SA-MIRI 2025

Practice Pd: CUDA

Jakub Seliga (jakub.seliga@estudiantat.upc.edu)
Thomas Aubertier (thomas.aubertier@estudiantat.upc.edu)

Task 5.1 Your First Hello World in CUDA

- CUDA already installed on `alogin1` sessions only !
- CUDA programs must use `nvcc`

```
$ module load cuda
```

```
[nct01029@alogin1 ~]$ which nvcc
/usr/local/cuda-12.2/bin/nvcc
```

```
$ nvcc hello_world_CUDA.cu -o h_CUDA
```

```
[nct01029@alogin1 ~]$ ./h_CUDA
Hello World from CPU !
Hello World from GPU !
Hello World from GPU !
Hello World from GPU !
```

Task 5.2 Dimensionality of a Thread Block and Grid

- Same commands as before

```
[nct01029@alogin1 ~]$ ./p_CUDA

 Check grid + block dimension from HOST side :
grid.x=2 grid.y=1 grid.z=1 | block.x=3 block.y=1 block.z=1

 Check grid + block dimension from KERNEL side :
threadIdx:(0,0,0) blockIdx:(0,0,0) blockDim:(3,1,1) gridDim:(2,1,1)
threadIdx:(1,0,0) blockIdx:(0,0,0) blockDim:(3,1,1) gridDim:(2,1,1)
threadIdx:(2,0,0) blockIdx:(0,0,0) blockDim:(3,1,1) gridDim:(2,1,1)
threadIdx:(0,0,0) blockIdx:(1,0,0) blockDim:(3,1,1) gridDim:(2,1,1)
threadIdx:(1,0,0) blockIdx:(1,0,0) blockDim:(3,1,1) gridDim:(2,1,1)
threadIdx:(2,0,0) blockIdx:(1,0,0) blockDim:(3,1,1) gridDim:(2,1,1)
```

- Result is coherent with 3 threads for each 2 blocks

Task 5.3 Investigating Parallel Execution with Multiple Threads

- Code that adds integers, modified to run on multiple threads:

  add<<<1,4>>>(d_a, d_b, d_c);

  ```
  $ nvcc add.cu -o add
  ```

```
[nct01042@alogin1 Chapter.05]$ ./add
GPU: computed 2 + 7 = 9
GPU: computed 2 + 7 = 9
GPU: computed 2 + 7 = 9
GPU: computed 2 + 7 = 9
CPU: received result 2 + 7 = 9
```

- All four print identical messages, because they’re all performing the same operation on the same data
-
- When multiple threads access same memory without coordination (like atomic ops, synchronization, or separate memory per thread), simultaneous access causes race conditions and nondeterministic results

Task 5.4 Element-wise Vector Addition Using CUDA

- Code initializes vectors, host, allocates and copies them to the device, launches the kernel, and retrieves the result.

```
__global__ void add(int *a, int *b, int *c, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n)
        c[index] = a[index] + b[index];
}
```

```
int main(void) {
    int *a, *b, *c;             // host copies of a, b, c
    int *d_a, *d_b, *d_c;       // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c
    a = (int *)malloc(size);
    b = (int *)malloc(size);
    c = (int *)malloc(size);
```

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);


add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a,d_b,d_c,N);
  add<<< ... >>>(...);


// Copy result back to the host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

printf("vector c:\n");
print_vector(N, c);
```

Task 5.4 Element-wise Vector Addition Using CUDA

- Code initializes vectors, host, allocates and copies them to the device, launches the kernel, and retrieves the result.

```
$ nvcc add_vectors.cu -o add_vectors
```

```
[nct01042@alogin1 Chapter.05]$ ./add_vectors
vector a:
|0|1|2|3|4|5|6|7|8|9|10|11|12|13|14|15|16|17|18|19|20|21|22|23|24|25|26|27|28|29|30|31|32|33|34|35|36
|37|38|39|40|41|42|43|44|45|46|47|48|49|50|51|52|53|54|55|56|57|58|59|60|61|62|63
vector b:
|0|1|2|3|4|5|6|7|8|9|10|11|12|13|14|15|16|17|18|19|20|21|22|23|24|25|26|27|28|29|30|31|32|33|34|35|36
|37|38|39|40|41|42|43|44|45|46|47|48|49|50|51|52|53|54|55|56|57|58|59|60|61|62|63
vector c:
|0|2|4|6|8|10|12|14|16|18|20|22|24|26|28|30|32|34|36|38|40|42|44|46|48|50|52|54|56|58|60|62|64|66|68|
70|72|74|76|78|80|82|84|86|88|90|92|94|96|98|100|102|104|106|108|110|112|114|116|118|120|122|124|126
```

Task 5.5 Parallel Matrix Multiplication with CUDA

- Example with N=4

```
[nct01029@alogin1 ~]$ ./m_CUDA
***** MATRIX A *****
75.000000,52.000000,79.000000,77.000000
61.000000,91.000000,98.000000,3.000000
41.000000,36.000000,93.000000,51.000000
83.000000,7.000000,46.000000,6.000000
***** MATRIX B *****
85.000000,24.000000,59.000000,86.000000
55.000000,63.000000,89.000000,66.000000
6.000000,46.000000,26.000000,90.000000
83.000000,39.000000,12.000000,21.000000
***** MATRIX C *****
16100.000000,11713.000000,12031.000000,18609.000000
11027.000000,11822.000000,14282.000000,20135.000000
10256.000000,9519.000000,8653.000000,15343.000000
8214.000000,4783.000000,6788.000000,11866.000000
Execution time with N=4 processes: 183.85 ms
```

rad CALCULATION

$$\begin{bmatrix} 61 & 91 & 98 & 3 \\ 41 & 36 & 93 & 51 \\ 83 & 7 & 46 & 6 \end{bmatrix} \begin{bmatrix} 55 & 63 & 89 & 66 \\ 6 & 46 & 26 & 90 \\ 83 & 39 & 12 & 21 \end{bmatrix}$$

$$\begin{bmatrix} 16100 & 11713 & 12031 & 18609 \\ 11027 & 11822 & 14282 & 20135 \\ 10256 & 9519 & 8653 & 15343 \\ 8214 & 4783 & 6788 & 11866 \end{bmatrix}$$

Task 5.5 Parallel Matrix Multiplication with CUDA

- Example with `N=512`

```
[nct01029@alogin1 ~]$ ./m_CUDA
Execution time with N=512 processes: 186.54 ms
```

- Example with `N=4096`

```
[nct01029@alogin1 ~]$ ./m_CUDA
Execution time with N=4096 processes: 923.97 ms
```

- Note : higher values of N overflow `int` memory space and require more code optimisations.

Task 5.6 Running CUDA Jobs with SLURM

- We can use the same template

```
#!/bin/bash
#SBATCH -J matrix_mult
#SBATCH -t 00:15
#SBATCH -o %x_%J.out
#SBATCH -e %xo_%J.err
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --gres=gpu:1
#SBATCH --exclusive
#SBATCH --account nct_345
#SBATCH --qos gp_debug

module load cuda

echo "[ nvcc matrix_mult_CUDA.cu -o m_CUDA ]"
nvcc matrix_mult_CUDA.cu -o m_CUDA

echo "[ ./m_CUDA ]"
./m_CUDA
```

```
[nct01029@alogin1 ~]$ cat matrix_mult_30706301.out
[ nvcc matrix_mult_CUDA.cu -o m_CUDA ]
[ ./m_CUDA ]
Execution time with N=256 processes: 167.79 ms
```

Task 5.7 Profiling Matrix Multiplication on the GPU

```
[nct01029@login1 ~]$ nsys nvprof ./m_CUDA
WARNING: m_CUDA and any of its children processes will be profiled.

Execution time with N=256 processes: 260.50 ms
Generating '/scratch/tmp/nsys-report-7487.qdstrm'
[1/7] [========================100%] report1.nsys-rep
[2/7] [========================100%] report1.sqlite
[3/7] Executing 'nvtx_sum' stats report
SKIPPED: /gpfs/home/nct/nct01029/report1.sqlite does not contain NV Tools Extension (NVTX) data.
[4/7] Executing 'cuda_api_sum' stats report

 Time (%)  Total Time (ns)  Num Calls   Avg (ns)    Med (ns)   Min (ns)  Max (ns)  StdDev (ns)           Name
 --------  ---------------  ---------  ----------  ----------  --------  --------  -----------  ---------------------
     68.5         89294753          3  29764917.7      3095.0      1911  89289747   51550014.4  cudaMalloc
     30.7         39952298          1  39952298.0  39952298.0  39952298  39952298          0.0  cudaDeviceReset
      0.4           526144          3    175381.3     79653.0     70798    375693     173531.5  cudaMemcpy
      0.3           336561          1    336561.0    336561.0    336561    336561          0.0  cudaLaunchKernel
      0.1           136986          3     45662.0      8612.0      3437    124937      68702.9  cudaFree
      0.0            22969          1     22969.0     22969.0     22969     22969          0.0  cudaDeviceSynchronize
      0.0             2900          1      2900.0      2900.0      2900      2900          0.0  cuCtxSynchronize
      0.0             1647          1      1647.0      1647.0      1647      1647          0.0  cuModuleGetLoadingMode

[5/7] Executing 'cuda_gpu_kern_sum' stats report

 Time (%)  Total Time (ns)  Instances  Avg (ns)  Med (ns)  Min (ns)  Max (ns)  StdDev (ns)                      Name
 --------  ---------------  ---------  --------  --------  --------  --------  -----------  ----------------------------------------------
    100.0            20544          1   20544.0   20544.0     20544     20544          0.0  matrixProduct(double *, double *, double *, int)

[6/7] Executing 'cuda_gpu_mem_time_sum' stats report

 Time (%)  Total Time (ns)  Count  Avg (ns)  Med (ns)  Min (ns)  Max (ns)  StdDev (ns)      Operation
 --------  ---------------  -----  --------  --------  --------  --------  -----------  ------------------
     64.7            23712      2   11856.0   11856.0     11776     11936        113.1  [CUDA memcpy HtoD]
     35.3            12960      1   12960.0   12960.0     12960     12960          0.0  [CUDA memcpy DtoH]

[7/7] Executing 'cuda_gpu_mem_size_sum' stats report

 Total (MB)  Count  Avg (MB)  Med (MB)  Min (MB)  Max (MB)  StdDev (MB)      Operation
 ----------  -----  --------  --------  --------  --------  -----------  ------------------
      1.049      2     0.524     0.524     0.524     0.524        0.000  [CUDA memcpy HtoD]
      0.524      1     0.524     0.524     0.524     0.524        0.000  [CUDA memcpy DtoH]

Generated:
    /gpfs/home/nct/nct01029/report1.nsys-rep
    /gpfs/home/nct/nct01029/report1.sqlite
```

Task 5.7 Profiling Matrix Multiplication on the GPU

- 99.2% of total time is taken by `cudaMalloc` (highest overall) and `cudaDeviceReset` (highest by call). However only one call from `cudaMalloc` does take significant time.

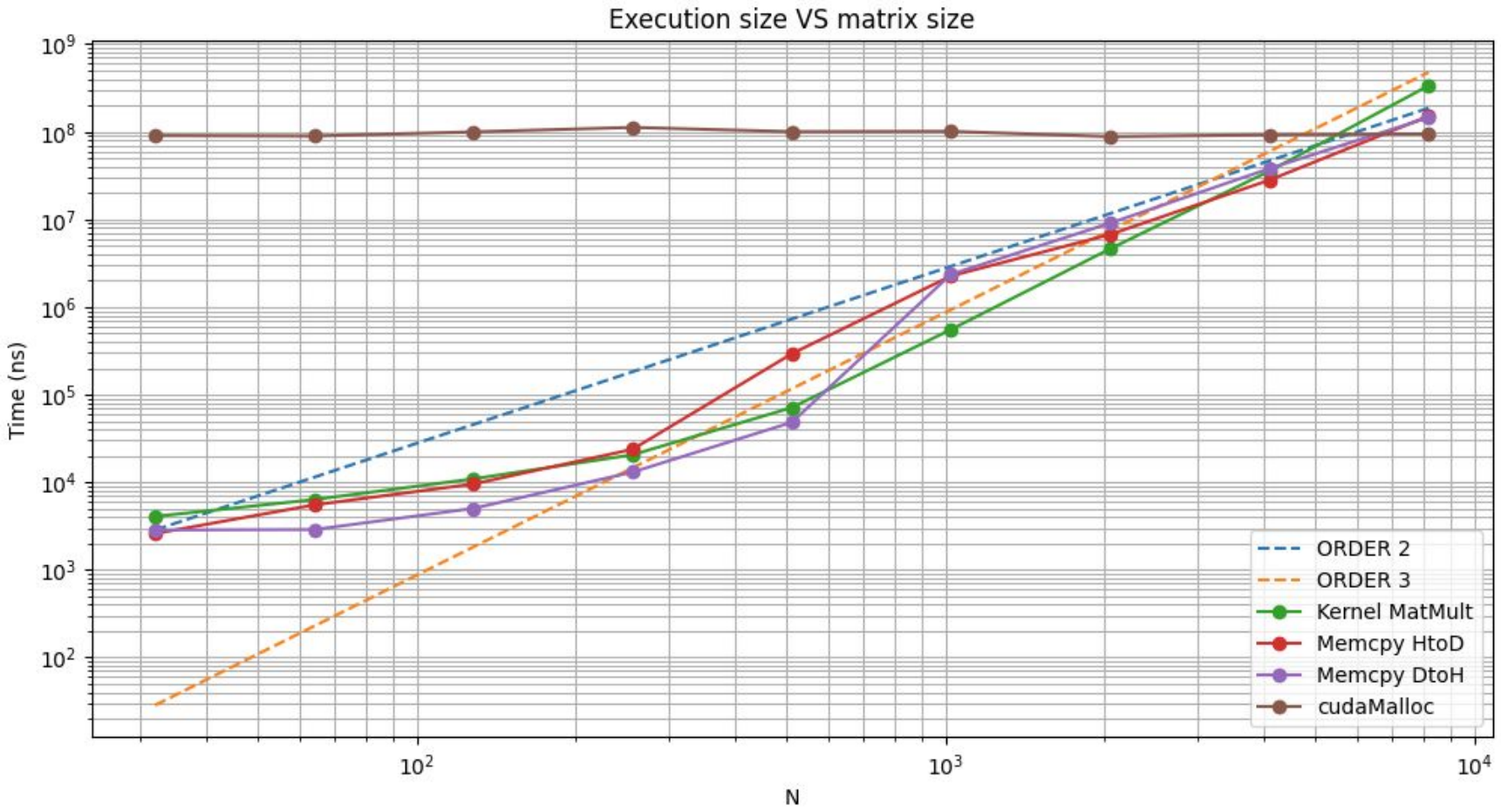
Task 5.7 Profiling Matrix Multiplication on the GPU

- With `N=1024`.

Time (%)	Total Time (ns)	Num Calls	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
63.1	92930857	3	30976952.3	73197.0	70047	92787613	53529602.4	cudaMalloc
30.0	44237149	1	44237149.0	44237149.0	44237149	44237149	0.0	cudaDeviceReset
3.5	5146273	1	5146273.0	5146273.0	5146273	5146273	0.0	cudaLaunchKernel
2.8	4174886	3	1391628.7	1027897.0	986217	2160772	666423.6	cudaMemcpy
0.4	558608	1	558608.0	558608.0	558608	558608	0.0	cudaDeviceSynchronize
0.2	330342	3	110114.0	84306.0	78575	167461	49746.6	cudaFree
0.0	3252	1	3252.0	3252.0	3252	3252	0.0	cuCtxSynchronize
0.0	2149	1	2149.0	2149.0	2149	2149	0.0	cuModuleGetLoadingMode

- `cudaLaunchKernel` and `cudaMemcpy` are more significant than before, both around the same cost (3.5% / 2.8 %). Their usage is likely scaline with N, judging that `N->8N` multiplied their total time by around 10.
- `cudaMalloc` is still predominant, but did not increased significantly with `N->8N` . Its cost may not overcome the rest if considering huge N.

Task 5.8 Compute-Bound vs Memory-Bound



Task 5.8 Compute-Bound vs Memory-Bound

- `cudaMalloc` is stable (order 0).
- `cudaMemcpy` are increasing with N by an order of 2, `O(N^2)`, which respect the matrix size.
- `matrixProduct` (`cudaDeviceSynchronize`) are increasing with N by an order of 3, `O(N^3)`, which respect the matrix product complexity.
- The workload become compute-bound at `N=8192` as `cudaMalloc` is overcome by the rest (`cudaMemcpy` is also behind `cudaDeviceSynchronize`).

Time (%)	Total Time (ns)	Num Calls	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
42.0	336051700	1	336051700.0	336051700.0	336051700	336051700	0.0	cudaDeviceSynchronize
37.1	296939460	3	98979820.0	75742827.0	75612667	145583966	40360426.8	cudaMemcpy
11.8	94694548	3	31564849.3	420653.0	399782	93874113	53961406.2	cudaMalloc

- The memory/computation load shifts since the matrix product is heavier in complexity than its memory size.