**Assignment 3(OpenCL) Documentation**

Md Rafiqul Islam - 12123971

Technology used:

C, C++, OpenCL

How to run program:

1. Compile .cpp files:

* g++ -O3 -I/usr/local/cuda-11.4/targets/x86\_64-linux/include/ -L/usr/local/cuda-11.4/targets/x86\_64-linux/lib/ -o opencl\_version.exe opencl\_version.c -lOpenCL (for OpenCL code)

1. Run program:

* ./ opencl\_version.exe

Code Structure:

The project has three one C++ file: opencl\_version.cpp. and one kernel file: kernel.cl

* opencl\_version.cpp – This c plus plus file contains the sequential version and OpenCL version of code. We put both version of code in a same file to measure their performance and compare them. This file has different sections. First part of the file contains the helper functions we need. Then next part of the code contains sequential version of code which inside Sequential\_code() function and last part of the code contains the OpenCL host code. We also put OpenCL code inside another function called OpenCL\_code() to clean our code.
* Kernel.cl–In this file the OpenCL kernel had been written. The main part of parallelism happed here.

Tables:

Table-1 performance of sequential and openCLversion:

|  |  |  |  |
| --- | --- | --- | --- |
| Sequential time (s) | OpenCL time (s) | Speed up | Optimized OpenCL time (s) |
| 17.3434 | 11.324 | 2.32 | 3 |

Speedup Graph and Description:

Fig-1: Mandelbrot speed up curve for different versions of code.

Fig-2: Convolution speed up curve for different versions of code.

Graph Description:

Three different version were tried to parallelize the code. First version was only using ‘omp task’ and second version was ‘omp taskloop’ and another version was ‘omp parallel for’. All version of code compiled and run in the ALMA multiple times and the result were almost same for each time run. The calculated speed up data for different version of Mandelbrot part shown in Fig-1 and convolution part shown in Fig-2. Both figures show the comparison between ‘omp task’, ‘omp taskloop’ and ‘omp parallel for’ version of the code. In these both speed up curve figures, x-axis contains thread numbers and y-axis contains speedup values.

If we see Fig-1, for Mandelbrot part there are three speed up curve. One curve is for ‘omp task’, one for ‘omp taskloop’ and another one for ‘omp parallel for’. From these curves is clearly seen that All versions were performed almost same but for 16 threads the ‘omp parallel for’ version performed slightly better than others and ‘omp taskloop’ a little slow. For parallelizing different threads were used, they are 1, 2, 4, 8 and 16. When the threads number are increasing for parallel for versions of code, the speedup value is also increasing, which means the parallelization is working.

Similarly, From Convolution part in Fig-2 we also can see that when the threads number are increasing the speedup value of all versions also increasing. In this part all the versions of code perform almost same. There was small difference which is not visible in the curve.

Discussions:

* Performance differences between omp task, taskloop and parallel for versions: There is almost no performance difference between these versions. ‘omp Parallel for’ version divide the total loop iteration (depends on how many processor using for parallelization), then these division of iterations execute parallelly. This way code executes faster than sequential code which means it boost performance. Similarly, when we tried ‘omp task’ version it created multiple tasks in single threads and other threads parallelly execute these tasks. During tasking we must take care of synchronization if we want data racing condition free code. Here we used omp atomic for synchronization. Taskloop version also need synchronization. In ‘omp taskloop’ version we set how many tasks will created and how many iterations will run in a task. All versions performance is almost similar but in Mandelbrot part we saw that only for 16 threads the ‘omp parallel for’ version performs slightly better. In all versions for 16 threads, we got almost around 12x performance.
* Task granularity (small vs big tasks): We also tested some other version of small tasking inside the nested loops. From working experience, it has seen that yes task granularity matters. When tasks are small then the parallel execution can’t perform well because these small tasks divided into threads, but these tasks are too small and there are a large number of tasks generated and all these tasks couldn’t finish at the same time. Then one thread has to wait for other. After all small tasing takes more time. But if we divide large work then it gives better result. For taskloop, it has seen that if we try to create a small number of task or a huge number of tasks for both case the performance dropped. If we balance the task depending on using threads number, then it gives best output.
* Distributed the work/task generated:

During parallelization with different numbers task were generated but form them the thread\*40 and grainsize image height\*40 gives better result.

Task generated for taskloop version:

1 thread - 1 \* 40 = 40

2 threads - 2 \* 40 = 80

4 threads - 4 \* 40 = 160

8 threads - 8 \* 40 = 320

16 threads - 16 \* 40 = 640

The formula for calculating the number of tasks is number of tasks = ((number of threads) \* 40.

For parallel for version:

Thread 3

Thread 2

Thread 1

Thread 0

Result

Fig-3: Distribution of work in parallel for

Here we discuss about work distribution for ‘parallel for’ version. In Fig-3 there is an example of work distribution for ‘parallel for loop’ using 4 threads. The work division depends on how many threads used for parallelization. If 2 threads were used, then loop would be divided by two parts. Similarly, if 16 threads were used then loop would be divided into 16 parts. For example, suppose we want to parallelize Mandelbrot part which has multiple for loops. The outer loop has total iteration is 1536, using 4 threads. So first, we just set thread number for parallel execution. This line ‘omp\_set\_num\_threads(4)’ ensure that out code will use 4 threads. Then ‘#pragma omp parallel for’ will divide the whole loop into 4 parts. Each part of the work will take 1536/4 iterations. And then it will start executing parallelly. After execution had finished the pragma will auto gathered the parallelly executed result and will give a final output.

* Differences in speedup: The speedup has been calculated for both part Mandelbrot and convolution. Each part has three different version ‘omp task’, ‘omp taskloop’ and ‘omp parallel for’. For both Mandelbrot and convolution part when all the versions using 1 thread, they almost give same speedup as sequential code and for 2, 4, 8 threads the speed up was increasing. The only exception happened in Mandelbrot part. For 16 threads the ‘omp parallel for’ performs slightly better than ‘omp task’ and ‘omp taskloop’ version. In ‘omp parallel for’ version for 16 threads we got speed up ~12.8237, in ‘omp tasloop’ version speedup was ~12.063 Furthermore, in ‘omp task’ version we got speed up ~12.5071.
* Differences in speedup I observed with different clauses: Yes, I observed different speed up in some cases with different clauses. During parallelize with ‘omp taskloop’ version first when we used only ‘#pragma omp taskloop’, the performance was not too good but after adding num\_task() and grainsize() clause the code gave better performance. Similarly, when I try to parallelize Mandelbrot part of the code using ‘#pragma omp parallel for’ I saw some differences in speedup when using different clauses. At first try I just used ‘#pragma omp parallel for’ here by default the schedule was static in that time I got some speed up but not much, like for 16 threads I got only around 5x speedup. But after adding dynamic scheduling using ‘#pragma omp parallel for schedule(dynamic)’, I got more speed up. It was around 12.8x which is great.
* Interesting findings: There are some interesting things I found during parallelization. In first try when I used ‘#omp task’ inside ‘#pragma omp parallel’ the code was superfast but that time I realize that the output of total pixel count is incorrect. After that I try to find the reason behind it. After many research I found some issues. Here, at first time I didn’t use any synchronization. That’s why when the threads running, they may wish to use shared variables and data race condition happened. To resolve race condition, I used omp atomic inside the nested loop. Similarly, for ‘omp taskloop’ to synchronize we used omp atomic and adding num\_task(), grainsize() clauses give better results.

After that I was trying to find another solution. Then I tried ‘omp parallel for’ version like this ‘#pragma omp parallel for’, this time also the execution speed most likely as exception but the same problem raised. The pixel counts still giving wrong result. Then with ‘omp parallel for’ I used reduction() clause which helps me a lot to getting correct output. Now code is faster but not too fast. After some research I found that schedule() clause. Normally omp used static schedule which can’t use time efficiently because when the parallel work is running some threads completed their works early and some threads still working that time. These ways the time-consuming increase in the code. But fortunately, omp has dynamic schedule. By using dynamic schedule this problem had been resolved. When dynamic schedule was used the threads who finished his works, take extra work from other threads, and execute code faster. By using schedule() clause I got better performance.