Submission for High Performance Computing Exercise 3

Authors: Danylo Kravchenko (kravch0000), Lukas Daugaard Schröder (schroe0006)

April 17, 2023

1 Launching 2D kernels and exploiting the massive parallelism

In this section, we need to move matrix multiplication to the device. It involves:

1. allocating memory on the host and device
2. copying the initialized matrices to the device memory
3. calling the kernel function
4. copying the result of the kernel back to the host memory
5. freeing memory on the host and device.

Very important moment is how to measure the time for GPU operations. It's bad practice to use CPU timers for measurements because it will stall the GPU (`https://developer.nvidia.com/blog/how-implement-performance-metrics-cuda-cc/`). For that, we need to use CUDA events.

Last, but not least, is to figure out the grid and block size for the kernel. GPU has a limit of 1024 threads per block. Naive matrix multiplication involves going through the i, j, k loops. The kernel takes a portion of the data assigned to it by the GPU and performs the multiplication on this data only. The kernel uses thread indexing to distribute the computation across the available threads on the GPU. That's why it's important to figure out the correct indices of the assigned portion of the data. The kernel gives us i and j indices of the data to operate on, and the kernel does a k loop similar to the naive implementations of past exercises. We have decided to use a temporary variable to store the sum of the k loop, since this is a minor change and gives a performance boost.

So, for defining the kernel, we need to use `dim3` structure and to specify each dimension. Because of the limit of 1024 threads per block, the product of the `blockSize` part has to be 1024. Hence, $\sqrt{1024} = 32$, `dim3 blockSize(32, 32, 1)`. Whereas, `gridSize` is depending on matrix size `N`. Hence, `dim3 gridSize(N / blockSize.x, N / blockSize.y, 1)`.

Because we have modified the way we call our executable to use the matrix size as an argument, we need to change the job script:

```
#!/bin/bash
#SBATCH --job-name=T1
#SBATCH --output=T1-result
#SBATCH --time=00:05:00
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=4000
#SBATCH --reservation=gpucourse
#SBATCH --account=s-gpu-course
# GPU settings
#SBATCH --partition=pascal
#SBATCH --gres=gpu:1
```

```
ml CUDA/7.5.18

srun T1.o 1024
srun T1.o 2048
srun T1.o 4096
```

Output of the job:

```
MATRIX SIZE: 1024, RUNS: 5
Mean execution time: 7.588102 ms
MATRIX SIZE: 2048, RUNS: 5
Mean execution time: 48.564902 ms
MATRIX SIZE: 4096, RUNS: 5
Mean execution time: 308.136157 ms
```

2 Experiencing parallelism using MPI + OpenMP + CUDA

For this exercise, we choose to use 2 versions of MPI communication to see the difference.

MPI One-sided communication: Use MPI Window and have the rank `0` process distribute the matrix addition to all other ranks, and after that, all ranks have calculated their partitioned addition matrix `C2`, the rank `0` process gathers all the partitioned `C2` matrices into one. We used MPI's built-in timer `MPI_Wtime` to measure the time for the processes and GPU to finish.

We got the following execution time for `4096`×`4096` matrices:
`Time taken in seconds: 0.762095`

MPI collective: initialize matrices `A` and `B` on the node with rank `0` and scatter the data between other nodes. Then perform the addition on each data partition on a node and gather the data on the master node again.

Execution time for `4096`×`4096` matrices:

```
MATRIX SIZE: 4096
Total elapsed time for MPI + GPU: 787.850142 ms
```

Since the One-sided communication approach was faster we our `T2.c` and `T2.cu` files use this approach. To see the code for the collective approach see `T2_collective.c` and `T2_collective.cu`.