DD2360 Assignment I: GPU architecture and basic programming environments

Akhmed Al-Sayed
akhmed@kth.se

Han Yang
hany2@kth.se

Exercise 1 – Your first CUDA program and GPU performance metrics

1. **Explain how the program is compiled and run.**

The program is compiled using `nvcc` command, which is a command to interface with CUDA compiler. The compiler produces an executable that will run a program both on the host (i.e. CPU) and the device (i.e. GPU). The precise commands used are:

```
nvcc hw2_ex1.cu -o add_vecs       # compile
./add_vecs 512                    # run
```

2. **For a vector length of N:**
   - **How many floating operations are being performed in your vector add kernel?**
   - **How many global memory reads are being performed by your kernel?**

For two vectors of size N each, we perform N floating operations. For each corresponding pair, we perform one operation. On the other hand, we end up performing 2N reads, as each element in each array is read exactly once.

3. **For a vector length of 512:**
   - **Explain how many CUDA threads and thread blocks you used.**
   - **Profile your program with Nvidia Nsight.**

In code, we defined 256 to be the number of threads to be used per block. Given that we have 512-elements sized arrays, we will end up using 2 blocks. This in turn leads to 512 threads being utilized.

To check achieved occupancy, we ran the binary through a profiler. This was done with a following command:

```
sudo ncu add_vecs 512
```

The occupancy achieved was 16.06%. A full table with occupancy results can be seen below:

```
Section: Occupancy
------------------------------- ----------- ------------
Metric Name                     Metric Unit Metric Value
------------------------------- ----------- ------------
Block Limit SM                        block           24
Block Limit Registers                 block           16
Block Limit Shared Mem                block           16
Block Limit Warps                     block            6
Theoretical Active Warps per SM        warp           48
Theoretical Occupancy                     %          100
Achieved Occupancy                        %        16.06
Achieved Active Warps Per SM           warp         7.71
------------------------------- ----------- ------------
```

4. **Now increase the vector length to 262140**:
   - **Did your program still work? If not, what changes did you make?**
   - **Explain how many CUDA threads and thread blocks you used.**
   - **Profile your program with Nvidia Nsight. What Achieved Occupancy do you get now?**

When running the script with longer vectors, the program still worked as intended. The only differences observed were the run times of the program. The addition time of vectors did increase from 150µs to 190µs, but the copy times (to and from a device) have received a much more significant hit with copy-to-device increasing from 40µs to 600µs and copy-from-device increasing from 12µs to 300µs.

We can find the number of CUDA threads and thread blocks we used in the same way as we did previously. With a vector length of 262,140 and threads-per-block count, we obtain 1024 blocks. Since each block will launch 256 threads, we will get a total of $262,144$ ($= 1024 \times 256$) total threads.

The total occupancy registered by the profiler is 86.61%. A full table of occupancy results can be seen below:

```
Section: Occupancy
------------------------------- ----------- ------------
Metric Name                     Metric Unit Metric Value
------------------------------- ----------- ------------
Block Limit SM                        block           24
Block Limit Registers                 block           16
Block Limit Shared Mem                block           16
Block Limit Warps                     block            6
Theoretical Active Warps per SM        warp           48
Theoretical Occupancy                     %          100
Achieved Occupancy                        %        86.61
Achieved Active Warps Per SM           warp        41.57
------------------------------- ----------- ------------
```

Exercise 2 – 2D Dense Matrix Multiplication

1. **Name three applications domains of matrix multiplication.**

   - Computer Graphics - for rotating, scaling and translating objects in 2D/3D space;
   - Neural Networks - computing outputs between layers using weights matrices;
   - Economic Analysis - modeling relationships between different sectors of an economy.

2. **How many floating operations are being performed in your matrix multiply kernel?**

   For a matrix multiplication C = A × B, where:
   - A is a (numARows × numAColumns) matrix
   - B is a (numBRows × numBColumns) matrix
   - C is a (numCRows × numBColumns) matrix
   - numAColumns = numBRows

   For each element C[i][j]:
   - We perform numAColumns multiplications (1 FLOP each)
   - We perform (numAColumns - 1) additions to sum up the products (1 FLOP each)
   - Total per element: (2 × numAColumns - 1) FLOPs

   For the entire matrix C, total FLOPs = numARows × numBColumns × (2 × numAColumns - 1)

3. **How many global memory reads are being performed by your kernel?**

   In the implementation of gemm, we compute one element of matrix C:

   ```
   sum += A[row * numAColumns + k] * B[k * numBColumns + col];
   ```

   For each thread (which computes one element of C):
   - Global memory reads from A: numAColumns reads
   - Global memory reads from B: numAColumns reads
   - Total global memory reads per thread: 2 × numAColumns

   Total global memory reads = (numARows × numBColumns) × (2 × numAColumns)

4. **For a matrix A of (64x128) and B of (128x64):**
   - **Explain how many CUDA threads and thread blocks you used.**
   - **Profile your program with Nvidia Nsight. What Achieved Occupancy did you get?**

   For a matrix A of (64x128) and B of (128x64), Matrix C will be 64x64.

   ```
   dim3 blockDim(16, 16);  // 256 threads per block
   dim3 gridDim(
       (64 + 16 - 1) / 16,  // ceil(64/16) = 4 blocks
       (64 + 16 - 1) / 16   // ceil(64/16) = 4 blocks
   );
   ```

   Each block: $16 * 16 = 256$ threads; Grid: $4 * 4 = 16$ blocks Total threads: 16 blocks × 256 threads = 4096 threads

```
Section: Occupancy
  ----------------------------- ---------- -----------
  Metric Name                   Metric Unit Metric Value
  ----------------------------- ---------- -----------
  Block Limit SM                     block          16
  Block Limit Registers              block           4
  Block Limit Shared Mem             block          16
  Block Limit Warps                  block           4
  Theoretical Active Warps per SM     warp          32
  Theoretical Occupancy                  %         100
  Achieved Occupancy                     %       24.61
  Achieved Active Warps Per SM        warp        7.88
  ----------------------------- ---------- -----------
```

5. **For a matrix A of (1024x1023) and B of (1023x8193):**
   - **Did your program still work? If not, what changes did you make?**
   - **Explain how many CUDA threads and thread blocks you used.**
   - **Profile your program with Nvidia Nsight. What Achieved Occupancy do you get now?**

   My program still works but with much longer running time. The grid dimensions is (513, 64) and the number of blocks is 32,832. The number of total threads is $32832 \times 256 = 8404992$ with 256 threads per block.

   ```
   Section: Occupancy
     ----------------------------- ---------- -----------
     Metric Name                   Metric Unit Metric Value
     ----------------------------- ---------- -----------
     Block Limit SM                     block          16
     Block Limit Registers              block           4
     Block Limit Shared Mem             block          16
     Block Limit Warps                  block           4
     Theoretical Active Warps per SM     warp          32
     Theoretical Occupancy                  %         100
     Achieved Occupancy                     %       99.50
     Achieved Active Warps Per SM        warp       31.84
     ----------------------------- ---------- -----------
   ```

6&7. **Further increase the size of matrix A and B and change DataType from double to float, plot a stacked bar chart showing the breakdown of time including (1) data copy from host to device (2) the CUDA kernel (3) data copy from device to host. For this, you will need to add simple CPU timers to your code regions. Explain what you observe.**

The following figure shows the breakdown of runtime with float and double data type. Some finding:

- Kernel execution time dominates as matrix size increases ($O(N^3)$ complexity)
- Memory transfer times (H2D and D2H) scale more linearly ($O(N^2)$)
- Double precision operations take approximately 2x longer for kernel execution
- Memory transfer times are also roughly doubled due to twice the data size
- The relative proportions of H2D:Kernel:D2H remain similar
- Double precision impact becomes more pronounced at larger matrix sizes

Scaling behavior:

- Memory transfer times scale with matrix area (N^2)

- Computation times scale with matrix multiply complexity (N^3)

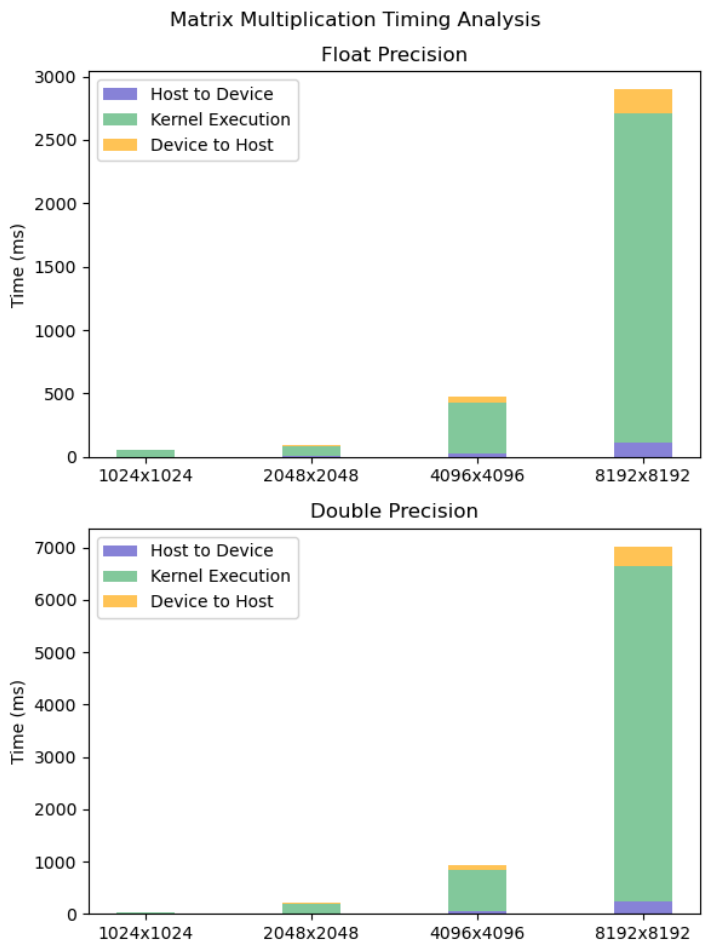

Figure 1: Breakdown of Runtime with Different DataType.