Lab 3

Vector Addition on GPU: Memory and Profiler

By Ardalan Ahanchi / Andrew Nelson / Rahil Mehta
CSS 535A - Winter 2020

0. Table of Contents

1. Introduction

Building off of Lab 2, the goal for this lab is get familiar with a profiler. Profilers are tools used that run a kernel (or multiple kernels) multiple times to gather data regarding the kernel's execution. The data gathered from these executions can contain details such as number of elapsed cycles, SM frequency, and cache hit rates. Profilers are very useful since the details they provide can help discover bottlenecks or other weak points in the software. Additionally, profilers can provide helpful suggestions based on the data they gather.

2. Baseline Configuration

The baseline configuration was used in parts 1 and 3 of the lab to test the significant edge-cases.

For this lab, we decided to the configurations as our baseline:

Table 2.1. *Baseline configuration utilized for the performance results.*

Config	N	Block Size	Thread Size
0	2097152	1	1
1	2097152	8	16
2	2097152	32768	1
3	2097152	32768	1024
4	16777216	32768	1024

These configurations were chosen based on the results found in the tests performed in Lab 2. The configurations here were either points in Lab 2 where we started to see significant differences in performance when comparing the CUDA implementation and the sequential implementation. From these points, the "edge" configuration of blocks and threads were also considered. By "edge", we're referring to the minimum number of blocks and threads for execution and the maximum number of blocks and threads for execution that was performed in lab 2 (32768 blocks, 1024 threads). Each configuration has been assigned a unique configuration ID to help match the configuration to specific results shown later on in the report.

3. Global Memory (Part 1)

3.1. Modifications

Our implementation of lab 2 allowed for dynamic configurations in the run-time (passed as command line arguments). Thus, our implementation would automatically calculate the number of elements that each thread has to process. Due to this feature, we did not need to do any modifications to the lab2 code to get this part of the lab working. However, our extra configurations (explained in section 3.3) were carefully chosen to examine the usage of global memory.

3.2. Baseline Configurations

From the baseline configurations mentioned in section 2, we gathered our baseline profiler results when running it with the normal implementation (which uses the global memory):

Table 3.1.1. *Results for running the global memory implementation with baseline configuration.*

Config	GFLOPS	L1 Cache Hit Rate	L2 Cache Hit Rate	# Registers / Thread
0	0.00926	90.62%	23.04%	19
1	0.65809	90.62%	87.97%	19
2	6.92220	90.62%	89.66%	19
3	1.70754	0%	33.24%	19
4	13.63857	0%	32.26%	19

Interestingly, in our configurations that utilize all 1024 threads, we notice that the L1 cache hit rate has dropped to 0%. Our suspicion for this behavior is that since there are many threads reading and writing data, each thread is constantly overriding data in the L1 cache, causing continuous misses in the cache the next time it’s trying to be read.

3.3. Extra Configurations

From Lab 2, all of our test values of N were multiples of 32. For this reason, we’ve decided to have all values of N for this part of the experiment be non-multiples of 32. For the block and thread size, we decided to try different numbers that better fit the size of N and values that would better suit a configuration designed for multiples of 32.

Table 3.2.1. *Additional configurations to test with the global memory implementation.*

Config	N	Block Size	Thread Size
p1-0	1000000	1000	1000
p1-1	2000000	1024	1024
p1-2	3000000	1024	1000
p1-3	4000000	32	1000
p1-4	5000000	2048	1024

After running the same kernell performed previously under the new configurations, we obtained the following profiler results:

Table 3.2.2. *Results for running the global memory implementation with the extra configs.*

Config	GFLOPS	L1 Cache Hit Rate	L2 Cache Hit Rate	# Registers / Thread
p1-0	16.49185	13.67%	36.37%	19
p1-1	20.42400	26.31%	61.29%	19
p1-2	11.27446	62.91%	62.90%	19
p1-3	0.90100	0%	34.66%	19
p1-4	12.80468	56.79%	64.33%	19

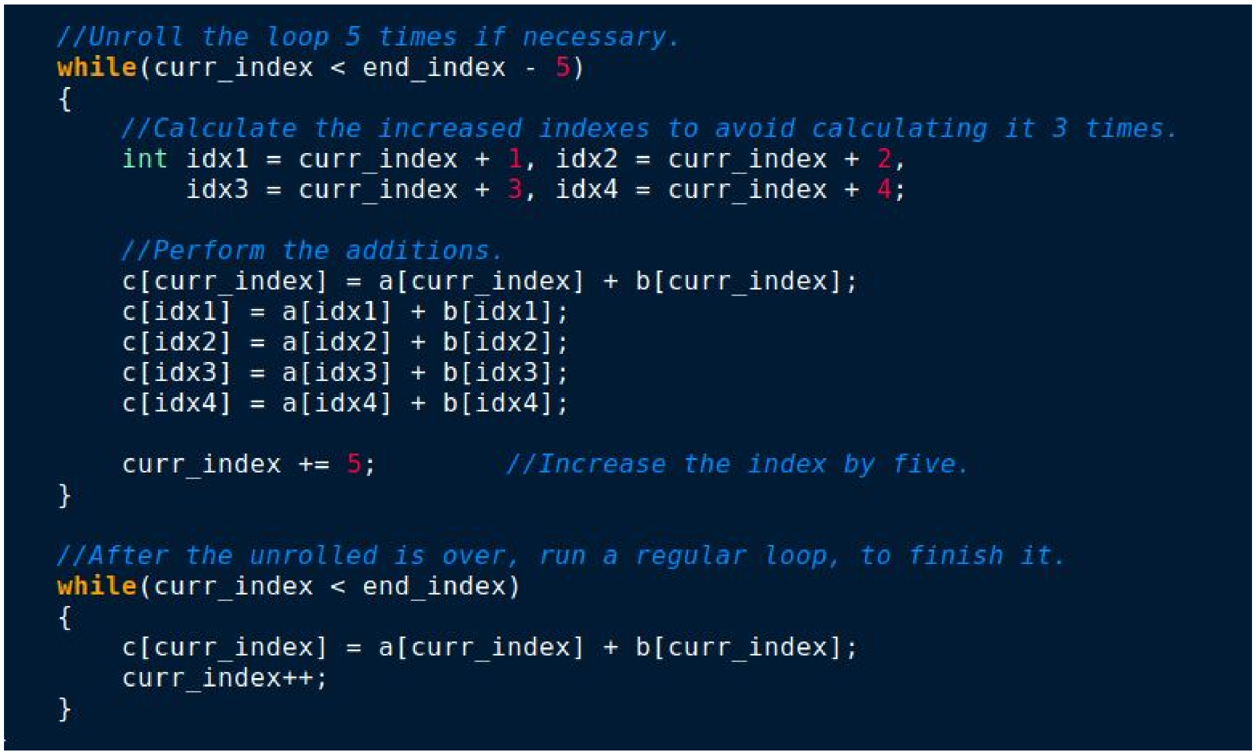
The results that this experiment yielded values that differ quite a bit from the other results in the report. Overall, it appears that as blocks and threads were added, the cache hit rates and GFLOPS improved. What our team had gathered from this experiment is that it appears that a configuration that matches closer to the value N has a better overall performance. This would be due to not wasting additional threads or overworking any specific thread more than it needs to.

The unique result from this data set is the profiler results for p1-3. For this result, we noticed a significant decrease in GFLOPS and cache hit rates. As the cache hit rates decrease, it's expected to see the GFLOP value decrease as well. This is most likely due to each thread processing a large amount of elements without utilizing the cache (by iterating each additional index in a loop).

4. Registers (Part 3)

4.1. Modifications

To take advantage of registers, we attempted to unroll the loops in each thread. Since the number of elements that each thread has to process is calculated in runtime, the only way to unroll the loops was to conditionally unroll them when each thread had more than 5 (chosen to unroll specifically when the use of registers is beneficial). In other words, loops are unrolled in 5 element bursts of calculations. The remaining of the loop is not rolled and proceeds normally. Figure 4.1.1 demonstrates the algorithm used for loop unrolling within the kernel.



```
//Unroll the loop 5 times if necessary.
while(curr_index < end_index - 5)
{
    //Calculate the increased indexes to avoid calculating it 3 times.
    int idx1 = curr_index + 1, idx2 = curr_index + 2,
        idx3 = curr_index + 3, idx4 = curr_index + 4;

    //Perform the additions.
    c[curr_index] = a[curr_index] + b[curr_index];
    c[idx1] = a[idx1] + b[idx1];
    c[idx2] = a[idx2] + b[idx2];
    c[idx3] = a[idx3] + b[idx3];
    c[idx4] = a[idx4] + b[idx4];

    curr_index += 5;        //Increase the index by five.
}

//After the unrolled is over, run a regular loop, to finish it.
while(curr_index < end_index)
{
    c[curr_index] = a[curr_index] + b[curr_index];
    curr_index++;
}
```

Figure 4.1.1. *The loop unrolled algorithm which unrolles in bursts of 5.*

However, this modification does not make any sense and it should not be attempted in hope of performance improvements. The reason is that the loop is dynamic and the start/end index of the loop will be determined on the runtime. This would mean that the indexes should be calculated again in the runtime (and possibly take extra registers). Moreover, in our implementation, we calculated the offset indexes only once (they can be calculated up to 3 times each if they are used as indexes in each call).

4.2. Results

Our team performed loop unrolling to observe the changes in the number of registers used during the configurations. Each thread performed an additional 5 culations to help populate the result vector. The following profiler results were obtained as a result of this modification:

Table 4.2.1. *Results for running loop-unrolled version with the baseline configurations.*

Config	GFLOPS	L1 Cache Hit Rate	L2 Cache Hit Rate	# Registers / Thread
0	0.01113	90.62%	24.16%	24
1	1.22284	90.62%	90.01%	24
2	4.46204	90.62%	90.02%	24
3	1.81567	0%	33.25%	24
4	13.45726	0%	33.34%	24

For this modification, the data trends here shows a similar data trend in comparison to our baseline. However, there is a noticeable improvement on the L2 cache hit rate. The suspicion here is that now that each thread is writing and reading 5 indices at time, the thread can grab all of the needed values earlier prior to another thread overriding that value in the cache. We also noticed that the number of registers has gone up by a value of 5. This is expected due to our kernel defining 5 extra variables to help declare the extra indices to modify.

5. Verification

To verify the results of our vector addition, we compare it’s output with the results from the host execution (sequential, and on the CPU). If any element is different, an error message will be displayed. In all the tests that we ran, the results from the GPU execution was exactly the same as the CPU, thus we know that the code is working as expected.

Additionally, we used the *cuda-memcheck* tool to check for memory issues, and out of bounds memory accesses (we faced a few during the development, but they were resolved).

6. Specifications

6.1. Hardware

All of the tests and profiling results gathered in this report were performed on the following hardware:

- Processor: Intel Core i7-9750h
    - Frequency: 2.6 GHz to 4.5 GHz.
    - Cores: 6 Physical Cores (12 logical cores using hyper-threading).
    - Cache: 32K L1-data, 32K L1-instruction, 256K L2, 12288K L3.
    - TDP: 45 Watts.
- Graphics: Nvidia RTX-2070 MaxQ (8 GB GDDR5 VRAM, 85 Watts, CC 7.5).
- RAM: 32GB DDR4 2666 MHz.

6.2. Software

The tests for gathering the FLOPS were performed outside of the profiler environment. This is due to the extra time the profiler adds to the execution time (for tracing). The following are the software specifications for the system that was used for these results:

- OS: Linux Mint 19.3
- Kernel: Linux 5.3.0
- CUDA: 10.2
- Profiler: Nsight Compute
- Build Manager: CMake
- Version Control: Git + GitHub
- Other Tools: Cuda-memcheck, Python3