Second Individual Report

João Mena
s223186@dtu.dk
Technical University of Denmark
Lyngby, Denmark

1 EXERCISE 4: INTRODUCTION TO OPENMP

In this exercise we used the same *calc* code that was used in exercise 3 and worked on implementing the parallelizations proposed on our answer to the previous exercise in order to analyze the improvements in performance and calculate the speedup and efficiency for different numbers of threads used.

1.1 Writing a parallel version of *calc*

As mentioned in the previous report, there were two sections of the code that could be parallelized. In this exercise, those parallelizations were implemented with OpenMP [1] using shared memory. For this, I implemented two parallel regions with the directive *pragma omp parallel* for before the two respective for cycles that are meant to be optimized. This line combines the "*omp parallel*" and "*omp* for" directives, the first of which requests the use of multiple threads, and the later orders OpenMP to divide the for cycle iterations between the available number of threads. By default, the directives used apply synchronization, meaning the execution of the program will only continue when all threads are done with the loop. In the first loop, however, this is not needed, as it only stores a numeric value to each position of an array. For this reason, I added the "*nowait*" directive to this loop, which will ignore the synchronization and continue execution, reducing waiting times and improving performance. Later on, I decided to also try to use static scheduling with the directive *"schedule(static, INTERVALS/omp_num_threads)"*, in which *omp_num_threads* corresponds to the number of threads in use and is obtained from the environment variable OMP_NUM_THREADS with the *"getenv()"* function and converted to integer with the *"sscanf()"* function which allows for a dynamic chunk size.
From the results I analyzed, the use of scheduling proved to further improve the performance and, particularly for a higher number of threads, increase the consistency of the results.
There was also an attempt to parallelize the final *for* loop, however this was found to be ineffective. As it was predicted in the previous report, the number of iterations on this loop was so small that the improvements brought by parallelizing it's execution were nullified by the overhead implied by the use of OpenMP, even causing a small increase in execution time. This overhead can be influenced by the access to memory from individual threads, the sequential sections of the program, the time required to handle OpenMP constructs, among others. While the overhead for "$static\ schedule$" directives depends on the chunk size and has a very low variation, the overhead from "$parallel\ for$" directives increases with the number of threads used, meaning the efficiency of the parallelism will decrease. [2]
After measuring both the real time of execution and the user time of execution, I registered 5 measured values of real time of execution for each number of threads used, as well as the time of execution of the sequential program, and used the averages to plot the Speedup and the Efficiency

$$S = \frac{SequentialTimeofExecution}{RealTimeofExecution}$$

$$E = \frac{Speedup}{NumberofThreads}$$

.
The following graphics display the results obtained:

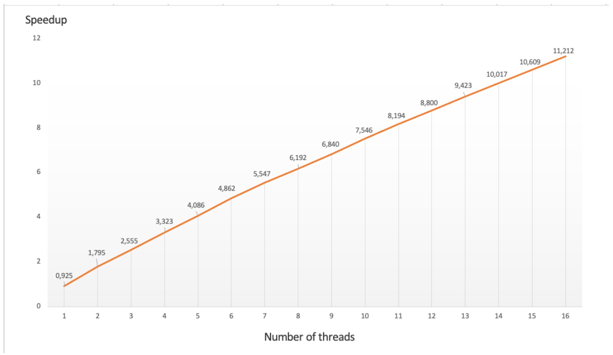

Figure 1: Performance Speedup versus number of threads used

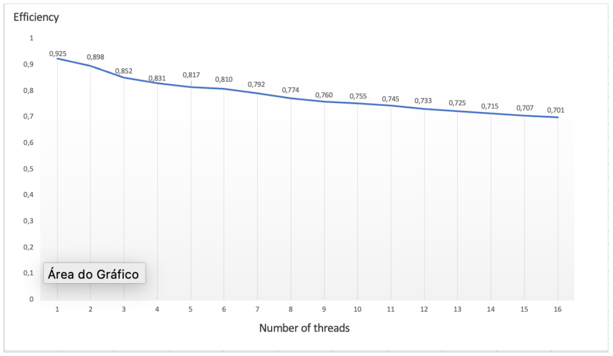

Figure 2: Parallelism Efficiency versus number of threads used

Despite observing a slight decrease in efficiency with the increase in number of threads, the speedup variation was approximately linear with a maximum average of 11,212 obtained when using 16 threads at an efficiency of 0,701.

2 EXERCISE 5: MPI

On this exercise, we explored and experimented with MPI [5] in order to understand the concept of message passing and the use and implementation of parallel message passing applications.

2.1 Submitting MPI jobs

To start the exercise, we were given a job script *run.sh* to look into and think about a few questions. Following an analysis of the script and some reflection, i was able to infer that the script will run the program *hello_mpi* and store the output to the *mpi.log* file. A few arguments are passed to the resource manager, using the *#BSUB* command, such as: "-q hpc", which specifies the queue to the HPC, "-J My_Applicattion" sets the job name to "My_Application". "-n 4" sets the number of cores to 4, "-R "span[hosts=1]"" and "-R "rusage[mem=2GB]"" that specify the same host and memory needed of 2Gb for every core, "-M 3GB" defines the maximum memory usage for each core, "-W 24:00" limits the real time of execution, or wall time, to 24 minutes, "-u your_email_address" saves the recipient's email address, "-B" sends an email notifying the recipient set when the job is started, "-N" sends another email notification the the job is concluded and "-o Output_%J.out" and "-e Error_%J.err" specify the error and output files.
After this analysis of the script, I created and compiled an MPI program in C similar to the one introduced in chapter 9.1 of the book[3], which will print for each thread the current *rank* and the *size* parameters used in the MPI functions. Running this job multiple times and looking at the output files, we can see that, despite what could be expected, and what we are used to seeing in sequential programming, the *rank* values are not printed in an ordered way. This happened because no methods of enforcing an order or synchronization were implemented, leading to an unpredictable order of execution.

2.2 Distributed Calculations

For the second part of this exercise, we went back to the *calc.c* code that was analyzed in the previous report and parallelized using OpenMP in the previous exercise, and this time we are prompted to write a parallel MPI version of it. To start the program, I used the typical MPI initialization functions to initialize MPI and set the communicator's rank and size. To parallelize the loops, we have to set an amount of interations that each process will perform by dividing the total amount by the *size* value. The first iteration of a process will be the product of the current rank and the number of iterations per rank and the last iteration will be the sum of the first iteration and the number of iterations per rank, if that is less than $INTERVALS$ - 1. There is also the need to assign the remaining iteration to a process, in this case the root process, in case the amount of iterations is not perfectly divisible by the amount of processes. I also used the MPI_Bcast function for inter process communication of the results in each operation, and MPI_Gather after each to loop to collect all the values in the root process before proceeding. To conclude the use of MPI, the MPI_Finalize function is called.
After compiling the code with *mpicc*, I scheduled jobs using different numbers of processes by using the $-n$ flag after the *bsub* < *run.sh* command, and modified the script to also use the Linux *time* function , which allowed me to check the real time of execution of each execution in the output files. After registering the values obtained, I plotted the Speedup and Efficiency for each number of cores and got the following results:

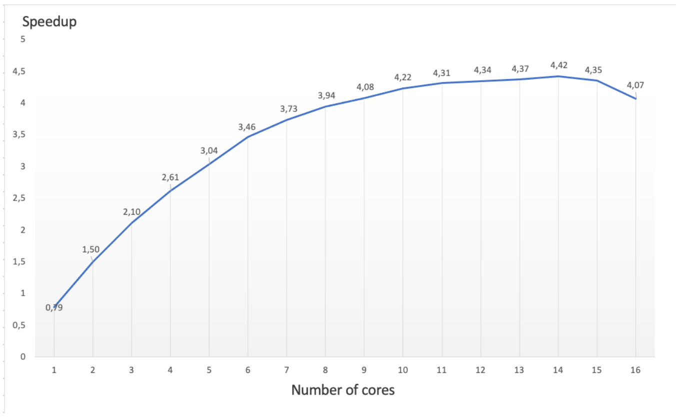

Figure 3: Performance Speedup versus number of cores using MPI

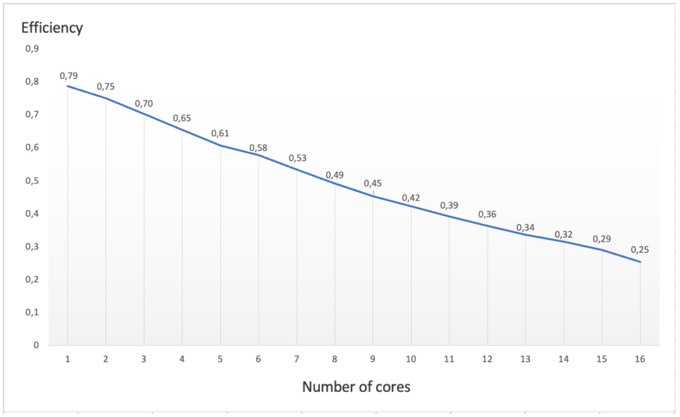

Figure 4: Efficiency versus number of cores using MPI

As we can see from the graphics, the performance obtained from the use of MPI is very different from the one we got using OpenMP. The speedup is not as linear, nor does it reach such high values as it only got up to 4,42 and even started decreasing when using more than 14 cores. The efficiency also declines massively with the used of more cores, reaching as low as 0,25 when using 16 cores. I believe this happens due to the fact that MPI uses distributed memory, and as such, the overheads associated are much higher. This happens because the processes are constantly communicating with one another and waiting for others to finish. With an increase in cores used, there are more communications happening between processes leading to more overhead time and a decrease in efficiency.

3 EXERCISE 6: OPENCL ON GPUS

For the last exercise covered in this report, we worked with OpenCL to write parallel programs in GPUs. We started with a few introductory exercises and concluded by rewriting the *calc.c* file in parallel using OpenCL.

To start off, we analyzed and ran a C program that uses the OpenCL library to examine the available OpenCL accelerators using the following API calls: clGetPlatformIDs will take a number of entries and return the number of OpenCL platforms available, as well as a list of those platforms; clGetPlatformInfo takes a platform ID returned by the previous function mentioned, a platform information specifier and a maximum size in bytes to allocate for the platform information data, and returns a pointer to a location in memory where information about the platform's profile, version, name, extensions or vendor will be returned to, and the actual size in bytes of the respective data; clGetDeviceIDs takes a platform returned by clGetPlatformIDs, a device type specifier and a number of possible entries, and returns the number of devices matching with the desired type, as well as a list of those devices; clGetDeviceInfo will take a device ID returned by clGetDeviceIDs, the device info specifier and a maximum memory size in bytes to allocate for the return information, and returns a pointer to the memory location where the data about the information requested will be stored, as well as the size in bytes of that data. This program combines this API calls and prints out relevant information about the machine being used. The documentation used to understand these API calls was the Khronos Registry for OpenCL 1.0[4].

After compiling and running the program on a CPU, I learned that I was using a AMD Accelerated Parallel Processing platform, from vendor Advanced Micro Devices, Inc, using OpenCL 2,0 AMD-APP version with 1 device: an Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz running version OpenCL C 1.2 with 24 compute units, 32Kb of local memory size, 192930 MB of global memory size, 48232 MB of max allocation size and a max work-group total size of 1024. After changing the Makefile, loading the CUDA module and recompiling, we run the same program on a GPU and get the following information: the platform is now a NVIDIA CUDA from vendor NVIDIA Corporation, running version OpenCL 3.0 CUDA 11.7.101 with one device: a Tesla V100-PCIE-16GB running version OpenCL C 1.2, 80 compute units, a local memory size of 48 KB, a global memory size of 16160 MB, a max alloc size of 4040 MB and max work-group total size of 1024. Notice that in both version, the lower version of OpenCL running in the devices are limiting, despite the higher versions running on the platforms.

The second exercise uses OpenCL API calls to build a program that adds vectors with kernel operations. It starts by declaring a function called KernelSource for the kernel addition operation. Inside the main function, at first the functions described before are called to get information about the platform and devices available. It then uses the functions clCreateContext, clCreateCommandQueue, clCreateProgramWithSource, clBuildProgram and clBuildProgramInfo to initialize an OpenCL context in a device, a command queue, create a program within that context using the KernelSource as source, build that program and get the build's info. After that, it uses the functions clCreateKernel, clCreateBuffer, clEnqueueWriteBuffer, clEnqueueNDRangeBuffer and clSetKernelArg for the kernel operations. After creating the kernel and the buffers for the inputs a and b and the output c, and writing them onto the buffers, the program sets those objects as arguments for the kernel and then enqueue the command to execute on the kernel. Finally, clFinish is used by the program to know when all commands are complete and stop the timer. After this, there is a series of function calls to release the objects and free the variables used. When compiling and running the program, it prints that for C = A + B there were 1024 results, of which 1024 were correct, and the time of execution was 0.011912 seconds. On the following exercise, we have a program identical to the one from the previous exercise, but set to run on a GPU. We are asked to start by changing the code to run on a CPU. We do this by changing the "device_type" parameter on the clGetDeviceIDs function call from $CL\_DEVICE\_TYPE\_GPU$ to $CL\_DEVICE\_TYPE\_CPU$. After this we compile and run the code. The output shows that 1024 results out of 1024 were correct and the kernel ran in 0.011262 seconds, which is as similar time to the one obtained in the previous exercise. We then readjust the code to run on the GPU by changing the clGetDeviceIDs back to $CL\_DEVICE\_TYPE\_GPU$, commenting some lines on the Makefile, loading the CUDA module and recompiling the program. After this, I ran the program on the GPU and once again got 1024 correct results out of 1024, but the time it took for the kernel to run was 0.002387 seconds, meaning a speedup of 4.718 compared to the same program running on the CPU. This result is expected because, the GPU has a much higher amount of cores than the CPU. Even though these processing units are not individually as powerful as the CPU's, the GPU's architecture focus is optimizing matrix arithmetic and floating point operations, and the high number of cores makes it ideal for highly parallel calculations. Given the nature of this program, which is mainly arithmetic, the GPU performs better than the CPU, despite a higher memory access time.

Finally, we were instructed to revisit the *calc.c* code and redo it using OpenCL. The implementation was similar to the other OpenCL programs we analyzed on the previous exercises: start by getting the platform info, securing a device, creating a context, a command queue, buffers and a program... with the main difference being that the two for loops in this program required the use of two kernel objects, associated to two kernel functions - one to fill the $from\ array$, and one to perform the operations. After creating the kernels, the buffers for the arrays were created, the arguments for the first kernel set and the command enqueued. I could only set the arguments for the second kernel after the execution of the first, since the modifications performed to the $from$ array we necessary in the second kernel. After compiling and running the program, I verified that the kernel operations took only 0,021537 seconds and the real time of execution was, on average, around 0,300 seconds, which is the best from all the methods used in this databar.

REFERENCES

[1] OpenMP Architecture Review Board. 2020. OpenMP. https://www.openmp.org/.

[2] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. 2007. *How to Get Good Performance by Using OpenMP.* 125–190.

[3] Gerhard Wellein Georg Hager. 2010. *Introduction to High Performance Computing for Scientists and Engineers.* Taylor Francis Group.

[4] The Khronos Group Inc. 2007. OpenCL 1.0 Reference Pages. https://registry.khronos.org/OpenCL/sdk/1.0/docs/man/xhtml/

[5] The Open MPI Project. 2022. Open MPI: Open Source High Performance Computing. https://www.open-mpi.org/