Assignment 4

Aniket Sanghi (170110)

13th November 2020

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

System and Architecture Information (Used CSE server)

```
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:               1200.311
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
```

Compilers Information

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2018 NVIDIA Corporation
Cuda compilation tools, release 10.0, V10.0.130
```

System SSH

```
ssh <username>@gpu2.cse.iitk.ac.in
```

All problems have been executed on this server

GPU Configuration (Used CSE server)

```
Device name: GeForce GTX 1080 Ti
        Integrated or discrete GPU? discrete
        Clock rate: 1544 MHz
        Compute capability: 6.1
        Number of asynchronous engines: 2

        Number of SMs: 28
        Total number of CUDA cores: 3584
        Max threads per SM: 2048
        Max threads per block: 1024
        Warp size: 32
        Max grid size (i.e., max number of blocks): [2147483647,65535,65535]
        Max block dimension: [1024,1024,64]

        Total global memory: 11177 MB
        Shared memory per SM: 96 KB
        32-bit registers per SM: 65536
        Shared mem per block: 48 KB
        Registers per block: 65536
        Total const mem: 64 KB
        L2 cache size: 2816 KB
```

All problems have been executed with this GPU

1 Problem 1

1.1 Compilation and execution instruction

```
export PATH=/usr/local/cuda-10.0/bin:${PATH}
nvcc -g -G -arch=sm_61 -std=c++11 assignment5-p1.cu -o assignment5-p1
./assignment5-p1
```

1.2 Performance Report and Explanation

Exact time details and different codes tried have been commented in code for all the below speedups

kernel1 (Naive Kernel, 8192) gave a speedup of about **12X**
kernel2 (Naive Kernel, 8200) without any optimisation gave a speedup similar to kernel1
kernel2 (Optimised Kernel, 8200) with below optimisations gave a final speedup of about **25X**

Optimisations

- **ThreadsPerBlock**:
  - In the naive version threadsPerBlock used was **1024** but after varying the threadsPerBlock for kernel2, Speedup increased from **12X with 1024** to **17X with 64**
  - The Increase in speedup can be attributed to more number of warps possible with decrease blockDims.
  - Also, the speedup actually worsened for threadsPerBlock = 32 (**15.7X**), reason can be attributed to the threaddivergence that will happen due to N not being a power of 2. We know thread divergence effect is high with lower dims.
- **Manual Unrolling and reusing valuess using registers**:
  - 4-way unrolling with value re-use using local variables (registers) increased the speedup to about **25X**. Value re-use helped as register access is much faster than global memory access and is also unaffected with cache conflicts. Unrolling helped due to less number of operations.
  - 2-way unrolling gave about **23X** and value re-use for higher dimension, 16 gave 10X which was considerably less. Performance lost due to higher number of for-loops (Please see code).

2 Problem 2

2.1 Compilation and execution instruction

```
export PATH=/usr/local/cuda-10.0/bin:${PATH}
nvcc -g -G -arch=sm_61 -std=c++11 assignment5-p2.cu -o assignment5-p2
./assignment5-p2
```

2.2 Algorithms tried and their impact

Exact time details and different codes tried have been commented in code for all the below speedups

Final code gave a speed of about **1.567X on N = (1 << 24)**

1. **Triangular Calculation**:
   - Launched (NxN) sized grid of threads. And for each i, did an atomicAdd of j-th element's value for all threads such that $i > j (y > x)$.
   - Was much slower for given N and smaller Ns as well, because of very large number of threads launched and the one of the bottleneck being atomicAdd synchronisation.
2. **Binary-Tree summation of Chunked Data**
   - Divide data into chunks of size **EPS_BLOCK_SIZE** and for each chunk compute the exclusive sum for its own elements only (Run this kernel once)
   - Then sequentially run another kernel **logX times** ($X = N/EPS\_BLOCK\_SIZE$) which takes current consecutive chunks for which individual exclusive prefix sum have been computer and compute the combined exclusive sum of both by summing the last value of chunk 1 to all values of chunk 2.
   - Launched N/2 threads for each kernel launch, and each thread added value to one element of chunk.
   - For each thread, first stored the value to be summed in **shared memory** and then used it for all threads of the block to add it to the corresponding elements.
   - It didn't give much speed up, it gave varying speedup from **0.9X to 1.1X**
3. **Optimised Binary-Tree summation of Chunked Data**
   - In the above implementation, we can see that for the kernel that do the summation, I launched 1 thread for each element whose value has to be updated. So the work per thread is quite less, to give sufficient work per thread code was updated. Now each thread work on **8 elements of the array**, i.e. the add the required value sequentially to 8 consecutive elements of array.
   - This after fine tuning other variables, gave a speedup of **1.567X** on $N = (1 << 24)$
   - The variables have been fine tuned for giving speedup for above N. For other N the speedups varies and for higher values of N, the variables have to be fine tuned differently for above implementation to give high speedups.

3 Problem 3

3.1 Compilation and execution instruction

```
export PATH=/usr/local/cuda-10.0/bin:${PATH}
nvcc -g -G -arch=sm_61 -std=c++11 assignment5-p3.cu -o assignment5-p3
./assignment5-p3
```

3.2 Performance Report and Explanation

Exact time details and different codes tried have been commented in code for all the below speedups

Best kernel gave a speedup of about **74X** with **Matrix Size = 4096**
Best kernel gave a speedup of about **80X** with **Matrix Size = 2048**

Optimisations [Below speedups are given for Matrix Size = 2048]

- **Register Optimisation**
  - In the naive kernel, instead of adding multiplied values directly to the result matrix, first create a local variable (register) and then update it, and then store it at loop end to result matrix. This increased speedup by **10 (43X to 53X)**. Reason - registers are much faster, also help fixing a register for each thread to be used specifically for this purpose.
- **Unrolling in the above implementation**
  - 8-way unrolling in the above implementation increased speedup from **53X to 55X**. Reason - less number of loop-comparison/update operations
- **Tiling + Unrolling + Register Optimisation**
  - Used Tiling to help exploit spacial locality helped increase the speedup further from **55X to 80X**
  - Also above optimisations of 8-way unrolling and register used. 4-way unrolling gave a bit less speed-up for 4096 but better speedup for 2048.

4 Problem 4

4.1 Compilation and execution instruction

```
export PATH=/usr/local/cuda-10.0/bin:${PATH}
nvcc -g -G -arch=sm_61 -std=c++11 assignment5-p4.cu -o assignment5-p4
./assignment5-p4
```

4.2 Performance Report and Explanation

Exact time details and different codes tried have been commented in code for all the below speedups

Speedups for Naive version for various MATRIX_SIZE

- **MATRIX_SIZE = 1024** Speedup of about **17.3X with Naive Kernel**
- **MATRIX_SIZE = 2048** Speedup of about **25.4X with Naive Kernel**
- **MATRIX_SIZE = 4096** Speedup of about **6.7X with Naive Kernel**

Speedups for optimised version for various **BLOCK_SIZE, MATRIX_SIZE and UNROLLING**

- **BLOCK_SIZE = 8**
  - **MATRIX_SIZE = 1024** Speedup of about **55.3X with Optimised Kernel**
  - **MATRIX_SIZE = 2048** Speedup of about **181X with Optimised Kernel**
  - **MATRIX_SIZE = 4096** Speedup of about **172X with Optimised Kernel**
- **BLOCK_SIZE = 16**
  - **MATRIX_SIZE = 1024** Speedup of about **56.7X with Optimised Kernel**
  - **MATRIX_SIZE = 2048** Speedup of about **214X with Optimised Kernel**
  - **MATRIX_SIZE = 4096** Speedup of about **218.5X with Optimised Kernel**
- **BLOCK_SIZE = 32**
  - **MATRIX_SIZE = 1024** Speedup of about **61X with Optimised Kernel**
  - **MATRIX_SIZE = 2048** Speedup of about **231X with Optimised Kernel**
  - **MATRIX_SIZE = 4096** Speedup of about **238.7X with Optimised Kernel**
- **BLOCK_SIZE = 32 with 8-way UNROLLING**
  - **MATRIX_SIZE = 1024** Speedup of about **66.3X with Optimised Kernel**
  - **MATRIX_SIZE = 2048** Speedup of about **259X with Optimised Kernel**
  - **MATRIX_SIZE = 4096** Speedup of about **272X with Optimised Kernel**

5 Problem 5

5.1 Compilation and execution instruction

```
export PATH=/usr/local/cuda-10.0/bin:${PATH}
nvcc -g -G -arch=sm_61 -std=c++11 assignment5-p5.cu -o assignment5-p5
./assignment5-p5
```

5.2 Performance Report and Explanation

Exact time details and different codes tried have been commented in code for all the below speedups

Best kernel gave a speedup of about **1.7X** with **N = 512**
Best kernel gave a speedup of about **1.5X** with **N = 64**

Optimisations

- Naive Version on (N = 64) gave varying performance from **1.6ms to 3.4ms**, serial took 2.5ms on average.
- The Optimised version exploits spacial locality by tiling and it gives improved and consistent performance for (N = 64) **1.5ms to 1.7ms**
- The optimised version exploits spacial locality but for boundary points, it go for the global memory and this is one of the possible bottleneck because of which we didn't get much improvement in this scenario.