BLG562E – Parallel Computing for GPUs using CUDA Homework #1 Due on: 15/03/2023

For this homework you are given a source file written in C called **mat_mul.c**. The program in this source file performs basic matrix multiplication. Carefully read and try to understand the given source code. You will be performing optimizations on the given implementation and reporting on the performance of the given program.

In your homework submission, start with explaining your methodology, describing the processor you run your experiments, the compiler and its version and the specific compiler options that you use (could be different for each part of the homework).

Submit a single source code and your write-up for answering the questions below. Your source code should include all modifications that you are asked to do and explanations in the code as comments.

1. Identifying the Compute Intensive Function of the Program

In the first step of this homework, you are expected to identify the most compute intensive function of the program. For this purpose, without making any changes to the given source code compile the code with profiling option. You can use the following commands on Linux for compiling using ***gcc*** with ***gprof*** profiling option enabled:

```
➢ gcc –g –pg mat_mul.c –o mat_mul
```

Then run the binary:

```
➢ ./mat_mul
```

After running your program, there will be a generated output file to be used by *gprof* named ***gmon.out***.

Run the command below to get a readable output from ***gprof*** tool.

```
➢ gprof mat_mul gmon.out
```

What is the most compute intensive function of the program? Explain how you arrived to that result.

When you run a program Operating System calls adds noise to the performance results. For the rest of the homework, I suggest you make multiple runs (at least 5 times) and report on the average.

2. *Optimizations and Effects on the Performance*

You are now going to optimize the most compute intensive function (you must have found out that it is the *mat_mul* function) by performing a number of changes to it.

a) Without making any changes to the source code, compile the code with the command line below and run the program, and report the result in the table given below.

```
➢ gcc mat_mul.c –O0 –o mat_mul_original
➢ ./mat_mul_org
```

What does the –O0 option do? Explain it.

b) In the given source code, the matrix elements are defined as ***double*** type. Now, modify the source code to change the matrix elements defined as ***float*** type. Compile the code with the command line below and run the program, and report the result in the table given below.

```
➢ gcc mat_mul.c –O0 –o mat_mul_float
➢ ./mat_mul_org
```

How does the performance change? Try to explain how the changes you made might have an effect on the performance.

c) In function **mat_mul**, the inner loop writes directly to **`r[i][j].`** Modify the inner loop to write to a local variable and make only a single write to **`r[i][j]`** at the end of the inner loop. Compile your modified code using the command below and execute your program and record the result in the table given below.

```
➢ gcc mat_mul.c –O0 –o mat_mul_float_local
➢ ./mat_mul_local
```

How is the performance? Why do you see the speedup/slowdown? Try to explain it. (Hint: Register accesses vs. memory accesses)

d) The inner loop of the matrix multiply code access both matrix a and b. One of these matrices is accessed using row wise and the other column wise. To avoid a column wise access we can transpose the matrix. Write a function transpose, which swaps elements **`[i][j]`** with **`[j][i]`**.

Transpose the column-wise accessed matrix and then update your code so that both matrices are accessed using a row wise pattern. Compile, execute and record the result.

```
➢ gcc mat_mul.c –O0 –o
  mat_mul_float_local_transposed
➢ ./mat_mul_local
```

How is the performance now?

e) Now perform, loop unrolling to see if this improves performance. Compile, execute and record the result below. Do not use the transposed matrix.

```
➢ gcc mat_mul.c –O0 –o
  mat_mul_float_local_unrolled
➢ ./mat_mul_local
```

f) Finally in the optimization part, compile your code with following command line and run your code and report the result below. Do not use the optimization local variable usage, loop unrolling, and transposing.

```
➢ gcc mat_mul.c –O3 –o mat_mul_float_o3
➢ ./mat_mul_local
```

How does the performance change? What does **–O3** option do in the compilation command line?

Optimization	**Execution Time (seconds)**
None	
Use of float	
Local variable use for intermediate calculation	
Matrix transpose	
Loop Unrolling	
Use of –O3	

3. Algorithmic Changes to Matrix Multiplication and Its Performance Effect

The given source code implements the matrix multiplication as dot product multiplication. You can perform matrix multiplication as sum of outer products. Change the matrix multiplication function to implement the matrix multiplication as sum of outer products. Do not transpose the second matrix. Use double for matrix element type. Compile your code using the **gcc** option **–O0**. Run your program and report on the performance. How is the performance? Can you comment on the algorithmic changes and their expected effects on the performance?

4. Scalability Study (Optional, Extra Points)

Study the scalability of matrix multiplication (for both the dot product and sum of outer products implementation) by changing the input matrix size (defined by N in the source code) and reporting the performance.