CINECA

Programming GPUs with CUDA: memory, grid size & error handling

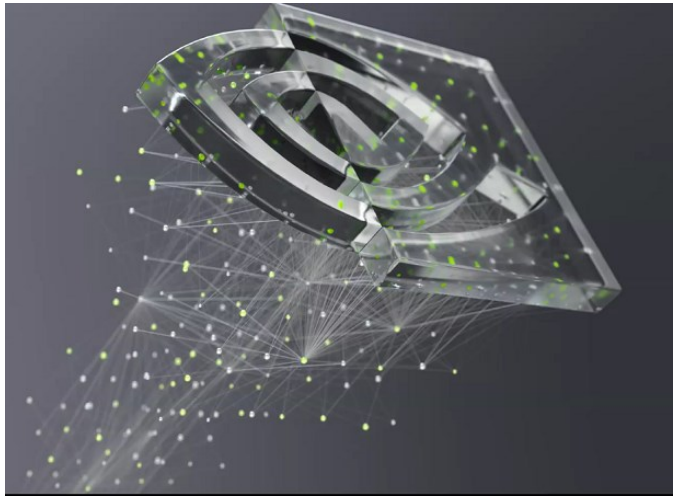
Lara Querciagrossa, Andrew Emerson, Nitin Shukla, Luca Ferraro, Sergio Orlandini

l.querciagrossa@cineca.it

July 12th, 2022

In this lecture…

- CUDA memory
- Grid size vs data size
- Error handling



CUDA memory

Unified Memory

- **Unified Memory** since CUDA 6+

```
int N = 10000;
size_t size = N * sizeof(int);

int *a;
a = (int *)malloc(size);

free(a);
```

CPU

```
int N = 10000;
size_t size = N * sizeof(int);

int *a;
cudaMallocManaged(&a, size);

cudaFree(a);
```

GPU

Unified Memory

- **Unified Memory** since CUDA 6+
- Allow to **allocate** and **free memory**

```
int N = 10000;
size_t size = N * sizeof(int);

int *a;
a = (int *)malloc(size);

free(a);
```

CPU

```
int N = 10000;
size_t size = N * sizeof(int);

int *a;
cudaMallocManaged(&a, size);

cudaFree(a);
```

GPU

Unified Memory

- **Unified Memory** since CUDA 6+
- Allow to **allocate** and **free memory**
- Allow to obtain a **pointer** that can be referenced in both host and device code

```
int N = 10000;
size_t size = N * sizeof(int);

int *a;
a = (int *)malloc(size);

free(a);
```

CPU

```
int N = 10000;
size_t size = N * sizeof(int);

int *a;
cudaMallocManaged(&a, size);

cudaFree(a);
```

GPU

Exercise: 6_double_element.cu

- This program allocates an array, initialize it with integers on the host, double each of its value in parallel on the GPU, confirm whether or not the doubling operation were successfully (on the host).
- Currently the program is not working: why?

1. Refactor the application so that array **a** is available for both host and device.
2. Free memory at **a** correctly.

Manual device memory management

- Worth using when it is known that data will **only be accessed by the device or the host.**

- Can also be used to allow the use of **non-default streams** (spoiler!) for overlapping data transfers with computational work.

- This approach shows clearly the **three-step process:**
  1. copy data from host to device,
  2. run work on device,
  3. copy back data from host to device.

Manual device memory management

1. Declare host and device arrays.
2. Allocate `device_a` directly on the GPU.
3. Allocate `host_a` on CPU, it is page-locked or pinned.
4. Call the initialize function on the host (no page faulting, since memory is already on the CPU).
5. Copy array `host_a` on the CPU to `device_a` on the GPU.
6. Launch kernel on the GPU using `device_a`.

```
int *host_a, *device_a;

cudaMalloc(&device_a, size);

cudaMallocHost(&host_a, size);

initializeOnHost(host_a, N);

cudaMemcpy(device_a, host_a,
size, cudaMemcpyHostToDevice);

kernel<<<blocks,
threads>>>(device_a, N);
```

Manual device memory management

7. Copy results back from GPU (`device_a`) to CPU memory into `host_a` array.

8. Verify results on `host_a` array on CPU.

9. Free memory both on GPU and pinned one on CPU.

```
cudaMemcpy(host_a, device_a,
size, cudaMemcpyDeviceToHost);

verifyOnHost(host_a, N);

cudaFree(device_a);
cudaFreeHost(host_a);
```

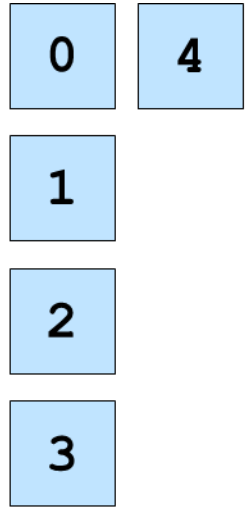
Exercise: 7_double_element_manual_memory_management.cu

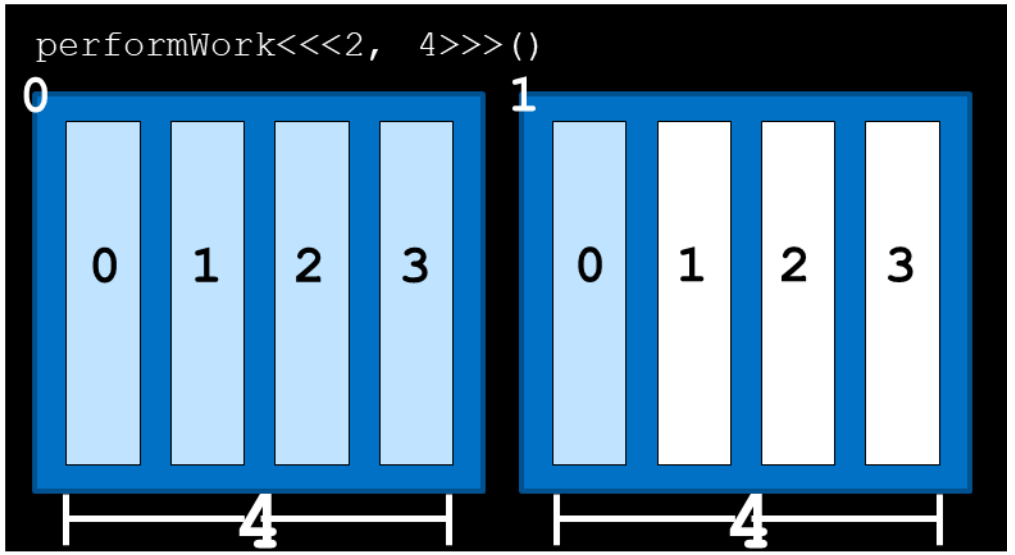
Start from the solution of the exercise 6 and replace the usage of unified memory with manual memory management.

Grid size vs data size

Grid size larger than data set

- What happens if there are more threads than work to be done? Or if we cannot express the exact number of threads with any execution configuration?





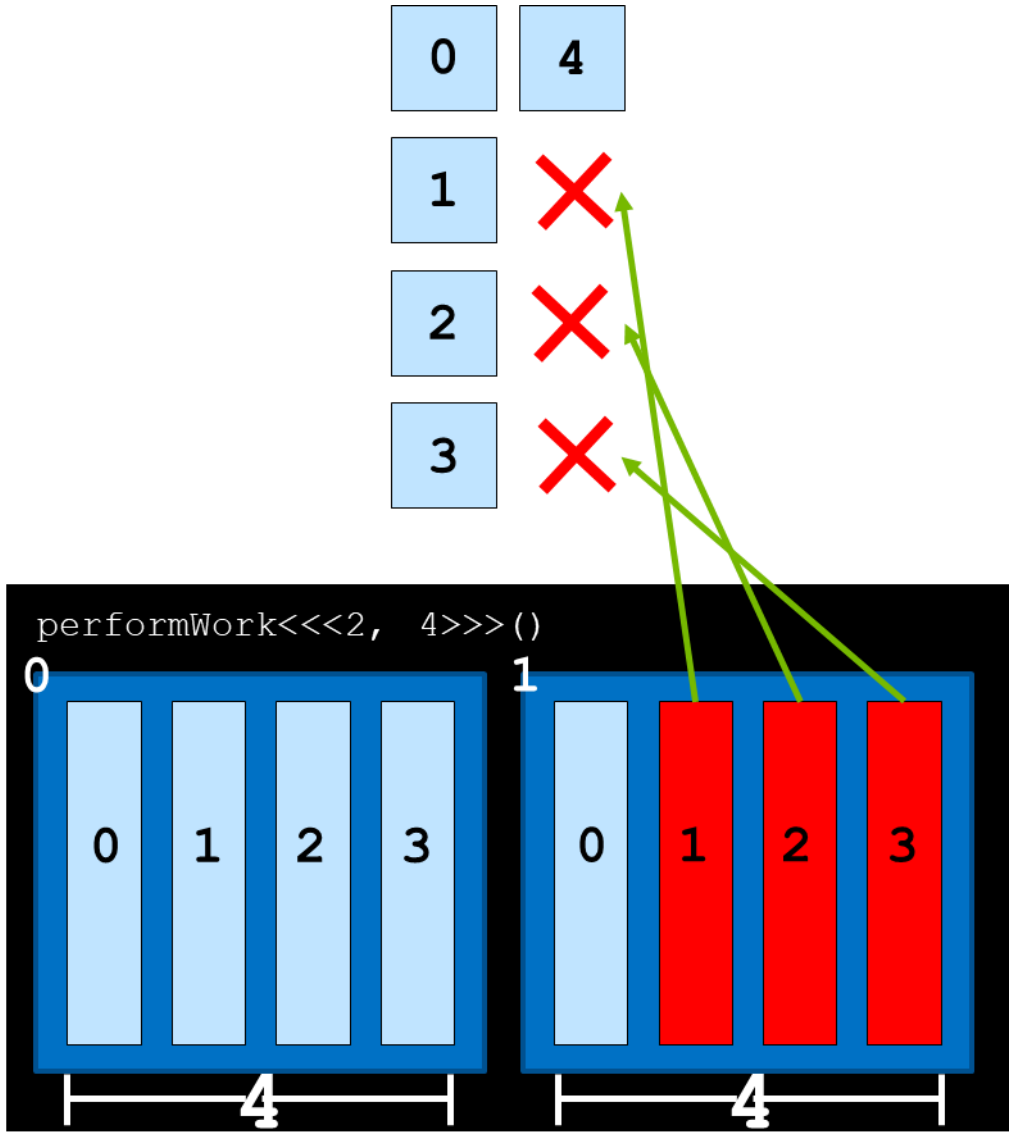
Grid size larger than data set

- What happens if there are more threads than work to be done? Or if we cannot express the exact number of threads with any execution configuration?

- Attempting to access non-existing elements can result in a runtime error.

- Kernel should work only if global index is **smaller** than **total number** of elements in the array

```
if (globalId < N)
```



Choosing the optimal grid size

- **Choose the optimal block size**: best performance for blocks that contain a number of threads that is a **multiple of 32**, due to GPU hardware traits.

- Example: we need to run 1000 parallel task with blocks containing 256 threads. How do we choose the optimal block size?

1. Write an *execution configuration* that creates *more threads* than necessary.
2. Pass a value as an *argument into the kernel* (`N`) that represents the *total size* of the data set to be processed/total threads needed to complete the work.
3. Calculate *the global index* and if it *does not exceed* `N` perform the kernel work.

Grid size larger than data set

- How do we ensure that there is a sufficient number of threads?

```
size_t threads_per_blocks = 256;

size_t number_of_blocks = (N  + threads_per_block - 1) /
threads_per_block;

kernel<<<number_of_blocks, threads_per_block>>>(N);
```

- This ensures that there are always at least as many threads as needed for N, and only 1 additional block's worth of threads extra.

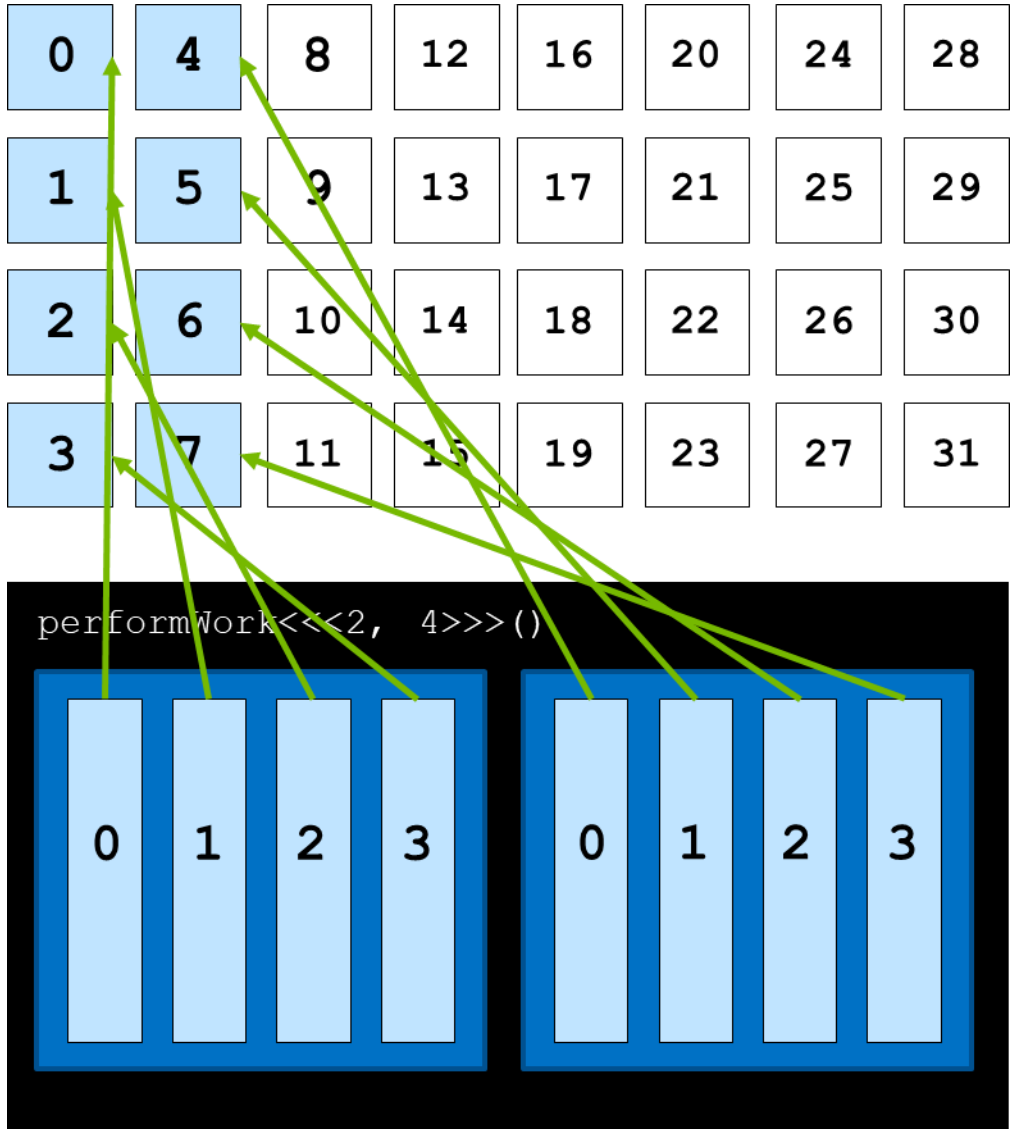
Exercise: 8_mismatch_loop.cu

The program allocates memory (with `cudaMallocManaged`) for a 1000 element array of integers, and then tries to initialize all the values of the array in parallel using a CUDA kernel.
The values of `N` and the number of `threads_per_block` should not be modified.

1. Assign a value to `number_of_blocks` that will make sure there are enough threads.
2. Update the `initializeElementsTo` kernel to make sure that it does not attempt to work on data elements that are out of range.

Data set larger than grid size: grid-stride loop

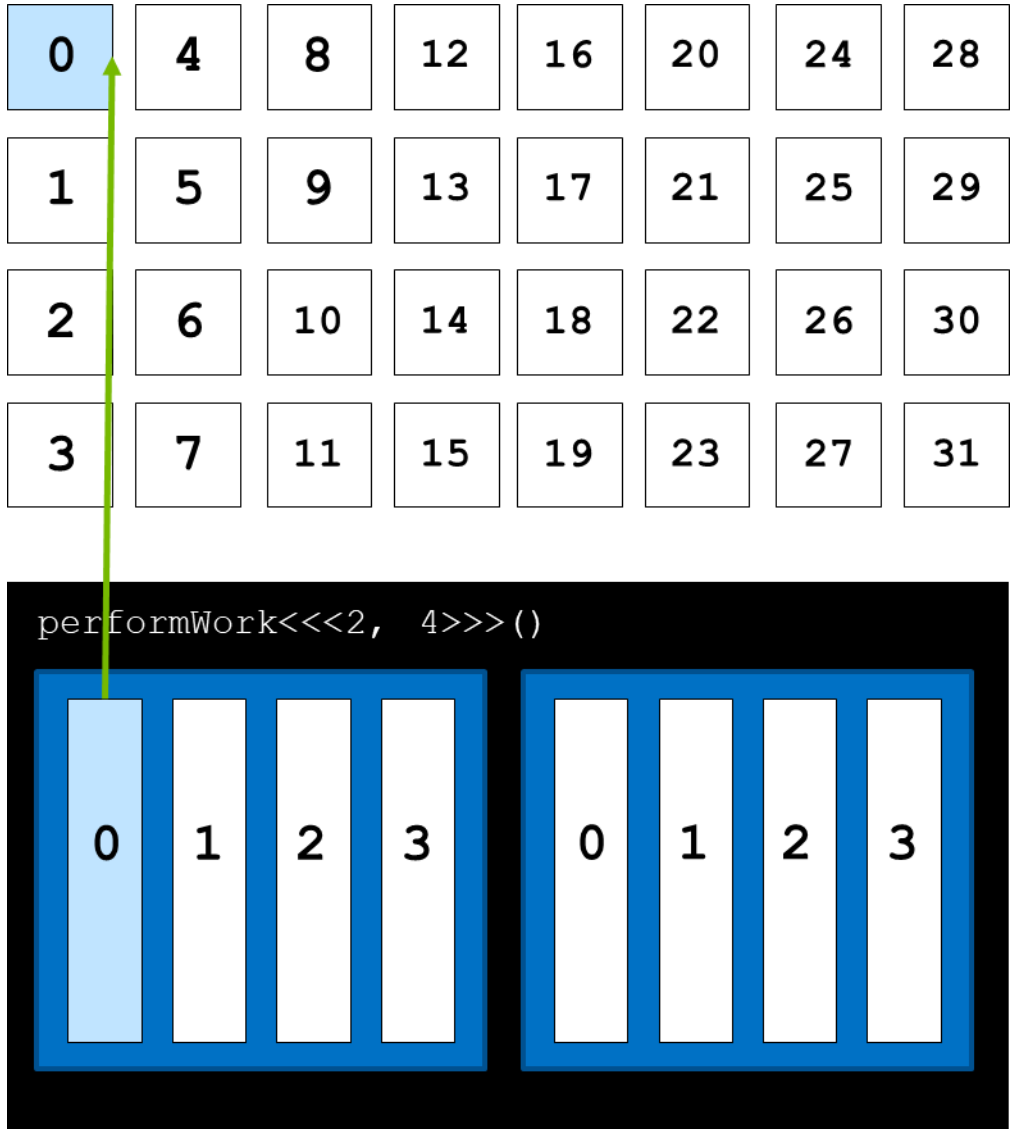
- In this scenario, each thread should work on more elements.
- Work can be assigned programmatically with a **grid-stride loop.**



Data set larger than grid size: grid-stride loop

- In this scenario, each threads should work on more elements.

- Work can be assigned programmatically with a **grid-stride loop:**
  - the first element to be assigned to a thread is calculated via the global index

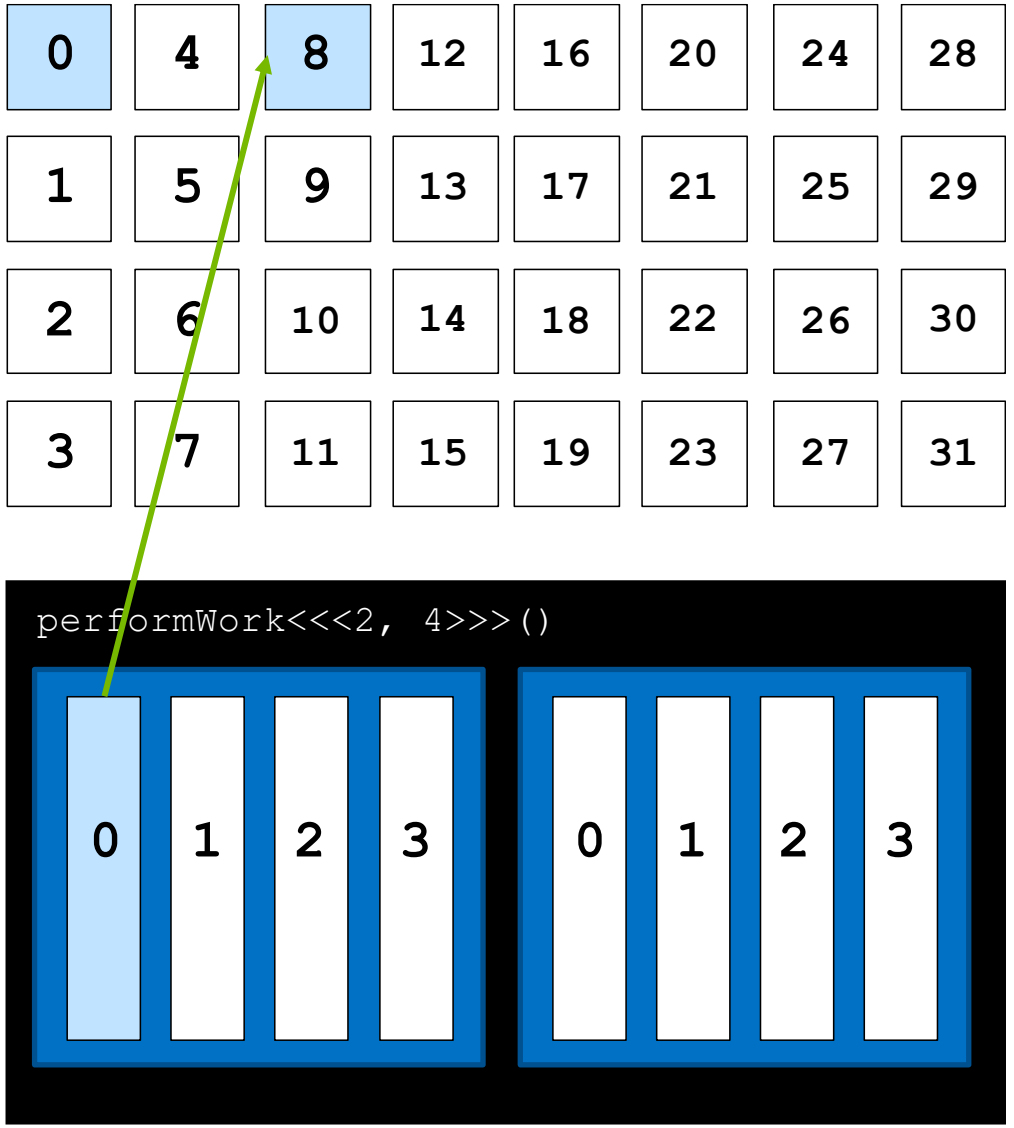


Data set larger than grid size: grid-stride loop

- In this scenario, each threads should work on more elements.

- Work can be assigned programmatically with a **grid-stride loop:**
  - the first element to be assigned to a thread is calculated via the global index,
  - the next one is obtained by summing the number of threads in the grid

```
stride = blockDim.x * gridDim.x
```

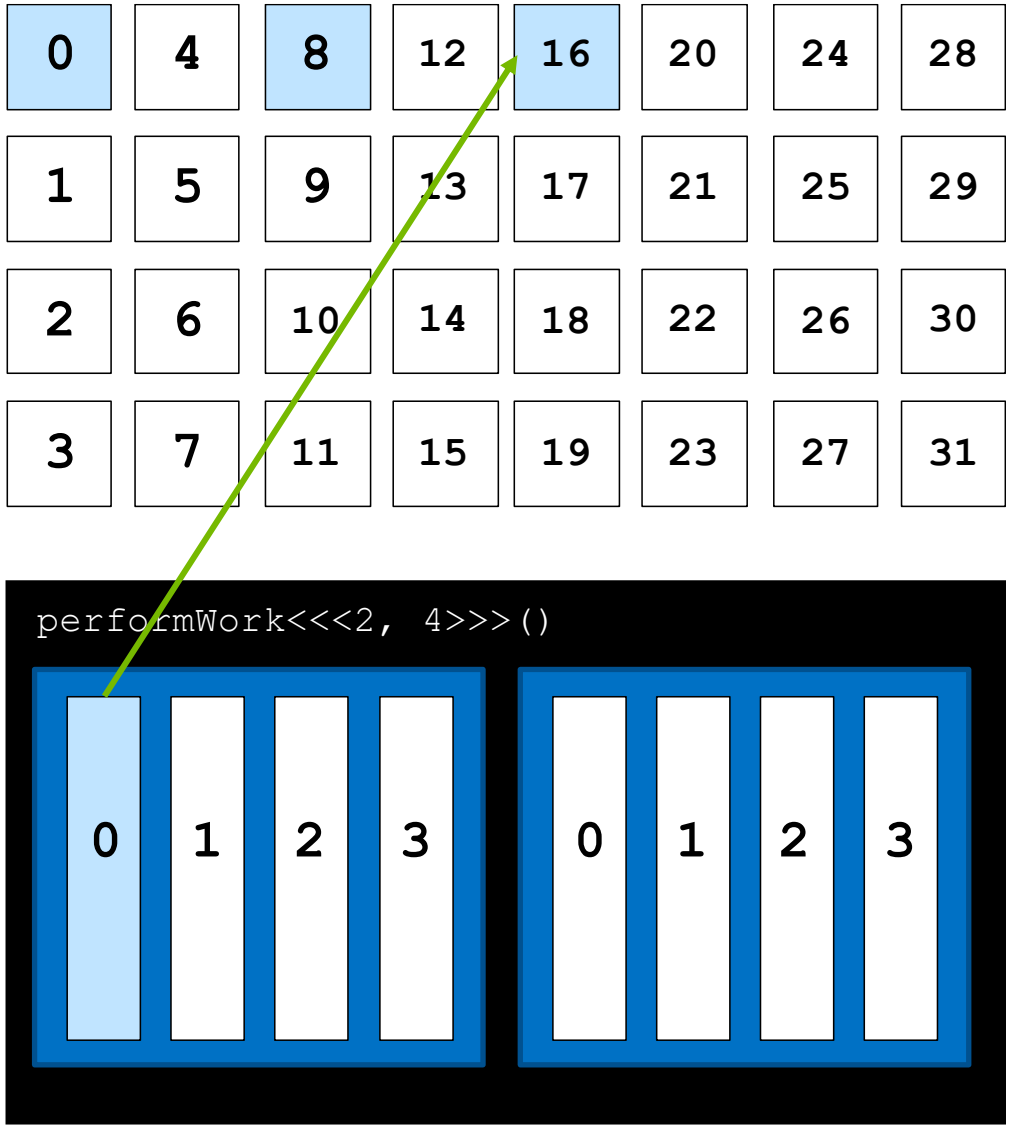


Data set larger than grid size: grid-stride loop

- In this scenario, each threads should work on more elements.

- Work can be assigned programmatically with a **grid-stride loop:**
  - the first element to be assigned to a thread is calculated via the global index,
  - the next one is obtained by summing the number of threads in the grid

```
stride = blockDim.x * gridDim.x
```

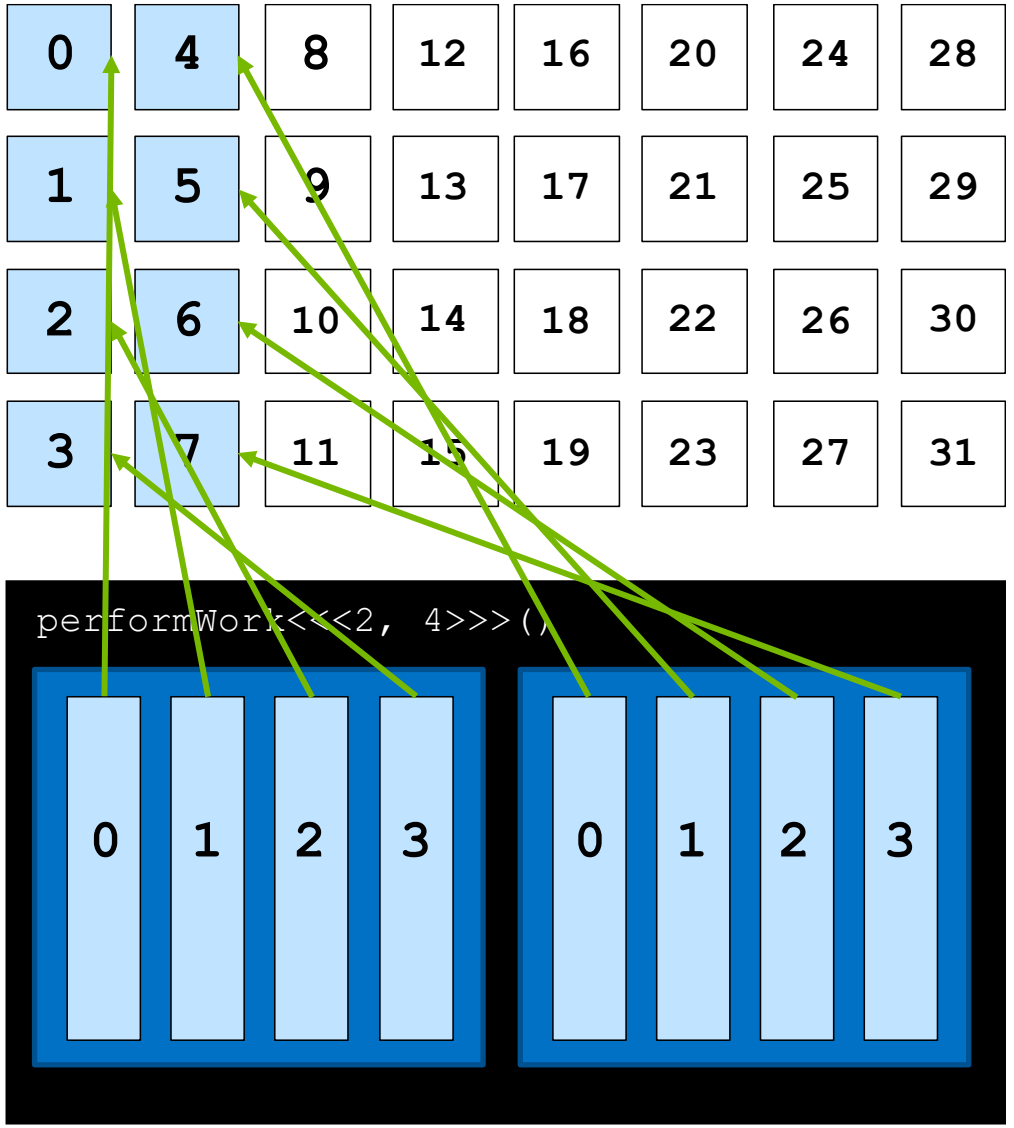


Data set larger than grid size: grid-stride loop

- In this scenario, each threads should work on more elements.

- Work can be assigned programmatically with a **grid-stride loop:**
  - the first element to be assigned to a thread is calculated via the global index,
  - the next one is obtained by summing the number of threads in the grid

```
stride = blockDim.x * gridDim.x
```

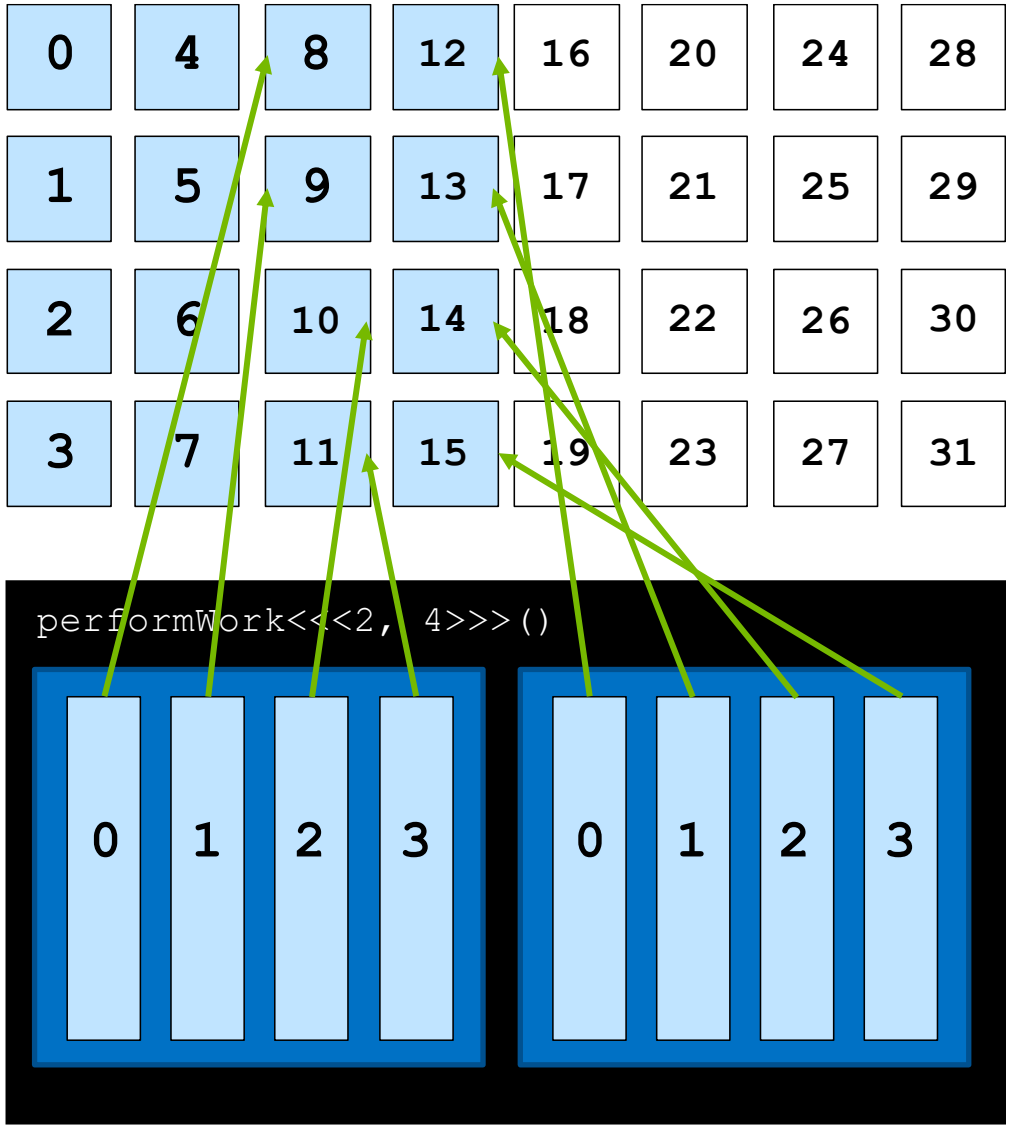


Data set larger than grid size: grid-stride loop

- In this scenario, each threads should work on more elements.

- Work can be assigned programmatically with a **grid-stride loop:**
  - the first element to be assigned to a thread is calculated via the global index,
  - the next one is obtained by summing the number of threads in the grid

```
stride = blockDim.x * gridDim.x
```

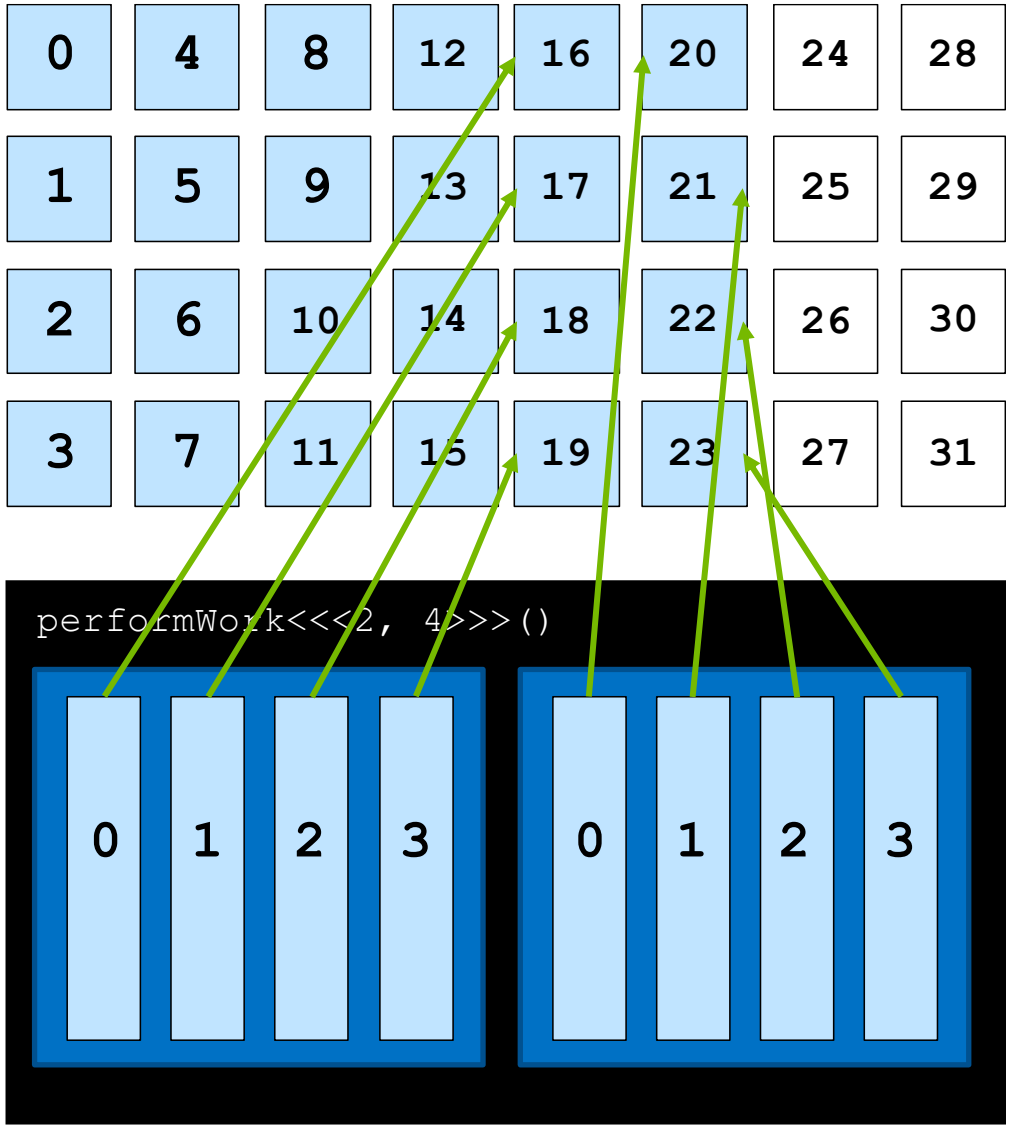


Data set larger than grid size: grid-stride loop

- In this scenario, each threads should work on more elements.

- Work can be assigned programmatically with a **grid-stride loop:**
  - the first element to be assigned to a thread is calculated via the global index,
  - the next one is obtained by summing the number of threads in the grid

```
stride = blockDim.x * gridDim.x
```

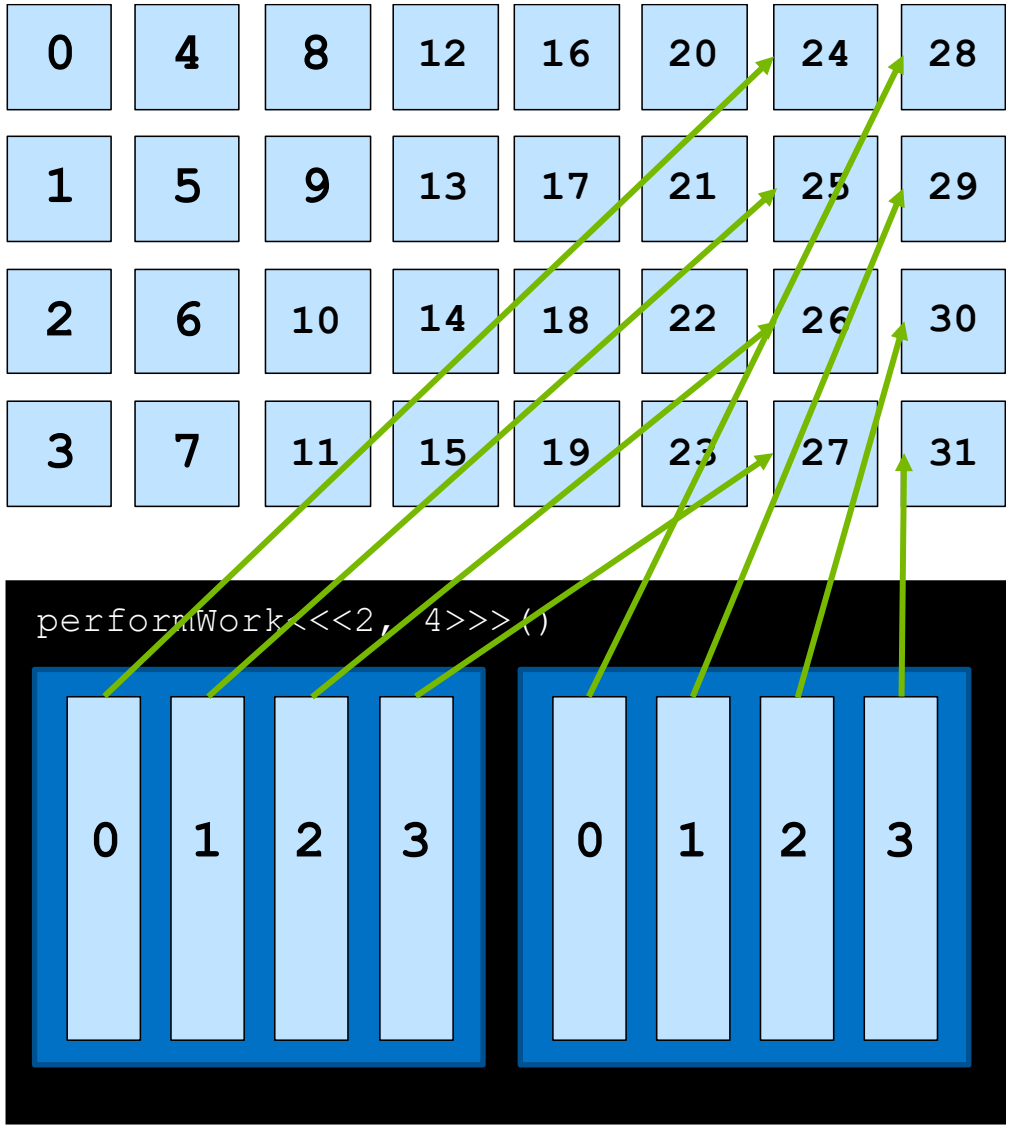


Data set larger than grid size: grid-stride loop

- In this scenario, each threads should work on more elements.

- Work can be assigned programmatically with a **grid-stride loop:**
  - the first element to be assigned to a thread is calculated via the global index,
  - the next one is obtained by summing the number of threads in the grid

```
stride = blockDim.x * gridDim.x
```



Data set larger than grid size: grid-stride loop

- Example: for an array with 1000 elements and a grid with 250 threads, each thread in the grid will need to be used 4 times.

- Operation inside the kernel will be executed in a grid-stride loop:

```
__global__ void kernel(int *a, int N) {
  int indexWithinTheGrid = threadIdx.x + blockIdx.x * blockDim.x;
  int gridStride = gridDim.x * blockDim.x;

  for (int i = indexWithinTheGrid; i < N; i += gridStride)
  {
    // do work on a[i];
  }
}
```

Exercise: 9_grid_stride_loop.cu

This code initialize on the CPU a vector of 10k elements. Then launch a kernel to double its value on GPU and finally perform a check to test values correctness on CPU.
Here the size of vector is bigger than the grid.

1. Refactor the kernel to use a grid-stride loop, so that each parallel thread works on more than one element of the array.
2. What is the size of the grid?

Error handling

Catching errors in CUDA

- Error handling in accelerated CUDA code is essential.
- Most CUDA functions return a value of type `cudaError_t`, which can be used to check if an error occurred while calling the function.
- If its value is **`cudaSuccess`** no error occurred.
- An error message can be printed with `cudaGetErrorString`.

```
cudaError_t err;
err = cudaMallocManaged(&a, N)

if (err != cudaSuccess) {
  printf("Error: %s\n", cudaGetErrorString(err));
}
```

Kernel launch errors

- **Kernel** are defined to return void, so they do not return a value of type `cudaError_t`.

- Example: launch configuration is erroneous.

- **`cudaGetLastError`** function return a value of type `cudaError_t` can be used to catch kernel errors.

```
someKernel<<<1, -1>>>();

cudaError_t err;
err = cudaGetLastError();
if (err != cudaSuccess) {
  printf("Error: %s\n", cudaGetErrorString(err));
}
```

Asynchronous errors

- To catch errors that occur in **asynchronous part** of the code (for example during the execution of an asynchronous kernel), check the **status returned by a subsequent synchronizing CUDA runtime API call**, such as `cudaDeviceSynchronize`.

```
cudaError_t asynchErr;
asynchErr = cudaDeviceSynchronize();
if (asynchErr != cudaSuccess) {
  printf("Error: %s\n", cudaGetErrorString(err));
}
```

Macro

- A macro that wraps CUDA function calls for checking errors could be useful.
- Can be wrapped around any function that returns a `cudaError_t`.

```
#include <stdio.h>
#include <assert.h>
inline cudaError_t checkCuda(cudaError_t result) {
  if (result != cudaSuccess) {
    fprintf(stderr, "CUDA Runtime Error: %s\n", cudaGetErrorString(result));
    assert(result == cudaSuccess);
  }
  return result;
}
int main() {
  […]
  checkCuda( cudaDeviceSynchronize() )
}
```

Exercise: 10_failing_double.cu

This code doubles the value of an array initialized on CPU, by launching a GPU kernel. Finally, a test on value correctness is performed on CPU.
This time the code has an error, and it prints that the elements of the array were not successfully doubled.

Refactor the application to handle CUDA errors: find the error(s) and fix it (them). Investigate both synchronous errors potentially created when calling CUDA functions, as well as asynchronous errors potentially created while a CUDA kernel is executing.

References

References

- Previous editions of this school at CINECA
- Oakridge National Laboratory's "Introduction to CUDA C++": https://www.olcf.ornl.gov/calendar/introduction-to-cuda-c/
- NVIDIA DL Institute Online Course: **main source of exercises**
- blogs.nvidia.com
- Wikipedia

THANK YOU!

Lara Querciagrossa
l.querciagrossa@cineca.it