Malay Pandey
CS 610A

Assignment IV

Roll No: 22111090
October 26, 2023

Problem 1

- **BlockSize:** `dim3(t, t, t)`, where t = {1, 2, 4, 8}
- **GridSize:** `dim3((N+t-1)/t, (N+t-1)/t, (N+t-1)/t)`

```
__global__ void kernel1(float* d_input, float* d_output) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  int j = threadIdx.y + blockIdx.y * blockDim.y;
  int k = threadIdx.z + blockIdx.z * blockDim.z;

  if (i >= 1 && i < N-1 && j >= 1 && j < N-1 && k >= 1 && k < N-1) {
    d_output[i*N*N + j*N + k] = 0.8f *
        (d_input[(i-1)*N*N + j*N + k] + d_input[(i+1)*N*N + j*N + k] +
         d_input[i*N*N + (j-1)*N + k] + d_input[i*N*N + (j+1)*N + k] +
         d_input[i*N*N + j*N + (k-1)] + d_input[i*N*N + j*N + (k+1)]);
  }
}
```

When we use CUDA kernel with shared memory, we are already saving the different indices of `d_input` into the shared memory, as a result of which, retrieving the values takes much less time compared to the Naive CUDA version. Also, using pinned memory proved beneficial for us, as it significantly reduced the time taken in `memcpy` function.

Version	Time (ms)	File Name
Sequential	1.974	22111090-prob1-v1.cu
CUDA (Naive) (t = 8)	0.603	22111090-prob1-v1.cu
CUDA (ShMem: 1)	0.892	22111090-prob1-v1.cu
CUDA (ShMem: 2)	0.225	22111090-prob1-v1.cu
CUDA (ShMem: 4)	0.216	22111090-prob1-v1.cu
CUDA (ShMem: 8)	0.231	22111090-prob1-v1.cu
CUDA (PiMem: 1)	2.265	22111090-prob1-v2.cu
CUDA (PiMem: 2)	0.178	22111090-prob1-v2.cu
CUDA (PiMem: 4)	**0.170**	22111090-prob1-v2.cu
CUDA (PiMem: 8)	0.183	22111090-prob1-v2.cu
CUDA (UVM: 1)	15.208	22111090-prob1-v3.cu
CUDA (UVM: 2)	5.694	22111090-prob1-v3.cu
CUDA (UVM: 4)	6.553	22111090-prob1-v3.cu
CUDA (UVM: 8)	3.592	22111090-prob1-v3.cu
Best Speedup	$\sim$ 11.612	-

```
==487== Profiling application: ./22111090-prob1-v1
==487== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   44.60%  1.0623ms         4  265.57us  88.190us  778.20us  kernel2(float const *, float*)
                   32.95%  784.82us         5  156.96us  156.48us  157.47us  [CUDA memcpy DtoH]
                   17.41%  414.62us         2  207.31us  163.20us  251.42us  [CUDA memcpy HtoD]
                    5.04%  120.13us         1  120.13us  120.13us  120.13us  kernel1(float*, float*)
```

Figure 1: nvprof (Naive + Shared Memory)

```
==472== Profiling application: ./22111090-prob1-v2
==472== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   49.47%  1.0645ms         4  266.12us  88.485us  780.55us  kernel(float const *, float*)
                   41.97%  902.96us         4  225.74us  205.64us  260.52us  [CUDA memcpy DtoH]
                    8.56%  184.14us         1  184.14us  184.14us  184.14us  [CUDA memcpy HtoD]
      API calls:   94.19%  436.41ms         3  145.47ms  15.600us  432.11ms  cudaMallocHost
```

Figure 2: nvprof Pinned Memory

```
==528== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  23.816ms         4  5.9540ms  2.7694ms  10.322ms  kernel(float const *, float*)
      API calls:   92.03%  453.25ms         3  151.08ms  50.800us  444.48ms  cudaMallocManaged
                    4.91%  24.196ms         4  6.0490ms  2.8004ms  10.410ms  cudaDeviceSynchronize
                    1.49%  7.3217ms         4  1.8304ms  43.900us  6.8881ms  cudaLaunchKernel
```

Figure 3: nvprof UVM

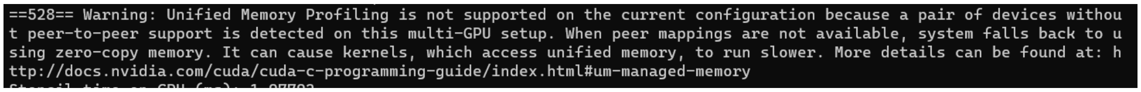

Figure 4: UVM issue

Here, we can see that the naive kernel (with t = 8), takes less time than Shared Memory Kernel with t = 1, but the performance of our Shared Memory Kernel significantly improves and it outperforms naive kernel from t = 2 onwards, since, we are increasing the number of elements being stored in the shared memory. Also, after using the Pinned Memory, we can see that the average time of CUDA `memcpy` falls down a bit. The complete nvprofiling details are attached in the zip file in the folder `nvprof_imgs`.

Please Note: The Unified Virtual Memory Version took so long because it was not supported on my Ubuntu virtualbox, as a result of which, the system falls back to using zero-copy memory, hence it runs a lot slower.

Command for running the code:

```
nvcc -lineinfo -res-usage -arch=sm_75 -std=c++14
22111090-prob1-<version>.cu -o 22111090-prob1-<version>
./22111090-prob1-<version>
```

Problem 2

Algorithm Used: Blelloch up-and-down sweep parallel scan algorithm (reference). An array of size N (2^{20}) has been initialized randomly, with each element in the range [0, 5]. The blocksize and gridsize to be used in the CUDA kernel are mentioned below respectively.

- **BlockSize:** `dim3(1024, 1, 1)`
- **GridSize:** `dim3((N+blockSize.x-1) / blockSize.x, 1, 1)`

Please Note: In the code provided, there's an assumption that the exclusive scan sum is well within the limits of `int`, we can change the data type accordingly to get exclusive scan of potentially larger arrays/values. Also, currently, the max value of N (size of array), can go upto 2^{20} elements, the algorithm fails to pass for sizes larger than that.

Version	Time (μs) (N = 2^{20})
Sequential	2847
CUDA (Blelloch + SHMEM)	**1027.33**
Thrust	6751
Best Speedup	$\sim$ 2.77

Command for running the code:

```
nvcc -lineinfo -res-usage -arch=sm_75 -std=c++14 22111090-prob2.cu
-o 22111090-prob2 ; ./22111090-prob2
```

Problem 3

The critical observation in this context is that due to the sorted nature of the iteration space $r_1, r_2, \ldots, r_{10}$, the values $x_1, x_2, \ldots, x_{10}$ are also stored in sorted order in the results file. Consequently, it is possible to initially store these values in a data structure such as a set, vector, array, etc., without sorting them during the parallel implementation. After the parallel execution is completed, the results will remain unchanged, allowing us to sort these values afterward to achieve the desired sorted order in the results.

The loops r1, r2 and r3 have been mapped one-to-one to threads, since we can launch a large number of threads on the GPU without worrying about context switching. For shared memory usage, since we have mapped the loops r1, r2 and r3 to the threads, it makes sense to store the values of x1, x2 and x3 to reduce the number of floating-point operations within the loops. However, we have to make sure that the Shared Memory usage within a block doesn't exceed the hardware limit. To ensure that, the following lines of code have been added.

```
int device;
cudaGetDevice(&device);
cudaDeviceProp prop;
cudaGetDeviceProperties(&prop, device);
size_t sharedMemoryPerBlock = prop.sharedMemPerBlock;
size_t sz = blockSize.x + blockSize.x*(blockSize.y + blockSize.y*blockSize.z);
// sz * 3 because we want to store x1, x2 and x3
assert((sz * 3 * sizeof(double)) <= sharedMemoryPerBlock);
```

- **BlockSize:** `dim3(x, y, z)`, where {x = 4, y = 4, z = 4}. These values can be changed with varying values of s1, s2 and s3 accordingly.

- **GridSize:** `dim3((s1+x-1)/x, (s2+y-1)/y, (s3+z-1)/z)`

Version	**Time (s)**	**File Name**
Sequential	380	22111090-prob3-v0.c
CUDA (Naive)	**102.133**	22111090-prob3-v1.cu
CUDA (Shared Memory)	129.478	22111090-prob3-v2.cu
Unified Virtual Memory	164.971	22111090-prob3-v3.cu
Thrust	128.010	22111090-prob3-v4.cu
Best Speedup	$\sim$ 3.721	-

Please Note:

- The Unified Virtual Memory Version took longer because it was not supported on my Ubuntu virtualbox, as a result of which, the system falls back to using zero-copy memory, hence it runs a lot slower.
- In the code provided, there's an assumption that maximum points to be written to text file is 20,000 (`const int max_results`). If we want to store more than these many points, we can simply change the value of this variable.

Command for running the code:

```
nvcc --extended-lambda -lineinfo -res-usage -arch=sm_75
-std=c++14 22111090-prob3-<version>.cu -o 22111090-prob3-<version>
./22111090-prob3-<version>
```

References

- Parallel Exclusive Scan - Hillis & Steel, Guy E. Blelloch

End.