CS473: Assignment on CUDA Programming Report

Yijia Diao

1 Introduction

In this assignment, I implement `RBFKernel` and try to optimize this program in CUDA. In order to implement basic `RBFKernel`, I split the calculation process to two part: **Norm2** and **Reduction**, and implement two kernels for them. To optimize `RBFKernel`, I focus on optimizing Norm2 kernel. The experiment result turns out that, the optimization of sequential addressing is efficient.

2 Analysis of Assignment Problem

2.1 Basic `RBFKernel`

`RBFKernel`'s definition is:

$$K(x_i, x_j) = e^{-\frac{||x_i - x_j||^2}{2\sigma^2}}$$

in which x_i, x_j are two n-dimensional vectors, $||x_i - x_j||^2 = \sum_{k=1}^{n}(x_i^{(k)} - x_j^{(k)})^2$. Therefore, the main calculation process that can be parallelized is $||x_i - x_j||^2$. I divide this process into two part:

- **Norm2**: for $1 \leq k \leq n$, calculate $(x_i^{(k)} - x_j^{(k)})^2$.
- **Reduction**: the sum calculation from $k = 1$ to n.

Below is the baseline implementation of the two kernels:

- **Norm2**: for each thread, read $x_i^{(k)}$ and $x_j^{(k)}$ from global memory, do the calculation, then write back to global memory.
- **Reduction**: referencing to NVIDIA example of Reduction[1] provided by TA. I use the code of Reduction#5: Unroll the Last Warp with Sequential Addressing as the Reduction kernel baseline.

2.2 Optimization

Since the **Reduction** process is related in detail in our class, my optimization mainly focus on **Norm2** kernel.

Refering to NVIDIA CUDA C++ Programming Guide Design Guide[2] and Better Performance at Lower Occupancy[3], my optimizing trial of Norm2 kernel contains these stages:

- **Using Shared Memory**
  As I have learned in class, shared memory is much more faster than global memory to match the speed of GPU. Therefore the first optimization step should be using shared memory. In this part, each thread read $x_i^{(k)}$ and $x_j^{(k)}$ from global memory to shared memory, do the mutiplication, then write back to global memory.

- **Concatenating two Vector in Global Memory**
  Since the global memory of x_i and x_j is separated and we cannot control the CUDA malloc space of the two vectors easily, I choose to concatenate the two vectors in doubled continuous space of global memory. The concatenate is quite simple in this stage, just put x_j after x_i. The kth thread read the global memory of `k` and `k+n` and do calculation.
- **Sequential Addressing**
  The simple concatenate has a apparent question of Interleaved Addressing. In order to do sequential addressing, I tried thread-level and Block-level sequential addressing.
  Thread-level Sequential Addressing: In CUDA memory copy stage, copy x_i and x_j alternately into global memory. For example, vector: $[x_i^{(1)}, x_i^{(2)}, x_i^{(3)}]$ and $[x_j^{(1)}, x_j^{(2)}, x_j^{(3)}]$ would be $[x_i^{(1)}, x_j^{(1)}, x_i^{(2)}, x_j^{(2)}, x_i^{(3)}, x_j^{(3)}]$ in global memory. Thus, each thread will read form adjacent position in global memory into shared memory, and one thread only needs to read global memory once.
  Block-level Sequential Addressing: In CUDA memory copy stage, copy a block number of x_i and x_j alternately into global memory. For instance, if `blockDim`=2, vector: $[x_i^{(1)}, x_i^{(2)}, x_i^{(3)}, x_i^{(4)}]$ and $[x_j^{(1)}, x_j^{(2)}, x_j^{(3)}, x_j^{(4)}]$ would be $[x_i^{(1)}, x_i^{(2)}, x_j^{(1)}, x_j^{(2)}, x_i^{(3)}, x_i^{(4)}, x_j^{(3)}, x_j^{(4)}]$ in global memory. Thus, for threads in one block, they can read adjacent space from global memory into shared memory, and one thread only needs to read global memory once.
- **Reducing Idle Threads**
  In sequential addressing, there would be half of threads idle when calculating. In this stage, based on block-level sequential addressing, I reduce the thread number per block in half. Each thread read twice within 2-block address.

Based on the final optimization of Norm2 kernel, I also change the Reduction kernel to Reduction #6: Completely Unrolled Loop, since **loop unrolling** will bring high efficiency.

3 Experiments & Analysis

3.1 Experiment and Time Result

In this part, I will show the baseline and all optimization performance. Table1 shows the corresponding relation among method, code and output program.

Table 1: Method, Source File and Program Comparison Table

Method	Source File	Program
Baseline	RBFKernel.cu	baseline.out
Shared Memory	RBFKernel-opt.cu	opt_sharedmem.out
Concatenate	RBFKernel-opt2.cu	opt_cat.out
Thread-level Sequential Addressing	RBFKernel-opt3.cu	opt_seq_thread.out
Block-level Sequential Addressing	RBFKernel-opt4.cu	opt_seq_block.out
Reducing Idle Threads	RBFKernel-opt5.cu	opt_seq_nidle.out
Reduction #6	RBFKernel-opt6.cu	opt_reduct6.out

For this report reviewer, you can run `make` to compile all code, then run the `.out` to watch the time result. `make clean` to clean the source file folder. If you only want to test Norm2 kernel, please uncomment `#define ONLY_NORM2` on the head of source file, if exist.

The time result of all implementation can be divided into two levels.

Level 1: Baseline

```
dlgpu@ArchLab102:~/yijia/hw1$ ./baseline.out
initialize ready
n=1024:
CPU cal 1024 size cost:0.7736870ms
GPU cal 1024 size cost:3.8298650ms
n=4096:
CPU cal 4096 size cost:3.1029740ms
GPU cal 4096 size cost:3.8165930ms
n=16384:
CPU cal 16384 size cost:12.4301340ms
GPU cal 16384 size cost:3.7894700ms
n=65536:
CPU cal 65536 size cost:49.7420110ms
GPU cal 65536 size cost:3.8891400ms
n=262144:
CPU cal 262144 size cost:199.1048040ms
GPU cal 262144 size cost:5.4446610ms
n=1048576:
CPU cal 1048576 size cost:797.8863370ms
GPU cal 1048576 size cost:8.1920940ms
```

Figure 1: Test output of Baseline

```
dlgpu@ArchLab102:~/yijia/hw1$ ./opt_sharedmem.out
initialize ready
n=1024:
CPU cal 1024 size cost:0.7738070ms
GPU cal 1024 size cost:3.7616600ms
n=4096:
CPU cal 4096 size cost:3.1047330ms
GPU cal 4096 size cost:3.7467700ms
n=16384:
CPU cal 16384 size cost:12.4278680ms
GPU cal 16384 size cost:3.7480910ms
n=65536:
CPU cal 65536 size cost:49.7698550ms
GPU cal 65536 size cost:3.8621910ms
n=262144:
CPU cal 262144 size cost:199.2172000ms
GPU cal 262144 size cost:5.3438640ms
n=1048576:
CPU cal 1048576 size cost:798.5567990ms
GPU cal 1048576 size cost:8.0893910ms
```

Figure 2: Test output of Shared memory

.

```
dlgpu@ArchLab102:~/yijia/hw1$ ./opt_cat.out
initialize ready
n=1024:
CPU cal 1024 size cost:0.7782740ms
GPU cal 1024 size cost:3.7531580ms
n=4096:
CPU cal 4096 size cost:3.1502400ms
GPU cal 4096 size cost:3.7464630ms
n=16384:
CPU cal 16384 size cost:12.8087370ms
GPU cal 16384 size cost:4.1877830ms
n=65536:
CPU cal 65536 size cost:50.8397760ms
GPU cal 65536 size cost:3.8736280ms
n=262144:
CPU cal 262144 size cost:203.3881600ms
GPU cal 262144 size cost:5.3752990ms
n=1048576:
CPU cal 1048576 size cost:814.2304610ms
GPU cal 1048576 size cost:8.1453930ms
```

Figure 3: Test output of Concatenate

Baseline, shared memory and concatenate method has almost the same performance.

.

Level 2: Sequential Addressing

Thread-level, Block-level and No-idle sequential addressing has almost the same performance.

Also, based on no-idle block-level sequential addressing, changing the Reduction kernel to Reduct #6 kernel will also have similar performance.

```
dlgpu@ArchLab102:~/yijia/hw1$ ./opt_seq_thread.out
initialize ready
n=1024:
CPU cal 1024 size cost:0.7726790ms
GPU cal 1024 size cost:3.6514450ms
n=4096:
CPU cal 4096 size cost:3.1286990ms
GPU cal 4096 size cost:3.6495320ms
n=16384:
CPU cal 16384 size cost:12.4354370ms
GPU cal 16384 size cost:3.6977480ms
n=65536:
CPU cal 65536 size cost:49.7897140ms
GPU cal 65536 size cost:3.4570970ms
n=262144:
CPU cal 262144 size cost:199.0905360ms
GPU cal 262144 size cost:4.2555540ms
n=1048576:
CPU cal 1048576 size cost:684.7517120ms
GPU cal 1048576 size cost:7.7105590ms
```

Figure 4: Test output of Thread-level Sequential Addressing

```
dlgpu@ArchLab102:~/yijia/hw1$ ./opt_seq_block.out
initialize ready
n=1024:
CPU cal 1024 size cost:0.7936720ms
GPU cal 1024 size cost:3.7078510ms
n=4096:
CPU cal 4096 size cost:3.1172530ms
GPU cal 4096 size cost:3.8871610ms
n=16384:
CPU cal 16384 size cost:12.5401520ms
GPU cal 16384 size cost:3.7892130ms
n=65536:
CPU cal 65536 size cost:50.4228570ms
GPU cal 65536 size cost:3.7933990ms
n=262144:
CPU cal 262144 size cost:201.2707870ms
GPU cal 262144 size cost:4.9805790ms
n=1048576:
CPU cal 1048576 size cost:797.9574520ms
GPU cal 1048576 size cost:7.5431540ms
```

Figure 5: Test output of Block-level Sequential Addressing

.

```
dlgpu@ArchLab102:~/yijia/hw1$ ./opt_seq_nidle.out
initialize ready
n=1024:
CPU cal 1024 size cost:0.7851060ms
GPU cal 1024 size cost:3.7560360ms
n=4096:
CPU cal 4096 size cost:3.1050300ms
GPU cal 4096 size cost:3.7166190ms
n=16384:
CPU cal 16384 size cost:12.4310070ms
GPU cal 16384 size cost:3.7299940ms
n=65536:
CPU cal 65536 size cost:49.7484810ms
GPU cal 65536 size cost:3.6596210ms
n=262144:
CPU cal 262144 size cost:199.0268180ms
GPU cal 262144 size cost:4.8739720ms
n=1048576:
CPU cal 1048576 size cost:797.8839920ms
GPU cal 1048576 size cost:7.5484850ms
```

Figure 6: Test output of No-idle Sequential Addressing

```
dlgpu@ArchLab102:~/yijia/hw1$ ./opt_reduct6.out
initialize ready
n=1024:
CPU cal 1024 size cost:0.7763590ms
GPU cal 1024 size cost:3.7411010ms
n=4096:
CPU cal 4096 size cost:3.1058050ms
GPU cal 4096 size cost:3.7332700ms
n=16384:
CPU cal 16384 size cost:12.4266480ms
GPU cal 16384 size cost:3.6937460ms
n=65536:
CPU cal 65536 size cost:49.7635520ms
GPU cal 65536 size cost:3.6887370ms
n=262144:
CPU cal 262144 size cost:199.0916760ms
GPU cal 262144 size cost:4.9838960ms
n=1048576:
CPU cal 1048576 size cost:798.5713700ms
GPU cal 1048576 size cost:7.5056000ms
```

Figure 7: Test output of Reduct #6 kernel

.

The time test of only Norm2 kernel:

```
dlgpu@ArchLab102:~/yijia/hw1$ ./baseline.out
initialize ready
n=1024:
CPU cal 1024 size cost:0.7514350ms
GPU cal 1024 size cost:0.4392940ms
n=4096:
CPU cal 4096 size cost:2.8961150ms
GPU cal 4096 size cost:0.4314110ms
n=16384:
CPU cal 16384 size cost:12.4322440ms
GPU cal 16384 size cost:0.4869640ms
n=65536:
CPU cal 65536 size cost:47.8912410ms
GPU cal 65536 size cost:0.4595480ms
n=262144:
CPU cal 262144 size cost:183.6377750ms
GPU cal 262144 size cost:0.4361260ms
n=1048576:
CPU cal 1048576 size cost:728.0001360ms
GPU cal 1048576 size cost:0.4391960ms
```

Figure 8: Test output of No-idle Sequential Addressing

```
dlgpu@ArchLab102:~/yijia/hw1$ ./opt_seq_nidle.out
initialize ready
n=1024:
CPU cal 1024 size cost:0.6924260ms
GPU cal 1024 size cost:0.4272280ms
n=4096:
CPU cal 4096 size cost:2.7704170ms
GPU cal 4096 size cost:0.4253760ms
n=16384:
CPU cal 16384 size cost:11.3315300ms
GPU cal 16384 size cost:0.4287480ms
n=65536:
CPU cal 65536 size cost:45.2420690ms
GPU cal 65536 size cost:0.4353230ms
n=262144:
CPU cal 262144 size cost:180.9832660ms
GPU cal 262144 size cost:0.4158180ms
n=1048576:
CPU cal 1048576 size cost:727.7964460ms
GPU cal 1048576 size cost:0.4196000ms
```

Figure 9: Test output of Reduct #6 kernel

.

3.2 Analysis

- Baseline, shared memory and concatenate methods Performance Similarity:
  This result that shared memory and concatenate has no distinct difference with baseline is beyond my expectation. When analysing the possible reason, I tried to consider the **operation proportion** of global memory access and GPU calculation. If we consider $\times$ and $+$ are both one GPU calculation, in each Norm2 kernel, global memory access : GPU calculation = 3:2. But in Reduction kernel, this propotion is 3:O(log `blockDim.x`). Although both of them are memory-bounded kernel, the computation is more dense in Reduction kernel, thus the use of shared memory can hide the global memory latency. But Norm2 kernel has less computation operation than memory access, thus shared memory is "useless" in this case.
- Overhead in Sequential Addressing Methods:
  Although in the time test, sequential addressing methods can reach better performance, it has much more overhead. When running the `opt_seq_thread.out` program, the larger the vector, the longer the program runs. The latency mainly happens when executing CUDA memory copy from host to device. Since we copy the vector element one by one, the total latency of the program is much more higher.
  Therefore, to reduce the total latency of the program, I decide to CUDA memory copy alternately at block level. Though in this program, the total latency of Block-level is similar to baseline, this method would bring huge overhead with larger input vector.

- Norm2 Kernel and Reduction Kernel

  As shown in Figure.8 and Figure.9, the time test comparison between final optimization of Norm2 kernel and baseline is almost similar. This indicates that, the time test for Norm2 kernel may be unsuitable, or the optimization is actually not obvious in Norm2 kernel.

  As shown in Figure.7 and Figure.6, the time test comparison between Reduction #5 kernel and Reduction #6 kernel is similar, which indicates that my optimization method of sequential addressing of Norm2 kernel is more effective.

4 Conclusion

In the process of implementing, optimizing and testing `RBFKernel`, I can reach the conclusion:

- The Sequential Addressing optimization is effective.
- For those memory access intensive program, the simple use of shared memory is not useful.

References

[1] M. Harris *et al.*, "Optimizing parallel reduction in cuda," *Nvidia developer technology*, vol. 2, no. 4, pp. 1–39, 2007.

[2] C. NVIDIA, "Cuda c++ programming guide: Design guide," PG-02829-001_v11.2, NVIDIA, Santa Clara, Calif, USA, Tech. Rep., 2021.

[3] V. Volkov, "Better performance at lower occupancy," in *Proceedings of the GPU technology conference, GTC*, San Jose, CA, vol. 10, 2010, p. 16.