Applied GPU Programming - Assignment 2

Group 19 - Diogo Gaspar && Diogo Melita

Contributions:
We worked together, in person, so both of us had the same workload and worked on the same (all) exercises.

Diogo Melita - `diogogm@kth.se` - 50%
Diogo Gaspar - `dgaspar@kth.se` - 50%

Exercise 1 - Your first CUDA program and GPU performance metrics

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

The program is compiled by running `nvcc -o hw2_ex1 hw2_ex1.cu`. To run it, we use this same executable file; we run `./hw2_ex1 <vector_length>`, with the argument defining the length of both the to-be-handled vectors.

2. **For a vector length of N:**
    - **How many floating operations are being performed in your vector add kernel?**
        - We're dealing with two vectors and doing an add operation for each of their entries (and they both have length N); as such, we end up performing N add floating operations.
    - **How many global memory reads are being performed by your kernel?**
        - We must read an entry per vector (there are two vectors) per operation; with us having N operations, that means that there'll be 2N global memory reads performed by the kernel.

3. **For a vector length of 512:**
    - **Explain how many CUDA threads and thread blocks you used.**

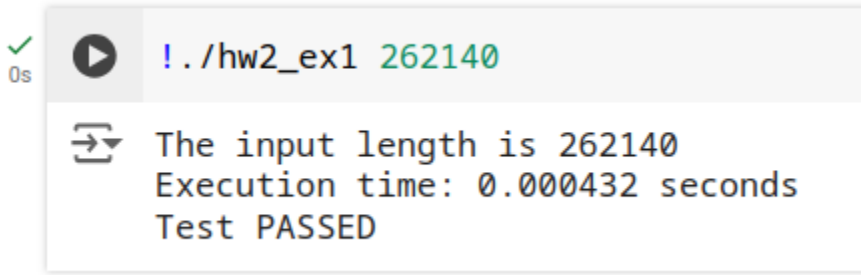
- We define TPB, as a global variable, as 256. We also have that the number of thread blocks is given by $\frac{inputLength+TPB-1}{TPB}$, which with these values will amount to $\lfloor \frac{512+256-1}{256} \rfloor = 2$. Furthermore, the number of CUDA threads is given by the product between number of thread blocks and TPB, which here amounts to 512.

- **Profile your program with Nvidia Nsight. What Achieved Occupancy did you get?**
  - To profile the program, we ran `ncu --metrics sm__warps_active.avg.pct_of_peak_sustained_active ./hw2_ex1 512` . The achieved occupancy we attained was 23.94%, as can be seen in the image below.

```
The input length is 512
==PROF== Connected to process 7082 (/content/drive/MyDrive/Colab_Notebooks/apgDD2360/assignments/assignment2/hw2_ex1)
==PROF== Profiling "vecAdd" - 0: 0%....50%....100% - 1 pass
Execution time: 0.235981 seconds
Test PASSED
==PROF== Disconnected from process 7082
[7082] hw2_ex1@127.0.0.1
  vecAdd(double *, double *, double *, int) (2, 1, 1)x(256, 1, 1), Context 1, Stream 7, Device 0, CC 7.5
    Section: Command line profiler metrics
    ---------------------------------------------- ----------- ------------
    Metric Name                                    Metric Unit Metric Value
    ---------------------------------------------- ----------- ------------
    sm__warps_active.avg.pct_of_peak_sustained_active           %        23.94
    ---------------------------------------------- ----------- ------------
```

4. **Now increase the vector length to 262140:**

   - **Did your program still work? If not, what changes did you make?**
     - Yes, it still worked.



   - **Explain how many CUDA threads and thread blocks you used.**
     - Using the same formulas we previously mentioned, the number of threads blocks is given by $\lfloor \frac{262140+256-1}{256} \rfloor = 1024$; the number of

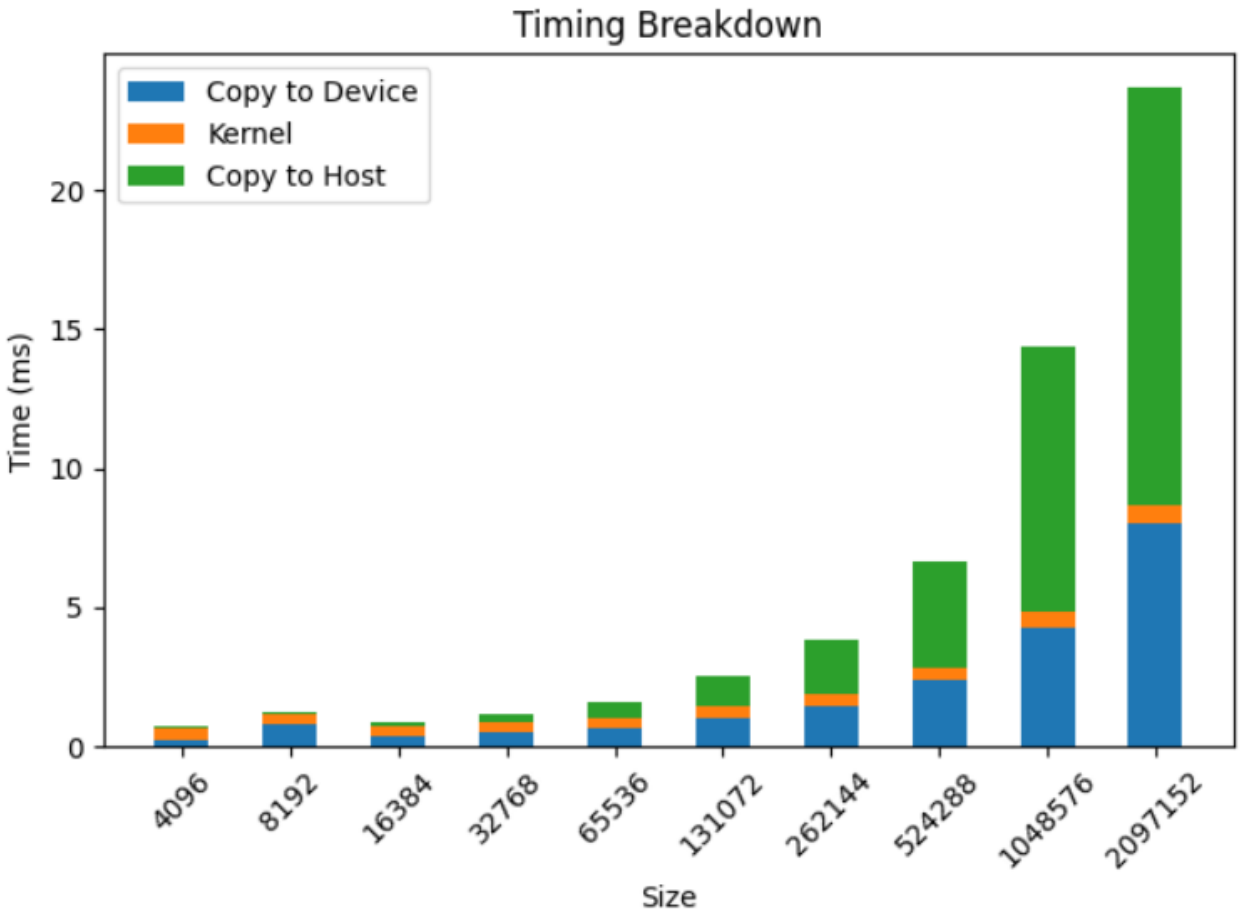
CUDA threads is, then, $1024 \times 256 = 262144$.

- **Profile your program with Nvidia Nsight. What Achieved Occupancy do you get now?**
  - As can be seen in the image below, we not get an Achieved Occupancy of 82.38%. We ran the same command as before to profile, replacing 512 with 262140.

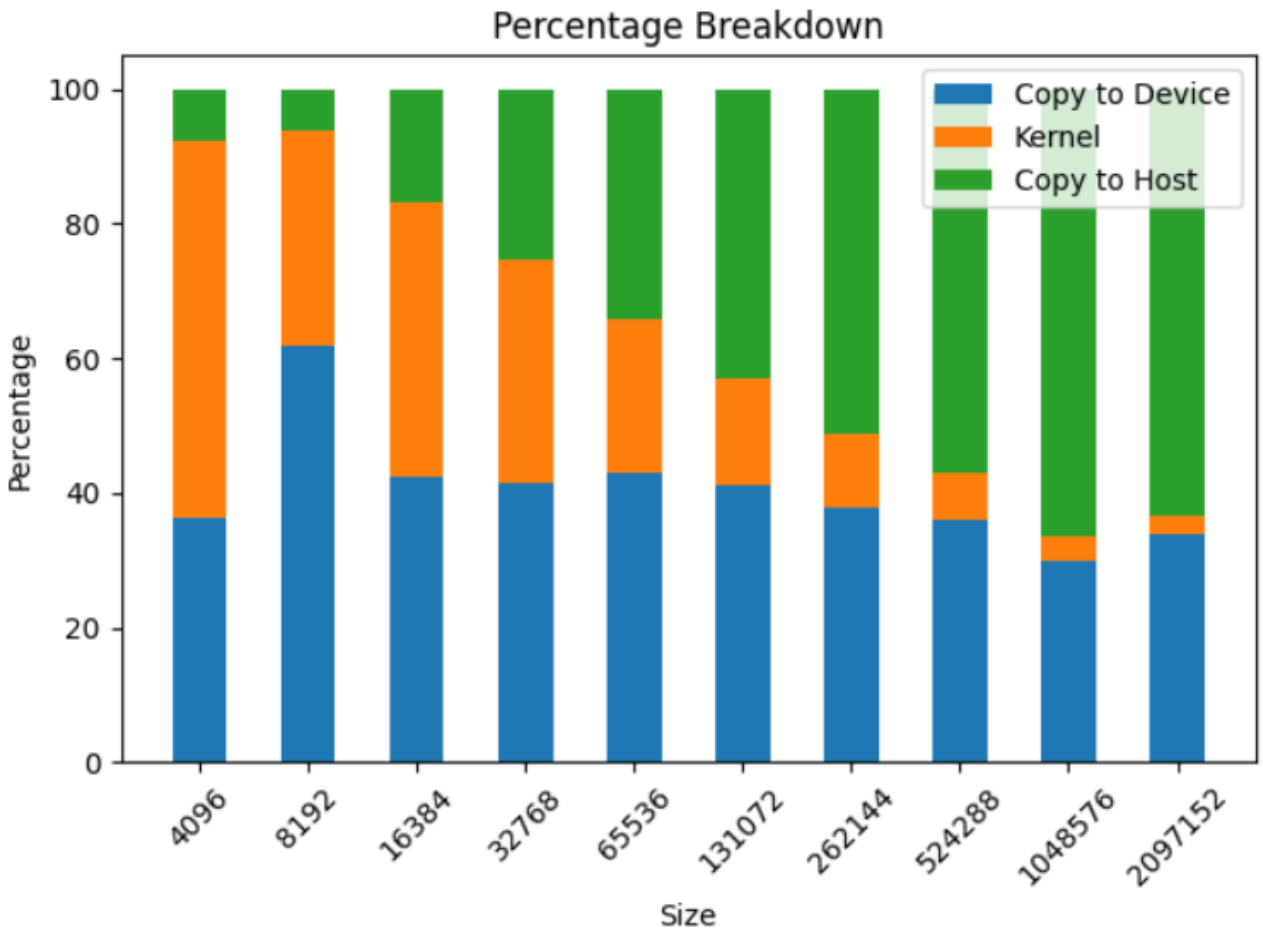
```
The input length is 262140
==PROF== Connected to process 7949 (/content/drive/MyDrive/Colab_Notebooks/apgDD2360/assignments/assignment2/hw2_ex1)
==PROF== Profiling "vecAdd" - 0: 0%....50%....100% - 1 pass
Execution time: 0.230911 seconds
Test PASSED
==PROF== Disconnected from process 7949
[7949] hw2_ex1@127.0.0.1
  vecAdd(double *, double *, double *, int) (1024, 1, 1)x(256, 1, 1), Context 1, Stream 7, Device 0, CC 7.5
    Section: Command line profiler metrics
    ---------------------------------------------------- ----------- ------------
    Metric Name                                          Metric Unit Metric Value
    ---------------------------------------------------- ----------- ------------
    sm__warps_active.avg.pct_of_peak_sustained_active           %        82.38
    ---------------------------------------------------- ----------- ------------
```

5. **Further increase the vector length (try 6-10 different vector length), 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.**

In the first image, we observe the breakdown of time spent on each of the three metrics, showcasing their individual contributions in milliseconds.



The second image provides a clear visualization of the percentage of time each metric consumes relative to the total execution time, offering a comparative perspective.



Exercise 2 - 2D Dense Matrix Multiplication

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

Machine learning, in particular multiplying weight matrices with input data which allows networks to learn; in computer graphics, to be able to rotate, scale images, among many other possibilities; in data mining and related fields, matrix multiplication is used to perform techniques such as Principal Component Analysis (PCA).

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

First, we have the following for matrix dimensions:

- **Matrix A**: `numARows x numAColumns`
- **Matrix B**: `numBRows x numBColumns`
- **Matrix C** (Result): `numCRows x numCColumns` , where:

- `numCRows = numARows`
- `numCColumns = numBColumns`

Each element $C[i][j]$ requires:

- **`numAColumns` multiplications**
- **`numAColumns - 1` additions**

Thus, for each $C[i][j]$:

$$\text{FLOPs per element} = 2 \times \text{numAColumns} - 1$$

Since C has `numCRows * numCColumns` elements, the total FLOPs are:

$$\text{Total FLOPs} = (\text{numCRows} \times \text{numCColumns}) \times (2 \times \text{numAColumns} - 1)$$

For matrix dimensions represented as **M** (rows of A), **K** (columns of A), **N** (columns of B), we can write the **approximate** total FLOPs as:

$$\text{Total FLOPs} \approx 2 \times M \times N \times K$$

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

Let's keep the same notation for the matrices' columns and rows.

From Matrix A, each thread responsible for a result element $C[i][j]$ needs all K elements in row i of A. Therefore, for $M \times N$ result elements this results in $M \times K \times N$.

From Matrix B, each thread responsible for $C[i][j]$ needs all K elements in column j of B. Similarly, this results in $M \times K \times N$.

This said, the total number of

$$\text{Total global memory reads} = 2 \times M \times K \times N$$

4. **For a matrix A of (64×128) and B of (128×64):**
    - **Explain how many CUDA threads and thread blocks you used.**
        - Having sizes as follows: **A**: 64 rows, 128 columns; **B**: 128 rows, 64

columns; leads to **C**'s size having 64 rows and 64 columns. Furthermore, the kernel uses a block size of `16x16` (total 256 threads per block), since `TPB = 256`. The number of blocks along the x- and y-directions (for columns of **C**) is calculated in the same way, $\text{gridSize.coord} = \lceil \frac{64}{16} \rceil = 4$. Thus, the grid size is 4×4 (16 blocks in total). As for threads, we have **Total threads** $= 4$ (blocks in x) $\times 4$ (blocks in y) $\times 256$ (threads in each block) $= 4096$ threads, with **Total blocks** $= 4$ (x) $\times 4$ (y) $= 16$ blocks.

- **Profile your program with Nvidia Nsight. What Achieved Occupancy did you get?**
  - We ended up achieving an occupancy of 96.01%.

```
Input matrix dim (64 x 128) (128 x 64) (64 x 64)
==PROF== Connected to process 2282 (/content/hw2_ex2)
Time for host to device memory copy: 0.000171 seconds
==PROF== Profiling "gemm" - 0: 0%....50%....100% - 1 pass
Time for kernel execution: 0.332970 seconds
Time for device to host memory copy: 0.000109 seconds
Test PASSED
==PROF== Disconnected from process 2282
[2282] hw2_ex2@127.0.0.1
  gemm(double *, double *, double *, int, int, int, int) (2, 2, 1)x(32, 32, 1), Context 1, Stream 7, Device 0, CC 7.5
    Section: Command line profiler metrics
    ------------------------------------------------- ----------- ------------
    Metric Name                                       Metric Unit Metric Value
    ------------------------------------------------- ----------- ------------
    sm__warps_active.avg.pct_of_peak_sustained_active           %        96.01
    ------------------------------------------------- ----------- ------------
```

5. F**or a matrix A of (1024×1023) and B of (1023×8193):**

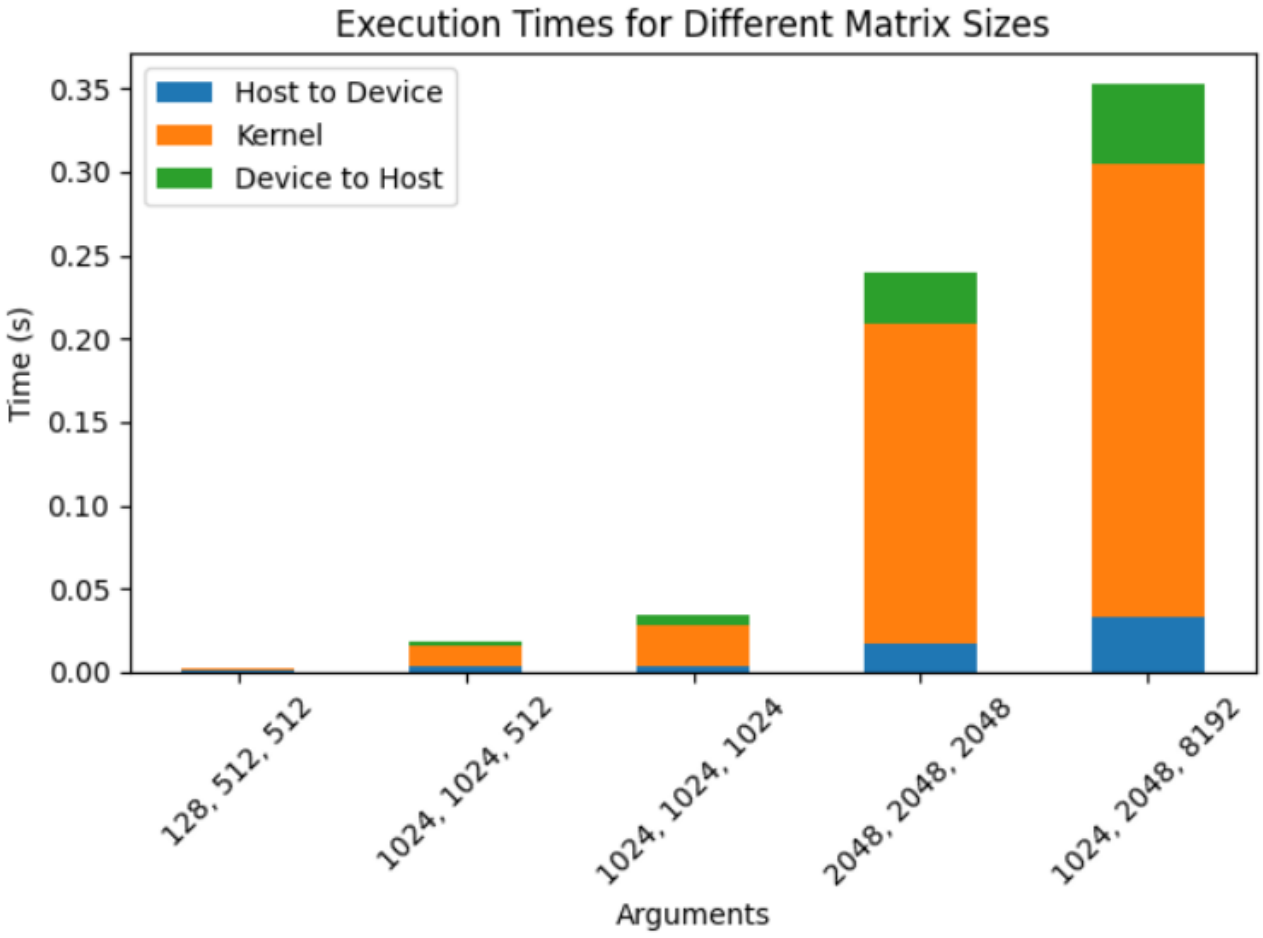
- **Did your program still work? If not, what changes did you make?** Yes.
- **Explain how many CUDA threads and thread blocks you used.**
  - Given sizes: **A**: 1024 rows, 1023 columns; **B**: 1023 rows, 8193 columns; leads to **C**'s size having 1024 rows and 8193 columns. Furthermore, the kernel uses a block size of `16x16` (total 256 threads per block), since `TPB = 256`. The number of blocks along the x- and y-directions (for columns of **C**) is calculated in the same way, $\text{gridSize.coord} = \lceil \frac{1024}{16} \rceil = 64$ and $\lceil \frac{8193}{16} \rceil = 513$. Thus, the grid size is 513×64 $(= 32,732$ blocks in total$)$. As for threads, we have **Total threads** $= 513$ (blocks in x) $\times 64$ (blocks in y) $\times 256$ (threads in each block) $= 8,404,992$ threads, with **Total blocks** $= 513$ (x) $\times 64$ (y) $= 32,732$ blocks.

- **Profile your program with Nvidia Nsight. What Achieved Occupancy do you get now?**
  - We ended up achieving an occupancy of 98.30%.

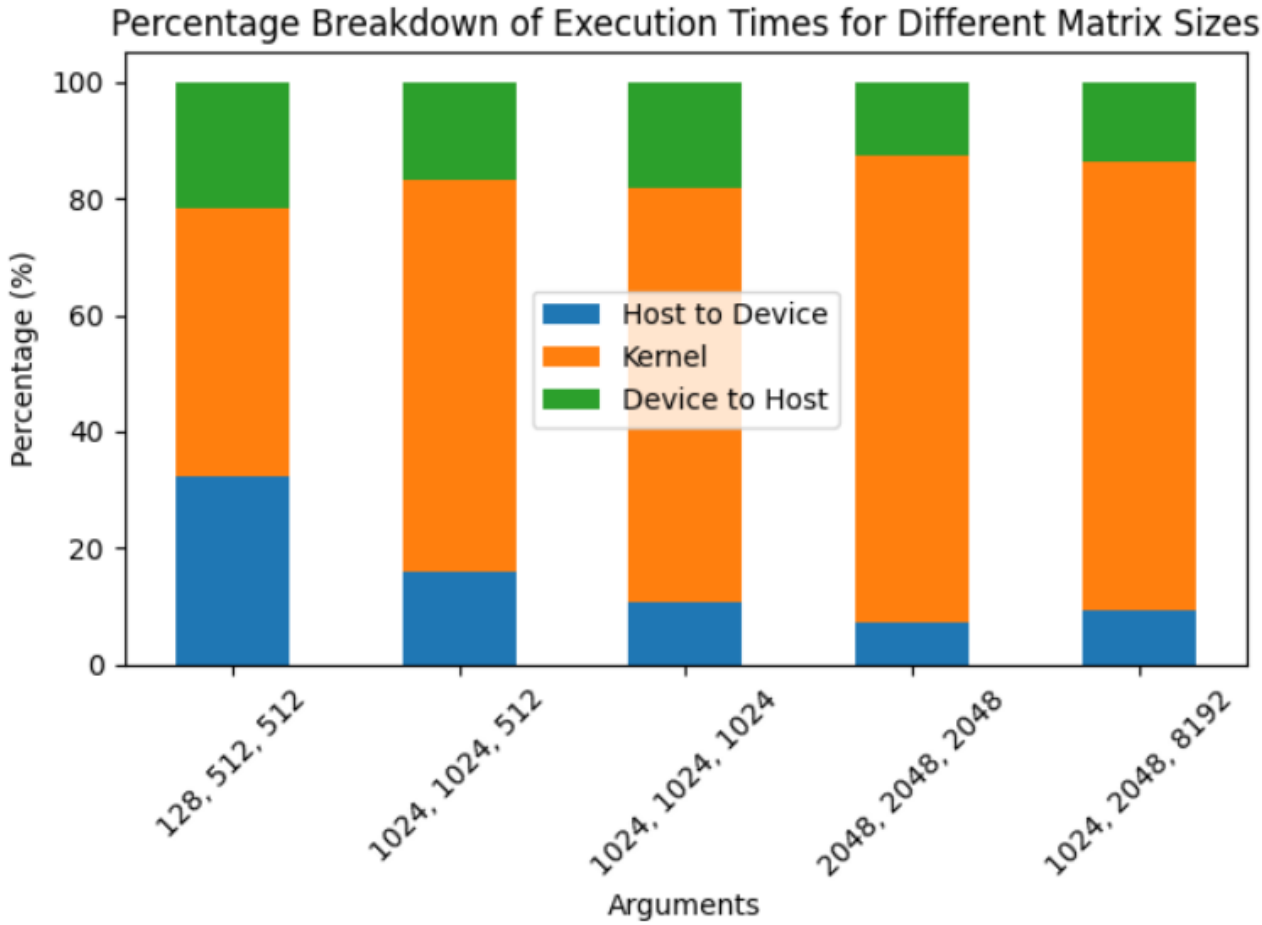
```
Input matrix dim (1024 x 1023) (1023 x 8193) (1024 x 8193)
==PROF== Connected to process 2729 (/content/hw2_ex2)
Time for host to device memory copy: 0.017173 seconds
==PROF== Profiling "gemm" - 0: 0%....50%....100% - 1 pass
Time for kernel execution: 0.433531 seconds
Time for device to host memory copy: 0.057093 seconds
Test PASSED
==PROF== Disconnected from process 2729
[2729] hw2_ex2@127.0.0.1
  gemm(double *, double *, double *, int, int, int, int) (257, 32, 1)x(32, 32, 1), Context 1, Stream 7, Device 0, CC 7.5
    Section: Command line profiler metrics
    ------------------------------------------------- ----------- ------------
    Metric Name                                       Metric Unit Metric Value
    ------------------------------------------------- ----------- ------------
    sm__warps_active.avg.pct_of_peak_sustained_active           %        98.30
    ------------------------------------------------- ----------- ------------
```

6. **Further increase the size of matrix A and B, 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.**

In the first image, we observe the breakdown of time spent on each of the three metrics, showcasing their individual contributions in milliseconds.



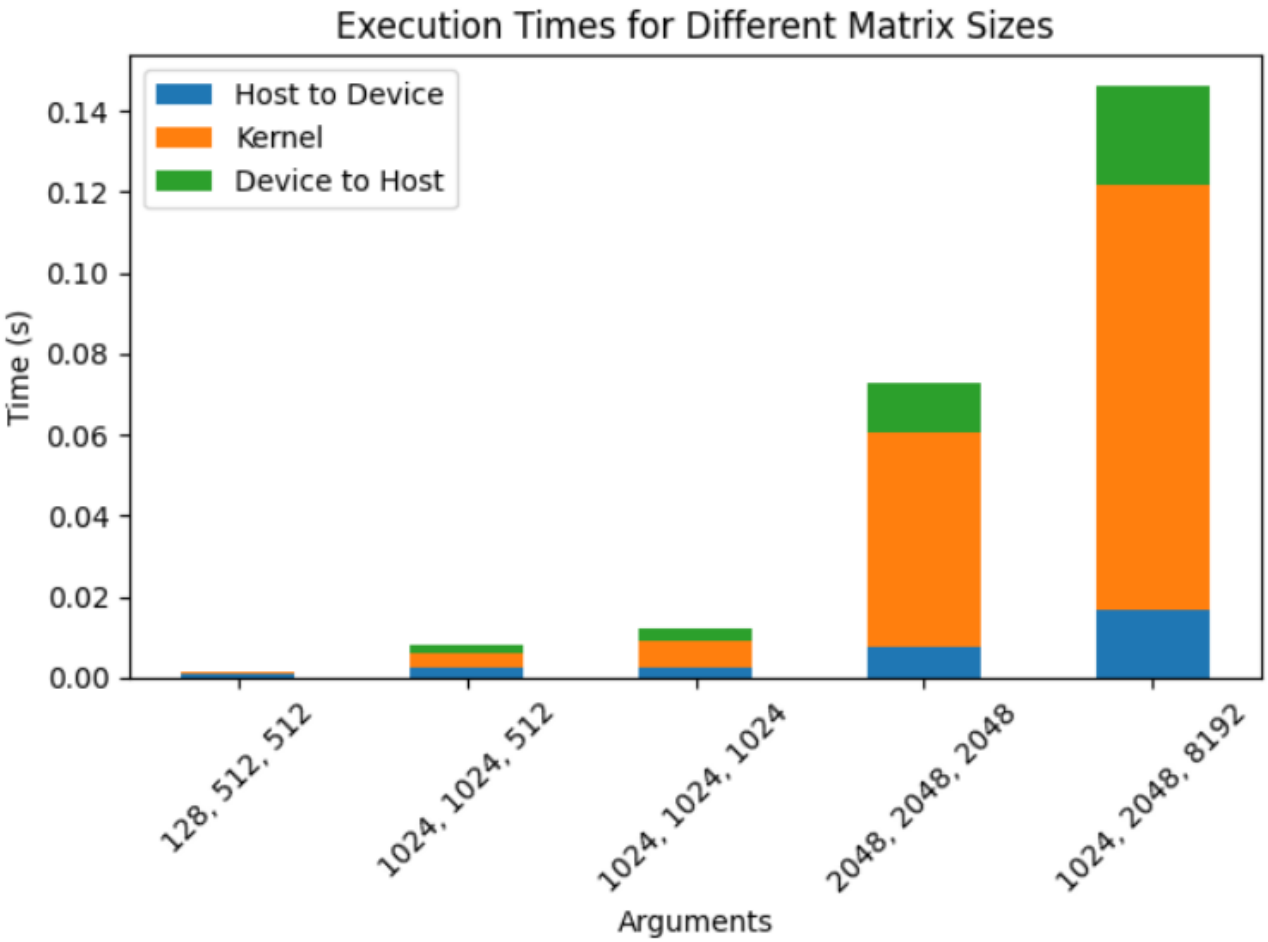
The second image provides a clear visualization of the percentage of time each metric consumes relative to the total execution time, offering a comparative perspective.



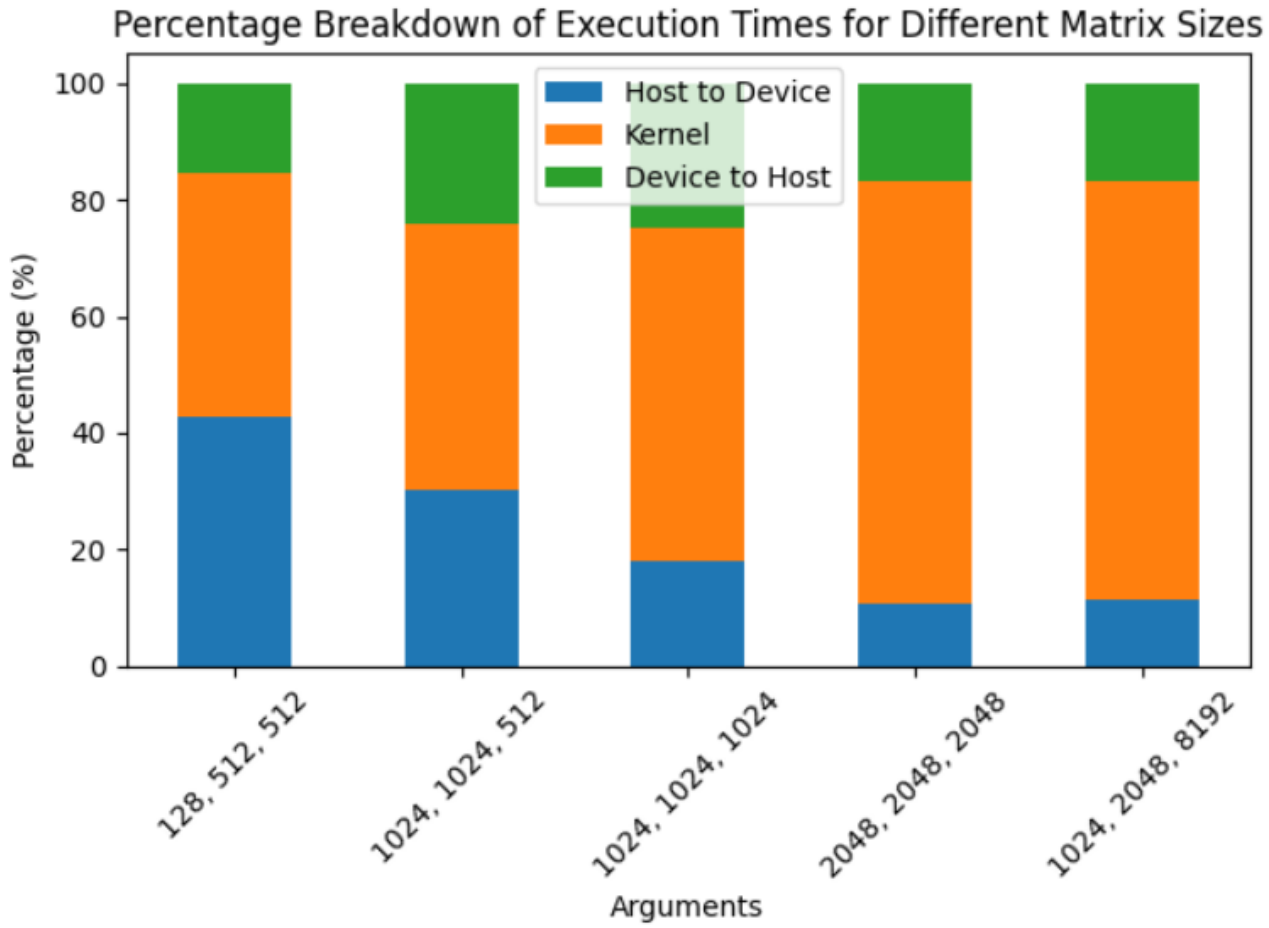
In all cases, a large chunk of the time is spent on the kernel, with all but the first scenario actually spending more than half of the time in it. Furthermore, we can see how it progressively increases as the matrix gets bigger, with the exception of the last two matrix sizes, where the `#rows` of the first (2048×2048) is larger than the second's (1024×8192), and vice-versa for the `#columns`: yet, the second's kernel time is larger than the first one.

7. **Now, change DataType from double to float, re-plot the a stacked bar chart showing the time breakdown. Explain what you observe.**

In the first image, we observe the breakdown of time spent on each of the three metrics, showcasing their individual contributions in milliseconds.



The second image provides a clear visualization of the percentage of time each metric consumes relative to the total execution time, offering a comparative perspective.



The trends between double and floats are very similar regarding the percentage breakdowns of execution times – same matrix sizes seem to spend similar slices of time on each metric. What we can see, though, is that in total execution time (ms), changing to float seems to effectivelt half (or better!) the total execution times spent for the same matrix size; this should be because floats only occupy half the size of doubles, thus the GPU can then process more data in parallel.