CS3210 - Assignment 2

Doan Quoc Thinh (e0960479), Nguyen Cao Duy (e0969050)

2024-10-21

Performance Optimization

Optimization 1: Organizing sample and signature data in a contiguous block

`kernel_without_opt1.cu` allocated memory and transferred data from host to device for each sample and signature separately. The first drawback is that this resulted in multiple `cudaMalloc` and `cudaMemcpy` calls as the number of samples and signatures increased. This will create more overhead as the GPU had to allocate many small chunks of memory and transfer data multiple times. The second drawback is that the data was not contiguous in memory on device. Based on our parallelization strategy, each thread accesses a pair of sample and signature data. If the data is not contiguous, each thread in a warp will access different memory locations, leading to scattered and non-coalesced memory accesses which can reduce performance.

To optimize this, `kernel_skeleton.cu` first concatenated all sample sequences, sample qualities, and signature sequences into 3 contiguous strings on the host before allocating and transferring them to the device. In doing so, we need to add an extra step of calculating additional offset arrays and pass them to the kernel to access the correct data for each thread. However, the gain in performance from coalesced memory accesses outweighs the overhead of calculating the offsets.

Optimization 2: Swapping sample and signature access pattern

`kernel_without_opt2.cu` coordinate the threads to share the same signature (`signature_idx` remains the same) but access different samples (`sample_idx` varies):

```
int signature_idx = idx / num_samples;
int sample_idx = idx % num_samples;
```

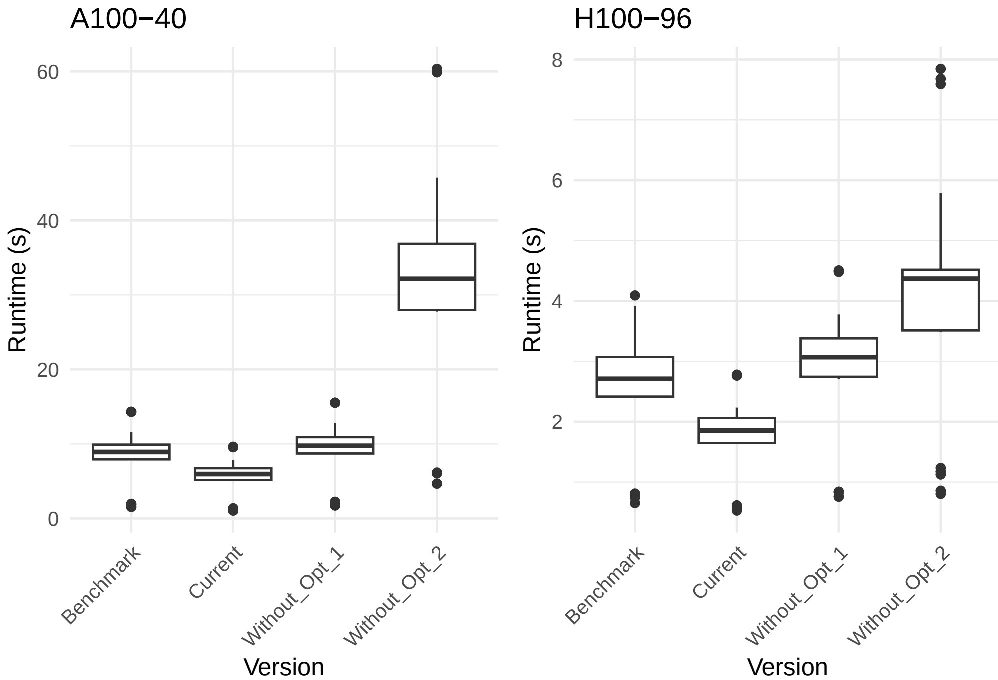
and `kernel_skeleton.cu` optimized it by coordinating the threads to share the same sample (`sample_idx` remains the same) but access different signatures (`signature_idx` varies):

```
int sample_idx = idx / num_signatures;
int signature_idx = idx % num_signatures;
```

This optimization works due to the length constraint of the samples and signatures being different (samples are at least **10 times** longer than signatures). Compared to the unoptimized code, our optimized code has a better utilization of cache as threads in a warp access the same memory location more often (sharing the same sample sequence, which is much longer than a signature) while access the different memory location (different signature sequences, which are much shorter than different sample sequences) less often.

Performance Comparison

Based on the graph below, we can see that the optimized versions of the kernel outperformed the unoptimized versions and the benchmark on both `a100-40` and `h100-96` GPUs. Optimization 2 has a significant improvement to our implementation. Details of the tests used can be found in the appendix.


A100−40
H100−96
Runtime (s)
Version
Benchmark
Current
Without_Opt_1
Without_Opt_2
0
20
40
60
2
4
6
8

Appendix

Result Reproduction

Input Input tests can be generated using the `gpu_gen.sh` script where we create 30 random tests with different parameters. Part of the script is shown below for reference. Take note of the composition of the tests.

```
# Random no wildcards tests
for i in {1..10}
do
    ./gen_sig 1000 3000 10000 0.0 > tests/sig_${i}.fasta
    ./gen_sample tests/sig_${i}.fasta 2000 20 1 2 100000 200000 10 30 0.0 > tests/samp_${i}.fastq
done

# Random wildcards tests
for i in {11..20}
do
    ./gen_sig 1000 3000 10000 0.1 > tests/sig_${i}.fasta
    ./gen_sample tests/sig_${i}.fasta 2000 20 1 2 100000 200000 10 30 0.1 > tests/samp_${i}.fastq
done

# Extreme min tests (2 tests with no wildcards + 3 tests with wildcards)
for i in {21..22}
do
    ./gen_sig 500 3000 3000 0.0 > tests/sig_${i}.fasta
    ./gen_sample tests/sig_${i}.fasta 980 20 1 2 100000 100000 10 30 0.0 > tests/samp_${i}.fastq
done
for i in {23..25}
do
    ./gen_sig 500 3000 3000 0.1 > tests/sig_${i}.fasta
    ./gen_sample tests/sig_${i}.fasta 980 20 1 2 100000 100000 10 30 0.1 > tests/samp_${i}.fastq
done

# Extreme max tests (2 tests with no wildcards + 3 tests with wildcards)
for i in {26..27}
do
    ./gen_sig 1000 10000 10000 0.0 > tests/sig_${i}.fasta
    ./gen_sample tests/sig_${i}.fasta 2180 20 1 2 200000 200000 10 30 0.0 > tests/samp_${i}.fastq
done
for i in {28..30}
do
    ./gen_sig 1000 10000 10000 0.1 > tests/sig_${i}.fasta
    ./gen_sample tests/sig_${i}.fasta 2180 20 1 2 200000 200000 10 30 0.1 > tests/samp_${i}.fastq
done
```

GPU Nodes Used To test on `a100-40 MIG GPU`, we use node `xgph10`. To test on `h100-96 GPU`, we use node `xgpi0`.

Execution Time Measurement The scripts `gpu_benchmark_{a/h}.sh` and `gpu_job_{a/h}.sh` are used with `sbatch` to run the benchmark and our implementation against all the tests generated above. Since `gpu_job_{a/h}.sh` only compiles the `kernel_skeleton.cu` file, we need to replace it with the version of the kernel that we want to test each time, such as `kernel_without_opt1.cu` and `kernel_without_opt2.cu`.

The overall time measurement of the `runMatcher` function is extracted from the standard error stream of the job output at the line containing `(FOR AUTOMATED CHECKING) Total runMatcher time:`. We extract these measurements through a helper script `extract_time.py` and store them in the CSV files `a100-40.csv` and `h100-96.csv` in the folder `report_data/optimization` for easier analysis. These CSV files are shown below for reference.

Table 1: A100-40

Test	Benchmark	Without_Opt_1	Without_Opt_2	Current
1	8.12419	8.78940	28.21890	5.37454
2	8.02356	8.80919	28.24970	5.43868
3	7.99186	8.72244	28.15090	5.17184
4	7.97231	8.72339	28.09760	5.15565
5	7.99204	8.71358	27.94780	5.24586
6	7.89186	8.69044	28.02430	5.13341
7	7.97244	8.72862	28.02690	5.22238
8	8.14516	8.75246	28.18790	5.45013
9	7.91519	8.67745	27.75710	5.13480
10	7.90614	8.90208	27.90350	5.20592
11	9.72864	10.60150	36.06490	6.46554
12	9.87024	10.92740	36.91700	6.63033
13	10.22290	10.98270	37.37980	6.67971
14	9.80637	10.85040	36.67150	6.53392
15	9.91722	10.82100	36.69510	6.69977
16	9.79837	10.84000	36.33190	6.74939
17	9.84011	10.71780	36.34440	6.85628
18	9.94069	10.94940	37.13980	6.78598
19	9.70356	10.65350	36.12550	6.66534
20	9.69087	10.71110	36.11040	6.69323
21	1.54535	1.73894	4.67685	1.05582
22	1.54935	1.74249	4.66958	1.05961
23	1.89895	2.12622	6.10961	1.31807
24	1.88166	2.21549	6.04776	1.36776
25	1.96136	2.23495	6.16287	1.29304
26	11.62330	12.83040	45.74150	7.80184
27	11.60460	12.51950	45.68200	7.72002
28	14.32010	15.51890	60.01580	9.61998
29	14.30310	15.52890	59.87360	9.59452
30	14.31080	15.53090	60.33960	9.55592

Table 2: H100-96

Test	Benchmark	Without_Opt_1	Without_Opt_2	Current
1	2.451690	2.831070	3.513420	1.711380
2	2.432200	2.758380	3.564490	1.669210
3	2.417450	2.742590	3.558960	1.643960
4	2.415420	2.759370	3.511540	1.646400
5	2.469700	2.763990	3.481880	1.648540
6	2.476970	2.747340	3.496970	1.654110
7	2.478450	2.766390	3.530660	1.637370
8	2.406450	2.767340	3.538870	1.661460
9	2.495510	2.704820	4.602420	1.655570
10	2.414020	2.743080	3.719540	1.650400
11	2.957500	3.470050	4.411640	1.992790

Test	Benchmark	Without_Opt_1	Without_Opt_2	Current
12	2.948530	3.313690	4.527560	2.095750
13	2.941710	3.357090	4.719600	2.066280
14	3.012940	3.333540	4.432860	2.020610
15	2.925190	3.529150	4.474760	2.033900
16	3.189580	3.368850	4.391290	2.052770
17	3.090240	3.348200	4.440950	2.025030
18	3.168950	3.382810	4.487060	2.062920
19	2.949840	3.373500	4.433560	2.032380
20	2.988560	3.307620	4.346320	2.027890
21	0.652620	0.761274	0.857004	0.544741
22	0.793596	0.756487	0.803334	0.528035
23	0.746783	0.839857	1.233710	0.614446
24	0.803458	0.833136	1.172730	0.597858
25	0.809958	0.839330	1.125140	0.601568
26	3.255250	3.710790	5.785540	2.233670
27	3.281490	3.777120	5.732060	2.233990
28	3.908380	4.481060	7.592860	2.763190
29	3.916150	4.493470	7.680140	2.779510
30	4.091800	4.506890	7.844970	2.767670