BLG562E – Parallel Computing for GPUs using CUDA

Homework #1

Mustafa İzzet Muştu - 504211564

Development and Test Environment

Development and testing is done on a docker container which uses Ubuntu 22.04.2 image and gcc v11.3.0 installed in it. Host machine is Windows 11 22H2. It is observed that the run time on container is as twice as the run time on host machine. Hardware information as follows: AMD Ryzen 7 5800X, 32 GB 3400 MHz, MSI RTX 3060 12 GB.

1. Identifying the Compute Intensive Function of the Program

When we look at the Table 1, we can see that most of time consumed by **mat_mul** function. We can confirm that observing the function code, there are 3 chained for loops and complexity is O(n^3).

Run Time (s)/Fn. Name	Total	main	mat_mul	init_random	init_empty	write_to_file
1	6.12	0.00	6.09	0.02	0.01	0.01
2	6.19	0.00	6.15	0.03	0.01	0.01
3	6.17	0.00	6.12	0.03	0.01	0.01
4	6.15	0.00	6.12	0.01	0.01	0.01
5	6.18	0.00	6.15	0.02	0.01	0.00

Table 1

2. Optimizations and Effects on the Performance

a) What does the –O0 option do? Explain it.
According to the gcc documentation on it's website, **-O0** argument disables the optimization done by compiler. At a result of this, run time increases upto **~6.8 seconds**.

b) How does the performance change? Try to explain how the changes you made might have an effect on the performance.
Changing type of **elem_t** from **double** to **float** reduced the run time downto **~6.2 seconds**. This happened because **double type is 8 byte** while **float type is 4 byte**.

c) How is the performance? Why do you see the speedup/slowdown? Try to explain it. (Hint: Register accesses vs. memory accesses)
Using a local variable makes the program to store the value in a certain register instead of accessing the memory and load it to a register in each iteration reduced the run time downto **~2.9 seconds**. This is because second method is more costly in terms of instruction number and accesing to the memory is also slower than accesing to the register.

d) How is the performance now?
Transposing before accessing the matrix reduced the run time about **0.1 seconds** even if we do not consider the overhead of the **transpose** function. In C, matrices

are stored in row-major order. It means that when we access to 2 certain element in a matrix in the same row, their values are more probably cached rather than in the same column.

e) Now perform, loop unrolling to see if this improves performance. Compile, execute and record the result below. Do not use the transposed matrix.
Nothing has changed because we access the memory same way.

f) How does the performance change? What does –O3 option do in the compilation command line?
Run time reduced to **~6.10 seconds**. **-O3** flag optimize the code at the level of 3. In each level up, compiler tries to reduce execution time more while increasing the compilation time.

Optimization	Execution Time (seconds)
None	6.80
Use of float	6.20
Local variable use for intermediate calculation	2.90
Matrix transpose	2.80
Loop Unrolling	2.80
Use of –O3	6.10

3. Algorithmic Changes to Matrix Multiplication and Its Performance Effect

We use 3 for loops for sum of outer products so function complexity does not change. However, we use less 1 line of code in which we access the memory. In addition, for the default algorithm, we access an element N^3 times iterating column-wise however, in sum of outer product algorithm, we access 2*N^2 + N times. As a result, run time decrased from **~6.80 seconds to ~6.10 seconds**.

4. Scalability Study (Optional, Extra Points)

Everything set to default to analysis the scalability. Different N values are chosen and plotted the Run-Time/N graph. It can be seen in Figrue 1. Doubling the input matrix size makes the run time approximately 8 times longer. This is caused by the 3 chained for loops. We can say that this algorithm can not be scaled up, we need to parallelize the task.

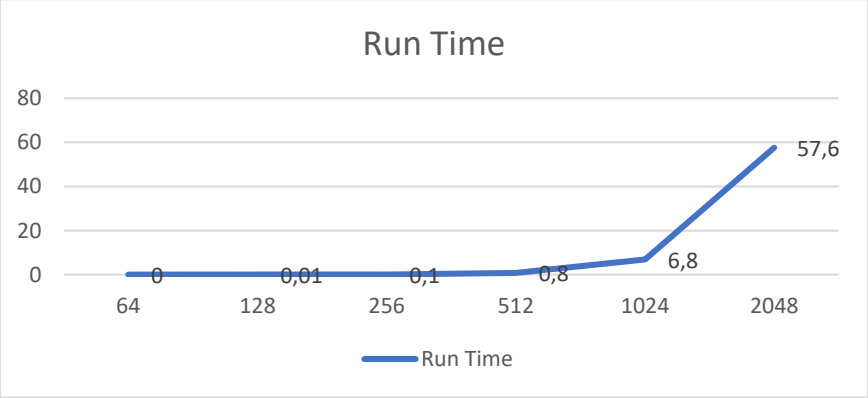

Figure 1