18646 How to Write Fast Code II
Mini-Project 2

Hoang Anh Le
Ke Xu
Chen Xie

1. Matrix Multiplication

1.1. Optimization goal

For this project, we try several approaches to optimize the matrix multiplication algorithm in GPU devices. The optimizations were mostly done by optimizing the use of shared memory in GPU.

Target device: NVIDIA GeForce RTX 2080 on GHC machines.

Relevant specifications:

- Maximum number of threads per block: 1024
- Total amount of shared memory per block: 49152 bytes
- Warp size: 32

1.2. Optimization process

Shared memory enables cooperation between threads in a block. When multiple threads in a block use the same data from global memory, shared memory can be used to access the data from global memory only once. Shared memory can also be used to avoid uncoalesced memory accesses by loading and storing data in a coalesced pattern from global memory and then reordering it in shared memory. Using this method, there is no penalty for unaligned accesses by a warp in shared memory.

a) Fix for any n

For the provided code, it did work with a matrix size that is not a power of 2. By providing a boundary check, we can make it work with any matrix size.

```
if(row < sq_dimension && col < sq_dimension)
  for(int k = 0; k < sq_dimension; k++)
    sq_matrix_result[row*sq_dimension + col] +=
      sq_matrix_1[row * sq_dimension + k] * sq_matrix_2[k * sq_dimension + col];
```

The average throughput for this case was only 3.8 GFLOPS on GHC63. We set it as a baseline for other optimizations.

b) Use max number of threads per block

The number of threads per block was increased from 2x2 to the maximum number, which is 32x32. More threads per thread block enables more parallelism and increases the use of shared memory later.

Speed gain (compared to part a): 6.26x

c) Use local sum for each thread

Instead of writing to the result matrix which is a global data structure directly, we accumulated the sum of each thread block to a local sum register before writing to global memory. This method avoids the high delay and contention of accessing the same global data structure.

Speed gain (compared to part b): 1.89x

d) Tiling

This approach used shared memory (in L1 cache) of each thread block to save a small tile of the matrix; therefore, each thread did not have to access the global data every time. Each element in a tile was read from global memory only once with no wasted bandwidth.

The tile size was set to be the same as the thread block width, which was 32x32.

Speed gain (compared to part c): 0.96x

This result was slower than the previous throughput. However it enables further optimizations.

e) Tiling with unrolling

The loop to aggregate the local sum in each thread block was unrolled to increase the execution performance. The unrolling factor was manually searched for, and 16 was found to be the best.

Speed gain (compared to part d): 1.17x

1.3. Optimization results

We tested these approaches on both GHC57 machine and Gradescope autograder in the above order. Note that GHC57 is slower. The throughput was shown in Figure 1 (for GHC) and Figure 2 (for Gradescope autograder).

At the end, we can achieved an average throughput of **233.0 GFLOPS** on Gradescope autograder, which is **13.3 times** the original throughput.

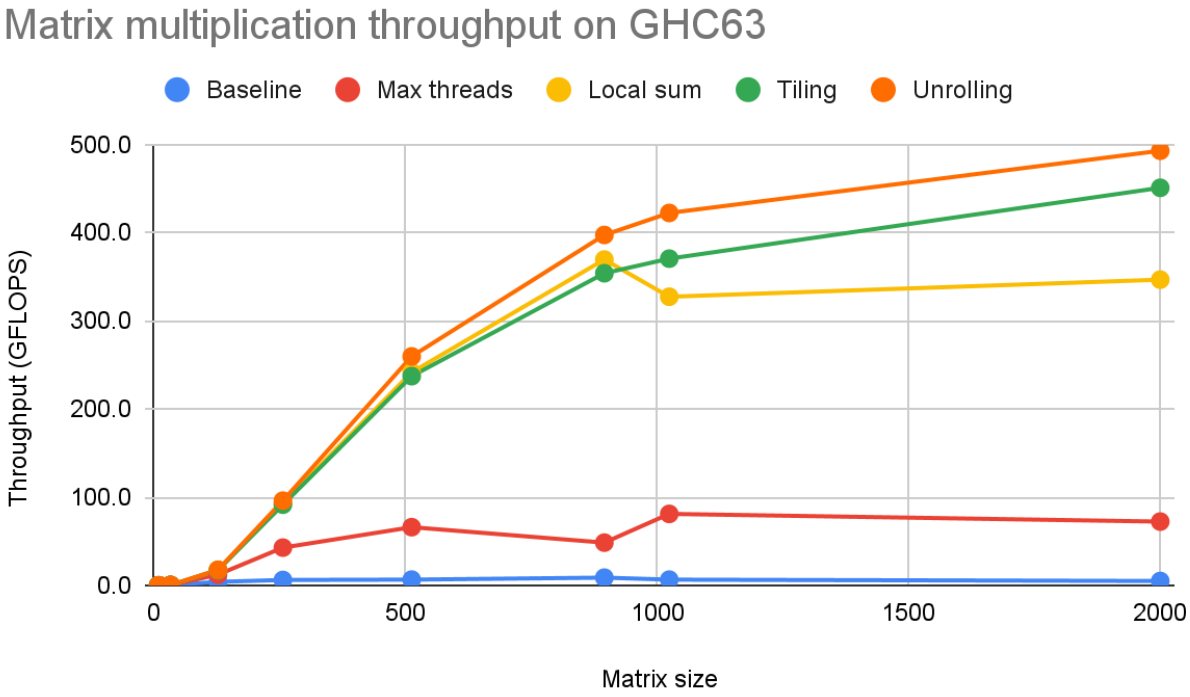

Figure 1

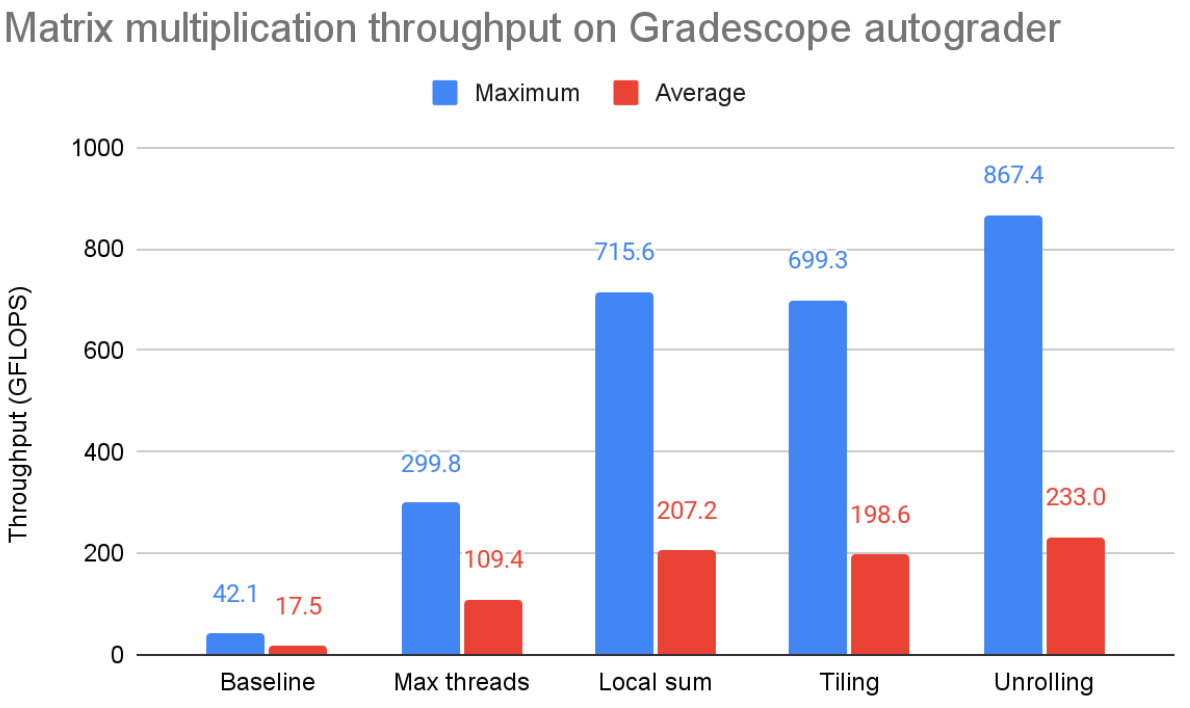

Figure 2

2. K-Means

2.1. Optimization goal

We optimized for the same hardware as above. Most optimizations moved the computation from CPU to GPU and used the shared memory to avoid global data access.

2.2. Optimization process

a) Fix compute_delta

Kmeans03 and 04 require more than 1024 threads per block => Invalid

Fix:

We limit max threads per block to 1024 by creating an array to hold the result for each block and sum later to find the final delta.

We set this as a baseline (2.79s).

b) Update clusters in device

We updated the clusters in device instead of host by creating 2 new kernels: update_clusters_sum and update_clusters_average

Speed gain (compared to part a): 1.64x

c) Use cluster size and cluster buffer

We use a shared memory to hold cluster size and cluster for each thread block, then aggregate them to the global data structure later.

Speed gain (compared to part b): 1.15x

2.3. Optimization results

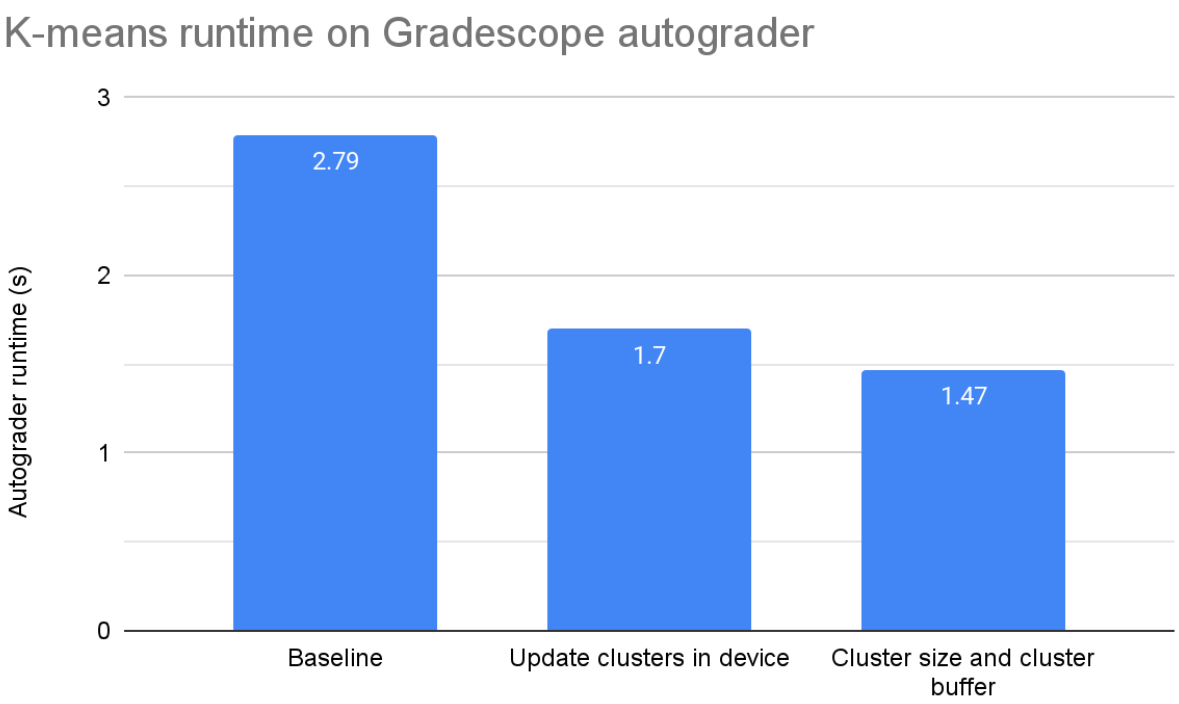

Figure 3

On Gradescope autograder, we get 1.9x compared to the baseline.