Assignment 5

Ayush Kumar (170195)

9 November 2020

I pledge on my honor that I have not given or received any unauthorized assistance.

Solutions

Please note that all benchmarking for this assignment (except problem 2 and 5) was performed on the GPU2 server of the CSE department at IIT Kanpur (ip: gpu2.cse.iitk.ac.in). The specifications of the Linux system as well as the NVIDIA GPU on which the calculations were performed are listed below. For all evaluations that we performed in this assignment, we took the median runtime of 5 consecutive runs of the program.

Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
CPU(s):	12
On-line CPU(s) list:	0-11
Thread(s) per core:	2
Core(s) per socket:	6
Socket(s):	1
NUMA node(s):	1
Vendor ID:	GenuineIntel
CPU family:	6
Model:	79
Model name:	Intel(R) Core(TM) i7-6850K CPU @ 3.60GHz
Stepping:	1
CPU MHz:	1199.427
CPU max MHz:	4000.0000
CPU min MHz:	1200.0000
BogoMIPS:	7196.28
Virtualization:	VT-x
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	15360K
NUMA node0 CPU(s):	0-11

Table 1: System Specifications of CSE GPU2 server

1. To run the code, use the following command:

```
nvcc -g -G -arch=sm_61 -std=c++11 assignment5-p1.cu -o assignment5-p1
```

   The CPU runtime for $N = 8192$ for the **serial implementation** of the given code is **27618.1 milliseconds**. Before parallelizing, we must note that the given code segment has multiple dependencies. Listing all the dependencies, we have the following dependence matrix for loops kij:

$$\begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

Clearly from the given matrix we observe that only the inner j loop can be parallelized.

(a) For the naive CUDA kernel, we parallelize the implementation such that each thread j computes the j^{th} column of the matrix F. We use a 1D grid with 1D blocks. The results obtained for $N = 8192$ with different number of threads in each block is summarized below:

Blocksize	Gridsize	Runtime in msec (milliseconds)
(1024)	(8)	2140.93
(512)	(16)	1644.45
(256)	(32)	1597.48
(128)	(64)	1586.84
(64)	(128)	1572.54
(32)	(256)	1569.42
(16)	(512)	1755.14

Table 2: CUDA runtimes obtained using kernel1 for N = 8192

From this, we observe that the best **runtime of 1569.42 msec** is obtained on a block size of (32) and gridsize of (256). This is a **speedup of 17.5976** over the original CPU run.

(b) Running the same kernel (kernel1) for a size of $N = 8200$ gives the following runtimes.

Blocksize	Gridsize	Runtime (in milliseconds)
(1024)	(8)	2147.12
(512)	(16)	1584.87
(256)	(32)	1589.03
(128)	(64)	1519.93
(64)	(128)	1507.52
(32)	(256)	1508.15
(16)	(512)	1755.14

Table 3: CUDA runtimes obtained using kernel1 for N = 8200

We get a **runtime of 1508.15 msec, speedup of 18.3125** for blocksize (32) and gridsize (256). We notice that the results for $N = 8200$ on the same kernel are slightly better than for $N = 8192$. The reason is that for $N = 8192$, there is more skewed divergence in the last warp of the last block due to the if condition (only the last thread doesn't proceed inside the if condition), while this is not the case with $N = 8200$ because of less divergence (about 16 of the 32 threads proceed inside the if statement), which explains why the latter performs slightly better.

To further optimize kernel1, in kernel2 we unroll loop i by 8 times and attain a slightly reduced **runtime of 1462.24 msec** with a 1D blocksize of (32) and 1D gridsize of (256). This amounts to a **speedup of 18.8875**, which is a decent improvement over the previous speedup.

2. To run the code, use the following command:

```
nvcc -g -G -arch=sm_61 -std=c++11 assignment5-p2.cu -o assignment5-p2
```

The **CPU runtime for the serial implementation of the prefix-scan is 55.1930 msec** for $N = 2^{24}$. We used the following algorithm for parallelizing the prefix-scan.

Algorithm:

- First, we define two kernels $kernel\_excl\_prefix\_sum\_init$ and $kernel\_excl\_prefix\_sum$. The description of both of these is given as below.

- The first kernel $kernel\_excl\_prefix\_sum\_init$ is called at the beginning of the exclusive-prefix-scan. It parallely computes independent prefix scans of 4096-sized consecutive sub-arrays and stores them in d_out.

- The second kernel $kernel\_excl\_prefix\_sum$ is iteratively called again and again for joining two consecutive independent prefix scans from the previous iteration. So if in the previous iteration, independent prefix scans of x-sized subarrays was computed, then in the current iteration, the independent prefix scans of 2 consecutive x-sized subarrays would be joined into the independent prefix scan of the 2x-sized combined subarray.

- Loop Invariant: After the end of i^{th} iteration of the while loop outside the second kernel call, $4096 * 2^i$-sized consecutive sub-arrays would contain their independent prefix-scans. Hence, at the end of the 12th iteration, the array would contain the full prefix-scan.

- The algorithm to join two consecutive prefix-scans is simple. If $d\_out[x : x+y]$ and $d\_out[x+y : x+2y]$ are two consecutive independent prefix scans, then to combine them, we simply add $d\_out[x+y] + d\_in[x+y]$ to each element of $d\_out[x + y : x + 2y]$.

To optimize this algorithm for $N = 2^{24}$, we varied the parameters $BLOCKSIZE$, CPT (columns_per_thread) and FAC (Number of data elements each thread updates) defined in the code and found that $BLOCKSIZE = 128$, $CPT = 4096$ and $FAC = 8$ gave the best performance. We also performed complete unrolling of the for-loop in the second kernel. Finally, we obtained a **CUDA runtime of 46.9418 msec, a speedup of (1.1758)**. Note that the speedup is very minor because parallelizing a prefix-scan is in itself a tough task and several research papers have been published about it.

3. To run the code, use the following command:

```
nvcc -g -G -arch=sm_61 -std=c++11 assignment5-p3.cu -o assignment5-p3
```

The serial code on CPU gives **runtimes of 3303.33 msec and 235181.5 msec** on $N = 1024$ and 4096. Now, we parallelize the computation on CUDA such that each thread (i, j) in a 2D block on the 2D grid, computes the value at position (i, j) in the matrix B. Hence we parallelize the i and j loops, while keeping the k loop serial inside the kernel. Using different blocksizes, we get the following table:

Blocksize	Gridsize	Runtime in milliseconds (msec)
(32, 32)	(128, 128)	5003.35
(16, 16)	(256, 256)	4937.02
(8, 8)	(512, 512)	4975.52

Table 4: CUDA runtimes for N = 4096 without blocking

Blocksize	Gridsize	Runtime in milliseconds (msec)
(32, 32)	(32, 32)	126.656
(16, 16)	(64, 64)	92.6011
(8, 8)	(128, 128)	94.7585

Table 5: CUDA runtimes for N = 1024 without blocking

To optimize our parallel implementation, we implement blocking by first loading all the data elements required by all the threads in a block, then calling $\_\_syncthreads()$ to make sure all data has been loaded in a shared memory. Then all threads of the block use the shared memory for faster access of data. We get the following runtimes for different block sizes,

Blocksize	Gridsize	Runtime in milliseconds (msec)
(32, 32)	(128, 128)	3560.95
(16, 16)	(256, 256)	3826.7

Table 6: CUDA runtimes for N = 4096 with blocking

Blocksize	Gridsize	Runtime in milliseconds (msec)
(32, 32)	(32, 32)	75.8739
(16, 16)	(64, 64)	62.1649

Table 7: CUDA runtimes for N = 1024 with blocking

For a final optimization, we unroll the k loop inside the kernel with a step size of 16 to get reduced runtimes. For $N = 1024$ and blocksize, gridsize of (16, 16), (64, 64) we get a **runtime of 55.3035 msec, a speedup of 59.7309** while for $N = 4096$ and blocksize, gridsize of (32, 32), (128, 128) we get a **runtime of 2663.76 msec, a speedup of of 88.2893**. This is a significant speedup and the main reason behind it is the usage of shared memory which provides fast access to frequently used data elements from the input matrix.

4. To run the code, use the following command:

```
nvcc -g -G -arch=sm_61 -std=c++11 assignment5-p4.cu -o assignment5-p4
```

The CPU runtimes for the serial code on $N = 1024$, 2048 and 4096 are **5181.89 msec, 42110.21 msec and 317144.49 msec** respectively.

Using a naive parallel CUDA implementation, where each thread (i, j) in a block operates on the (i, j) element of the matrix (serially executing the k loop), we get the following runtimes.

N	Blocksize	Gridsize	Runtime in milliseconds (msec)
1024	(32, 32)	(32, 32)	723.406
2048	(32, 32)	(64, 64)	5396.69
4096	(32, 32)	(128, 128)	43710.2

Table 8: CUDA runtimes for naive implementation for different N

Now, we use blocking with different block sizes. Again, we first fetch into shared memory all the data required by all the threads in a block, then call *__syncthreads*() before multiplying the row and column vectors and summing up. The results are summarized in the table below.

N	Blocksize	Gridsize	Runtime in milliseconds (msec)
1024	(32, 32)	(32, 32)	239.739
	(16, 16)	(64, 64)	281.62
	(8, 8)	(128, 128)	390.04
2048	(32, 32)	(64, 64)	1887.48
	(16, 16)	(128, 128)	2210.6
	(8, 8)	(256, 256)	3047.27
4096	(32, 32)	(128, 128)	15334
	(16, 16)	(256, 256)	17582.2
	(8, 8)	(512, 512)	24744.2

Table 9: CUDA runtimes for different N with blocking with different block sizes

To further optimize the parallel implementation with blocking, we also perform loop unrolling of the k loop inside kernel 2 with a step size of 16. Finally, we get the following runtimes.

N	Blocksize	Gridsize	Runtime in milliseconds (msec)	Speedup
1024	(32, 32)	(32, 32)	235.385	22.0145
2048	(32, 32)	(64, 64)	1864.38	22.5867
4096	(32, 32)	(128, 128)	14863.9	21.3366

Table 10: CUDA runtimes for different N after unrolling the k loop 16 times

Again, similar to the previous problem we have a **significant speedup of around 22x** due to blocking and the usage of fast shared memory.

5. To run the code, use the following command:

```
nvcc -g -G -arch=sm_61 -std=c++11 assignment5-p5.cu -o assignment5-p5
```

Running the serial code for stencil calculation on different array sizes gives the **CPU runtimes of 4.25696 msec, 21.6949 msec, 173.661 msec and 1425.86 msec** for $N = 64$, 128, 256 and 512 respectively.

(a) Naively parallelizing this using CUDA such that each thread (i, j, k) of a 3D block calculates index (i, j, k) of the stencil, we get the following runtimes:

N	Blocksize	Gridsize	Runtime in milliseconds (msec)
64	(16, 8, 8)	(4, 8, 8)	1.7919
	(32, 8, 4)	(2, 8, 16)	1.89238
128	(16, 8, 8)	(8, 16, 16)	12.7164
	(32, 8, 4)	(4, 16, 32)	11.719
256	(16, 8, 8)	(16, 32, 32)	102.806
	(32, 8, 4)	(8, 32, 64)	98.6665
512	(16, 8, 8)	(32, 64, 64)	841.841
	(32, 8, 4)	(16, 64, 128)	832.862

Table 11: CUDA runtimes for stencil operation for different N and block sizes

The **speedups obtained for** $N = 64$**,** 128**,** 256 **and** 512 **are 2.2495, 1.8512, 1.7601 and 1.7120** respectively with a blocksize of (32, 8, 4).

(b) We now optimize kernel1 by implementing tiling. For each block of thread that computes a stencil of size (a, b, c), we fetch the input submatrix of size (a+2, b+2, c+2) centred around the output stencil and store it in shared memory for the block. We then call *__syncthreads*(), after which each thread (i, j, k) computes the stencil for location (i, j, k) in the matrix as well as its neighbors (for handling boundary cases). We finally get the following runtimes:

N	Blocksize	Gridsize	Runtime in milliseconds (msec)	Speedup
64	(32, 8, 4)	(2, 8, 16)	1.5411	2.7623
128	(32, 8, 4)	(4, 16, 32)	10.1219	2.1434
256	(32, 8, 4)	(8, 32, 64)	95.9127	1.8106
512	(32, 8, 4)	(16, 64, 128)	820.6133	1.7376

Table 12: CUDA runtimes for optimized stencil operation using kernel 2

Note that the improvement in runtime obtained after tiling for the stencil operation is very minor. This is because size of the submatrix (a+2, b+2, c+2) fetched into the shared memory is not aligned with the block size of (a, b, c).