Programming Project #2

GPU Programming

How to compile the code:

```
For C++ code        $ g++ matrixMul_cpu.cpp -o matrixMul_gpu.exe
For CUDA code       $ nvcc matrixMul_gpu.cu -o matrixMul_gpu.exe
```

How to run the code:

```
For C++ code        $ ./matrixMul_cpu.exe
For CUDA code       $ ./matrixMul_gpu.exe
```

How to get execution time of code:

```
For C++ code        $ time ./matrixMul_cpu.exe
For CUDA code       $ nvprof ./matrixMul_gpu.exe
```

Part 1: Getting Started

Requirements of Part 1:

1. Complete the GPU version of the program and write a Makefile to compile your programs. Finally, please compile and run the codes and report the outputs.

```
CPU Output:    Size of matrix (N) is 512 by 512.
               Starting CPU computation
               It took 1214.386333 ms on avg.
               RUN OK.

GPU Output:    Size of matrix (N) is 512 by 512.
            Starting unoptimized GPU computation
               It took 9693.590333 ms on avg.
               RUN OK.
```

2. How do these two versions of the code compare? Report how long it takes to execute each version of the matrix multiplication.

   We can compare two versions of the code by only Computation Time, as we can't nvprof for CPU code to investigate more.
   CPU Version takes 1214.386333 ms
   GPU Version takes 9693.590333 ms

3. Is the GPU version optimal? Report why or why not?

   GPU Version is not Optimal, because it was taking more execution time that CPU Version. It because the

- initialization of card
- allocation of memory on host to device memory
- copy data from host to device memory
- Kernel execution
- Data copy from device to host
- Deallocation of memory

These all are performing with single thread, that why GPU is not optimized.

Part 2: Multiple Threads

Requirements of Part 2:

1. Please compile and run your modified code and report the outputs.
   Size of matrix (N) is 512 by 512.
   Starting Optimized GPU computation
   It took 7.279333 ms on avg.
   RUN OK.

2. Profile this code. How does this version compare to your GPU code in Part 1? Please report it.

   We can compare the version's using the nvprof command, the insight is Part 2 code was so much better optimized than Part 1, as we can clear notice the execution time.
   The factors better in Part 2 Code than Part 1 Code are:
   - Significant improvement in API Calls execution time
   - Efficient memory allocation in Host to Device

Part 3: Multiple Blocks

Requirements of Part 3:

1. Each cell in the matrix should be assigned to a different thread. Each thread should do O(N * *number of assigned cell*) computation. Assigned cells of different threads does not overlap with each other. And so, no need for synchronization. Please compile and run your modified code and report the outputs.
   Size of matrix (N) is 512 by 512.
   Starting Optimized GPU computation
   It took 5.362667 ms on avg.
   RUN OK.

2. Then, profile this code and report the output. How does this version compare to your GPU code in part 2? Report.

   We can compare the version's using the nvprof command, the insight is Part 3 code was optimized than Part 2, as we can notice the execution time.

The factors better in Part 3 Code than Part 2 Code are:

- Minor improvement in API Calls execution time
- Reducing the throughput by CUDA kernel using gridDim

Part 4: Optimize

Requirements of Part 4:

1. Can you optimize the number of threads and blocks you use? Report your effort.
   Yes, we can still optimize the no. of threads and blocks by using dim3 to specify dimensions of vector. Instead on using direct specific no of threads, we can the dynamic allocation of thread like this to better optimize.
   ```
   int THREADS = 8;
   int BLOCKS = N / THREADS;
   dim3 threads(THREADS, THREADS);
   dim3 blocks(BLOCKS, BLOCKS);
   ```

2. Move the data initialization to the GPU in another CUDA kernel and prefetch the data to GPU memory before running the kernel. Did the page faults decrease? Report why or why not?
   Yes, the pages faults are significantly decreased from Part 3 Code.
   The factors better in Part 4 Code than Part 3 Code are:
   - Data initialization by another CUDA Kernel
   - Prefetch the data into GPU memory
   - Significant decrease in page faults
   - Minor improvement in API Calls execution time
   - Taking Advantage of Unified Memory by efficient memory allocation into Device, which resulted only one transfer of data from device to host instead of vice-versa also.

Summary

Version	Execution Time
CPU	1214.386333
GPU	9693.590333
GPU_v2	7.279333
GPU_v3	5.362667
GPU_v4	5.347

We can observe that GPU versions are optimized gradually by using the threadIdx and blockDim in Part 2(v2) from Part 1(v1). By using the gridDim, blockIdx and threadIdx in Part 3(v3) from Part 2(v2). Again, in Part 4(v4) with the help of prefetch and kernel for data initialization we have improved further. For each requirement, the insights are mentioned as Above.