Homework 2

CS 406

Parallel Programming

Spring 2020-2021

Instructor:

Kamer Kaya

Faculty of Engineering and Natural Sciences

Sabanci University

Efe Şencan

25083

Table of Contents

1) Problem Statement

The goal of this project is to implement parallel Sinkhorn-Knopp scaling algorithm which is used to convert a given matrix to a doubly stochastic form, where every row and column sum is equal to one. We used Cuda for the parallel GPU implementation of this problem.

2) Sequential Implementation

The sequential implementation of the algorithm is as follows.

Sequential Sinkhorn-Knopp Algorithm:

Input: A: n x n matrix

Output: rv, cv : row / column scaling arrays (entries of the diagonal scaling matrices)

```
for i ← 1 to n do
    rv[i] ← 1
    cv[i] ← 1
for x ←1 to iterations do
    for i ← 1 to n do
        start ← xadj[i]
        end ← xadj[i + 1]
        for j ← 1 to (end - start) do
            rsum += cv[adj[start + j]]
        rv[i] ← 1/rsum
    for j ← 1 to n do
        start ← txadj[j]
        end ← txadj[j + 1]
        for i ← 1 to (end - start) do
            csum += rv[tadj[start + i]]
        cv[j] ← 1/csum
    global_error ←-1
    for i ← 1 to n do
        start ← xadj[i]
        end ← xadj[i + 1]
        for j ← 1 to (end - start) do
            cur_value += rv[i] * cv[adj[start + j]]
        cur_value = abs(1 – cur_value)
        if cur_value > global_error do
            global_error = cur_value
```

2.1) Sequential Code Optimization

We can improve the performance of the sequential algorithm by re-scheduling the place of the first inner for loop which is used to update the elements of ‘rv’ array. As we can observe from the algorithm, both the first and the last inner loop computes cv[adj[start + j]], and this value is used to compute the ‘rsum’ for the first inner loop while it is being used to compute the error for the last inner loop. Therefore, we can update the ‘rv’ array just inside the last inner loop. Thus, we could get rid of the burden of first inner loop, which runs ‘n’ (number of rows/columns of the matrix) many times at every iteration. However, we still need to update the ’rv’ array only for the first iteration, because the updated version of rv array is required for the computation of cv array and that array ‘cv’ is also required for the computation of error. For this reason, before starting the iterations, we updated the ‘rv’ array using the same inner for loop. The code of the optimized algorithm is as follows.

Optimized Parallel Sinkhorn-Knopp Algorithm:

```
for i ← 1 to n do
    rv[i] ← 1
    cv[i] ← 1
for i ← 1 to n do
    start ← xadj[i]
    end ← xadj[i + 1]
    for j ← 1 to (end - start) do
        rsum += cv[adj[start + j]]
    rv[i] ← 1/rsum
for x ←1 to iterations do
    for j ← 1 to n do
        start ← txadj[j]
        end ← txadj[j + 1]
        for i ← 1 to (end - start) do
            csum += rv[tadj[start + i]]
        cv[j] ← 1/csum
    global_error ←-1
    for i ← 1 to n do
        start ← xadj[i]
        end ← xadj[i + 1]
        for j ← 1 to (end - start) do
            cur_value += rv[i] * cv[adj[start + j]]
            rsum ← cv[adj[start + j]]
```

```
cur_value = abs(1 – cur_value)
**if** cur_value > global_error **do**
        global_error = cur_value
rv[i] ← 1/rsum
```

3) Parallel Cuda Implementation

3.1) Thread Based Parallelization

We utilized thread-based parallelism for implementing parallel Sinkhorn-Knopp scaling algorithm. First, let’s break the problem into three parts namely updating ‘rv’ array followed by updating ‘cv’ array and finally computing the error value based on these updated array values. For the first two parts, we have a nested for loop structure as showed in the sequential implementation section. To achieve GPU based parallelism, we created 1D thread blocks where every thread block contains 512 threads. The number of the thread blocks is determined by ceil(*nov / NO_THREADS + 1). To implement nested loop structure in Cuda, we first access the global thread id of each thread by calculating (blockIdx.x * blockDim.x) + threadIdx. Hence, each thread will perform the job of the outer loop structure that iterates through ‘*nov’ many times. By doing so we get rid of the outer for loop structure. Afterwards, we implement the inner for loop of the code without making further modifications to the for loop. On the other hand, the implementation of the error calculation part is more complicated compared to the first two parts. To compute maximum error, at every iteration, we have to store the local error values and then perform a reduction operation to obtain the global maximum error. To store the local error values, we utilized shared memory which will be discussed in the next section. Similar to the first two parts, we get rid of the outer loop structure that iterates ‘*nov’ many times by calculating the global thread id of each thread. The inner loop that computes the local error values remain same in our implementation. The run-time and speed-up values of the parallel Cuda implementation for Sinkhorn-Knopp algorithm and it’s optimized version is as follows.

Parallel Sinkhorn-Knopp Algorithm

Total iterations = 20, Measured time in second (s)s

Matrix	GPU -O0	GPU -O3
Nlpkkt_240	4.74	4.73
stokes	5.10	5.10
vas_stokes_1M	0.52	0.52
vas_stokes_4M	1.84	1.84

Optimized Parallel Sinkhorn-Knopp Algorithm

Total iterations = 20, Measured time in second (s)

Matrix	GPU -O0	GPU -O3	Speed-Up
Nlpkkt_240	3.28	3.28	1.44
stokes	3.53	3.52	1.44
vas_stokes_1M	0.36	0.36	1.44
vas_stokes_4M	1.27	1.27	1.44

3.2) Shared Memory

As we discussed in section 3.1, we used shared memory to store the local error values which will further be used to compute the global error. We created shared memory by ' __shared double partial_max[NO_THREADS]' which stores error values of type double and contains number of threads many elements which is 512. The reason why they store 512 elements is that every thread block has their own shared memory in the allocated space and every thread block contains 512 threads which is specified in our program. The advantage of using shared memory is that they allow faster memory access than accessing the global memory, but they have a limited memory capacity. Therefore, we are not able to store the whole matrices inside the shared memory. In our implementation, after the shared memory allocation, we access every thread's global id, and every thread inside a particular thread block is responsible for storing the local error value to its corresponding index of the shared memory. After computing and storing the local error values inside the shared memory, we performed reduction operation that computes the maximum errors for every thread block and store these maximum values inside a temporary array. We then compute the maximum error value of that temporary array inside the host memory and print that error value in our program.

3.2.1) Optimizations on Reduction

For the reduction step, our first attempt is to schedule threads which have even thread id, which are responsible for looking and the comparing the local error values in their adjacent cells. The disadvantage of this approach is that we are not utilizing all the threads, instead we are only using threads that have even thread id. Moreover, while checking the thread id, we are using modulo operation which is costly. To overcome this problem, instead of computing (threadIdx.x % (2*s) == 0) where s denotes the blockDim.x, we calculated the shared memory indices by index = 2 * s * threadIdx.x and then check whether we are within the range of blockDim.x. Our next optimization is avoiding shared memory bank conflicts. Therefore, we put a stride that is initialized as blockDim.x/2, and gradually decremented by a factor of 2. By doing so, every thread in a thread block, access partial_max[threadIdx.x] and partial_max[threadIdx.x + s]. Finally, we observed that we achieve better performance in terms of runtime when we utilize 256 threads at every thread block. After performing all of these optimizations and modifications in our program, our runtime and speedup values compared to the optimized algorithm is as follows.

Total iterations = 20, Measured time in second (s)

Matrix	GPU -O0	GPU -O3	Speed-Up
Nlpkkt_240	3.11	3.10	1.05
stokes	3.42	3.41	1.03
vas_stokes_1M	0.34	0.34	1.05
vas_stokes_4M	1.23	1.23	1.03

3.2) GFLOP

FLOP is the measure of how many floating-point operations are done per second by the program, and GFLOPs is a billion FLOPs. In order to measure GFLOPs, after running the program, we executed '**perf stat ./a.out**' command in Gandalf, so that we can observe the total number of instruction counts. By dividing the total instruction counts to the execution time of our program, we can obtain the FLOPs, and finally by dividing FLOPs to 10^9 we can measure the GFLOPs of our program. The **GFLOPs** result of the optimized version of our program with 20 iterations is as follows.

Matrix	GPU -O0	GPU -O3
Nlpkkt_240	0.001	0.001
stokes	0.001	0.001
vas_stokes_1M	0.012	0.012
vas_stokes_4M	0.003	0.003

4) Conclusion

We implemented the parallelized version of Sinkhorn-Knopp using Cuda. Based on our experiments, we obtained a decent performance improvement compared to the sequential algorithm and also the OpenMp version of the algorithm. With for loop optimization that we discussed in section 2.1 and other optimizations such as coalesced memory access in global memory, avoiding shared memory bank conflict, and thread-based parallelism, we achieved the best performance in terms of run time. Our implementation is further open to the improvements. In the future, the nested loop structure can be completely assigned to different threads and the second part of the reduction in the maximum error computation part can be done in GPU.