI\_Send to the master node/process, at 0.

7.) The Overall time is calculated by subtracting the start time from the end time. This Elapsed time and the Overall time is output to the terminal and is the only thing outputted by the program. Time is only outputted if the running total of the result array and the output array is less than 1 e-20, or practically 0 (they are the same).

Proof of compile and execution output is described in the figures found on the next page.

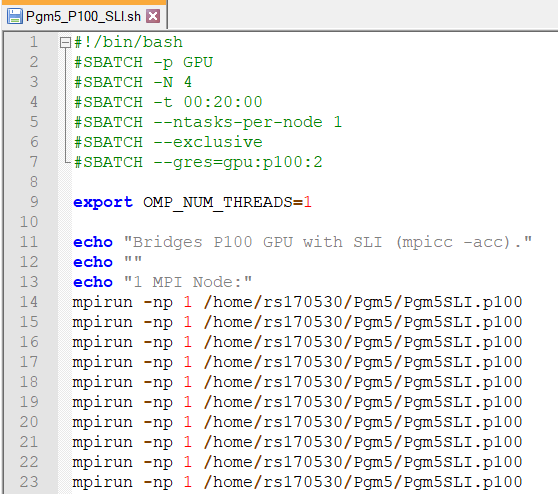


Figure 1: The Pgm5\_P100\_SLI.sh Bash Script

This is the script that allows the code to be run on the server using the SLURM command “sbatch ./<Script Name>. There are 2 Batch scripts, Pgm5\_P100, and Pgm5\_P100\_SLI, used for each of the environments. They each handle Pgm5.c (no multiple GPU support) and Pgm5SLI.c with multiple GPU support. Since it is a bash script few spaces are allowed. The stack size is made unlimited first as otherwise it would overflow. The “export OMP\_NUM\_THREADS=1” is used to limit the number of threads to the value of 1 (no OMP). Each program is called 10 times for multiple trials in the speedup analysis with 1, 2, 3, and 4 MPI nodes.

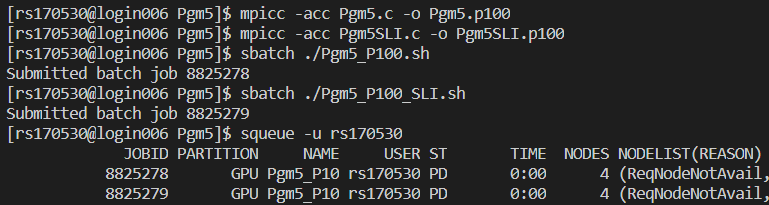


Figure 2: Compiling and Executing the Code on the Server.

These were the commands used to compile and run the program Pgm4.c on the respective server. The Pgm5.p100 was the code compiled without multiple GPU support. The Pgm5SLI.c supports multiple GPUs, and the compiled programs both use MPI and OpenACC, compiled with “mpicc –acc”. Each script uses one of the 2 compiled programs to yield the respective outputs for 1, 2, 3, and 4 MPI nodes.

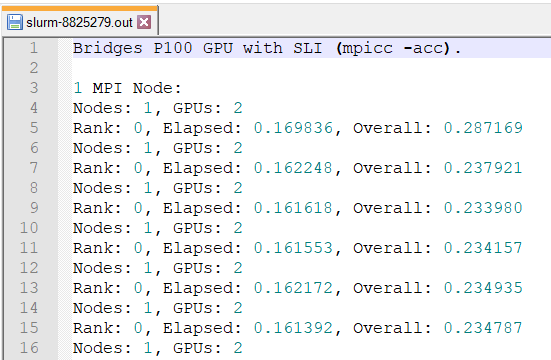


Figure 3: The Output for Batch Job 8825279, using the NVidia P100 Bridges GPU with SLI support.

Each run is proceeded by an output stating the number of MPI nodes and GPUs detected. Each line after shows the execution time for the given program that was run from the Pgm5\_P100\_SLI script, with the Elapsed and Overall times for each Node. The other script, Pgm5\_P100.sh are in the same format as Figure 3, and will follow the same guide, but with only the MPI Nodes stated since it does not have multiple GPU support. The first line tells of the environment the outputs are computed from, in this case the ‘Bridges GPU P100 with SLI’, using the ‘mpicc –acc’ compiler. The following lines are the results from each runs, with the Elapsed Times and the Overall Times labeled shown.

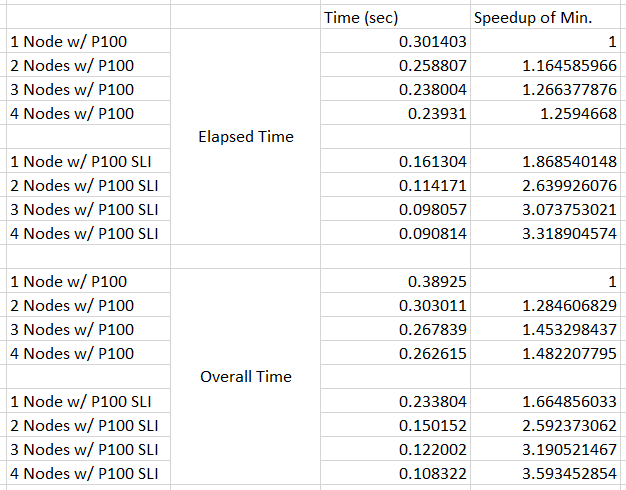
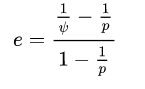


Figure 4: Speedup Analysis of the Outputs

Above is an excel sheet with data showing the speedup in performance of OpenACC on the Bridges K80 and P100 GPU. The control is the serial program that is limited to 1 MPI Node and 1 OMP thread (no OpenMP) and is represented by the row of the Elapsed and Overall time, the “1 Node w/ P100” row. The rows with ‘SLI’ represents the multiple GPU scaling within OpenACC, using 2 P100 GPUs.

The average speedup for each column is calculated by the average execution time of the serial program (1 Node w/ P100), divided by the average execution time of the respective row. Thus, the speedup of the serial program serves as the control with 1, shown in the ‘1 Node w/ P100’ row. The speedups are determined by the minimum times for each respective environment, determined after running each environment 10 times, for the sake of reducing invariability.



Using the Karp-Flatt Metric equation above, where ‘e’ represents the serial fraction, ‘ψ’ represents the speedup, and ‘p’ represents the number of processes, where p > 1. Using the Minimum speedups for 2, 3, and 4 MPI nodes computed above, the serial fraction, ‘e’ is:

|  |  |  |
| --- | --- | --- |
| Nodes (p) | Min Speedup (ψ) | Serial Fraction (e) |
| 1 | 1 | N/A (p > 1) |
| 2 | 1.164585966 | 0.71734853 |
| 3 | 1.266377876 | 0.68438705 |
| 4 | 1.2594668 | 0.72531572 |

The average serial fraction is 0.7090171, or about 70.09% across all the 4 workflows. Thus, the program is ‘predicted’ by the Karp-Flatt metric to be roughly 30.91% parallel. However, this metric does not account for the reduction in Parallelism already achieved by the OpenACC threads on the P100 GPUs. This metric is calculated after the OpenACC threads have parallelized and solved the Kernel (Elapsed time). Unfortunately, OpenACC threads could NOT be used as the node counts since OpenACC is designed to saturate as much as the GPU core allows. Thus, this Karp-Flatt metric shows the shows 30.91% parallel AFTER parallelization already performed by OpenACC, as well as the overhead to create the OpenACC threads. Or rather, the MPI nodes are not being fully utilized since the majority of parallelization is done by OpenACC on 1 GPU already.

Since the requirements specified the “speedups based on the minimum times”, it can be found as the last column of Figure 4, labeled “Speedup of Min.”. Based on the “Performance Criteria” document given in the instructions, the speedups found conform to those of the requirements, and shows the capabilities of MPI with OpenACC as a platform, especially when multiple GPUs are utilized correctly in OpenACC.