PMC Lecture 19

Gianmaria Romano

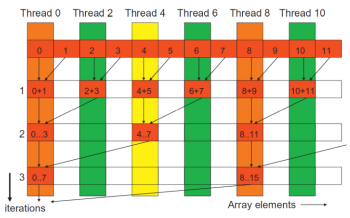
17 December 2025

Chapter 6

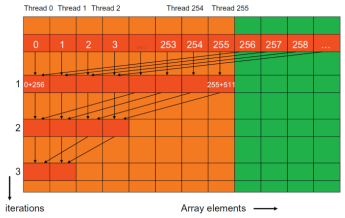
Compute Unified Device Architecture

6.1 Reduction operations in CUDA

While atomic operations can be used to implement reductions in CUDA, they often suffer from poor performance because, at each step, only half of the threads actually performs computations, resulting in a utilization drop.



For this reason, it is convenient to structure reductions so that threads within the same warp will always execute the same task, although only a part of the used threads may actually contribute to computations as the reduction progresses, therefore improving utilization.



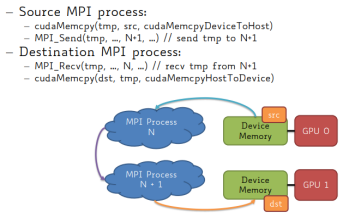
Example: The following code provides an efficient implementation of a reduction operation on an array:

```cpp
#include <stdio.h>
#include <cuda.h>

__shared__ float partialSum[SIZE];
partialSum[threadIdx.x] = X[blockIdx.x * blockDim.x + threadIdx.x];

unsigned int t = threadIdx.x;
for (unsigned int strid = blockDim.x / 2; stride >= 1; stride = stride>>1) {
    // N.B.: The loop divides the stride by 2 at each iteration.
    __syncthreads();
    i (t < stride) {
        partialSum[t] += partialSum[t + stride];
    }
}
```

Note that, if the number of elements in the array is larger than the number of threads within a block, it is possible to introduce some additional kernels and work on partial results that will be merged at the end.

6.2 GPU data transfers

When dealing with distributed GPU applications, data transfers between GPUs are often handled using message passing interface.
In particular, if the message passing interface is not GPU-aware, data must be explicitly copied from the sender device to the host memory before making the desired call.
On the receiving side, data is transferred from the host to the receiver device.



On the other hand, if the message passing interface is GPU aware, it can directly access device memory, allowing to handle communications without having to involve the host memory during data transfers.