UNIVERSITÀ di VERONA

Dipartimento di INFORMATICA

PARCO Parallel Computing Lab

NVIDIA CUDA

Part III

Laurea magistrale in Ingegneria e Scienze Informatiche
Laurea Magistrale in Medical Bioinformatics

Nicola Bombieri – Federico Busato

Agenda

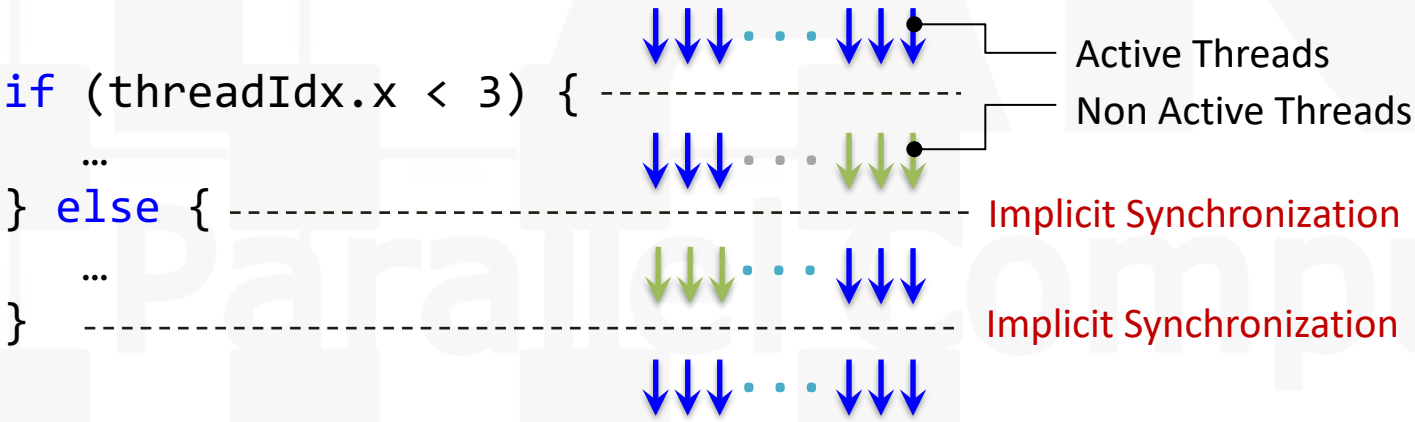
- Branch Divergence
- Example and Exercises: Vector Reduction
- References

CUDA Branch Divergence

PARTIAL DEFINITION:

Divergent branches: Threads within a single warp take different paths

- Hardware serializes the different execution paths
- Branch divergence induce latency



Avoiding Branch Divergence: The branch granularity must be a whole multiple of warp size -> all threads in a given warp follow the same control path

- `E.g. if (threadsIdx.x < 32 || threadsIdx.x >= 96)`
- Not always possible

Vector Reduction (I)

- Let $\otimes$ be a binary associative operator. Given a set of N elements:

  $$\left[x_0, x_1, \ldots, \mathrm{x}_N\right]$$

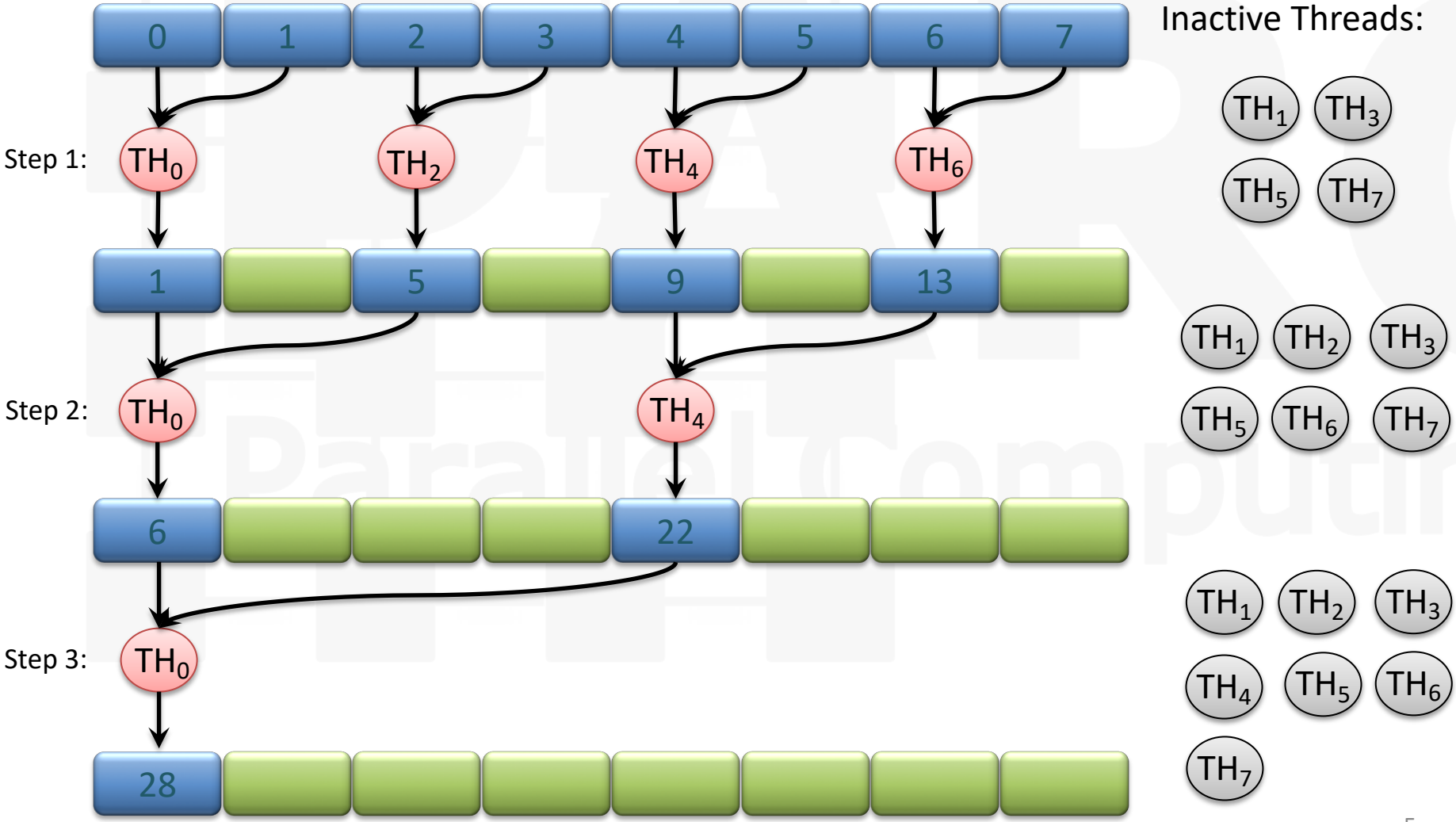
  The reduction operator returns

  $$x_0 \otimes x_1 \otimes \ldots \otimes \mathrm{x}_N$$

- The simple sequential reduction is not parallel at all: there's a sequential dependency on the accumulator variable that requires this reduction be done in a particular order, from front to back of the input array.
- Parallel reduction is a common building block for many parallel algorithms
- Typical parallel implementation:
  - Each thread adds two values
  - Takes $\log_2(n)$ steps and requires N/2 threads (at the beginning) for N elements
  - The final result will be stored in the first element

Vector Reduction (II)

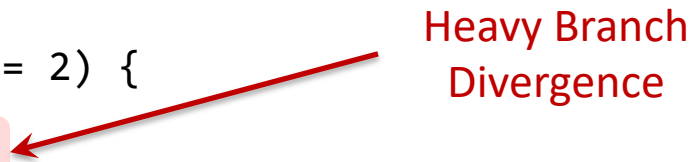
Block Reduction:



Vector Reduction (III)

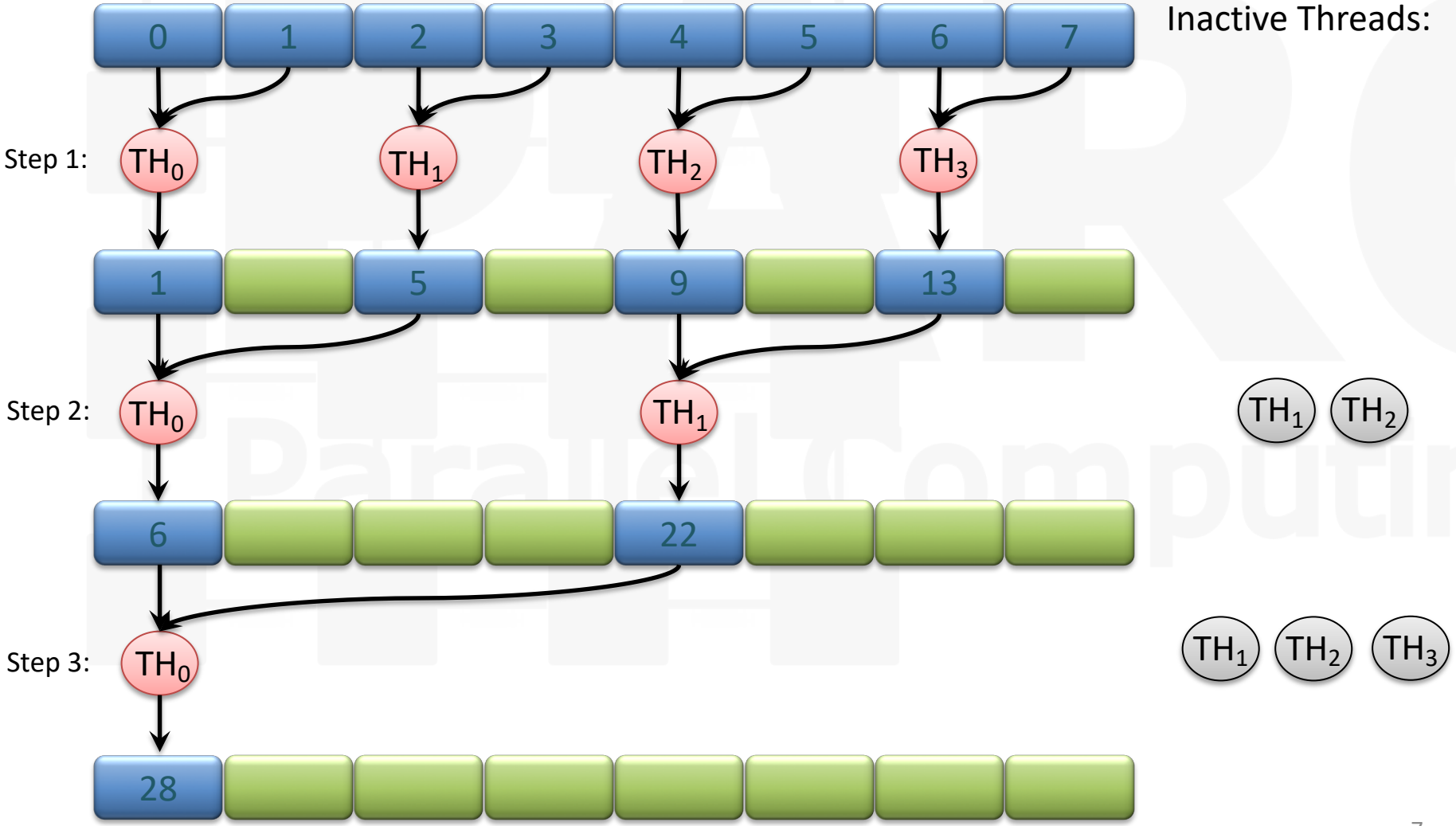
First Version:

```
__shared__ int SMem[1024];
int GlobalIndex = blockIdx.x * blockDim.x + threadIdx.x;
SMem[threadIdx.x] = VectorIN[GlobalIndex];
__syncthreads();
for (int i = 1; i < blockDim.x; i *= 2) {
    if (threadIdx.x % (i * 2) == 0)
        SMem[threadIdx.x] += SMem[threadIdx.x + i];
    __syncthreads();
}
if (threadIdx.x == 0)
    VectorOUT[blockIdx.x] = SMem[0];
```



Vector Reduction (IV)

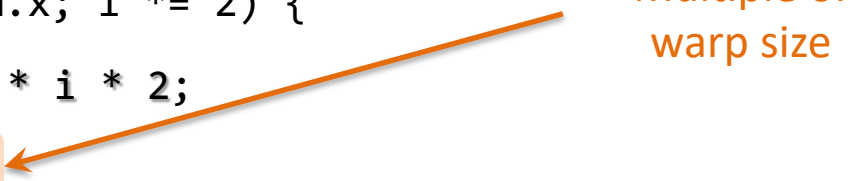
Block Reduction…a better pattern (N/2 threads):



Vector Reduction (V)

Second Version:

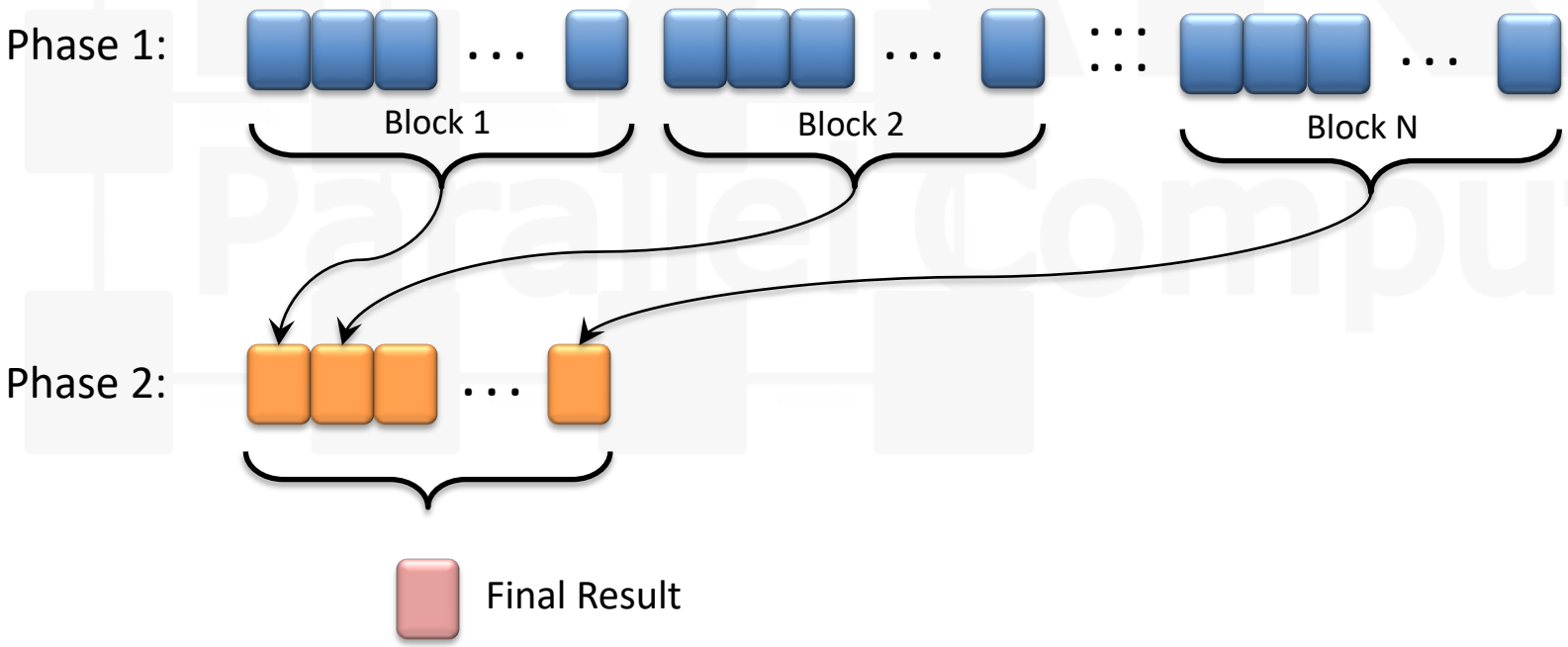
```
__shared__ int SMem[1024];
int GlobalIndex = blockIdx.x * blockDim.x + threadIdx.x;
SMem[threadIdx.x] = VectorIN[GlobalIndex];
__syncthreads();
for (int i = 1; i < blockDim.x; i *= 2) {
    int index = threadIdx.x * i * 2;
    if (index < blockDim.x)
        SMem[index] += SMem[index + i];
    __syncthreads();
}
if (threadIdx.x == 0)
    VectorOUT[blockIdx.x] = SMem[0];
```



Global Vector Reduction

Two-Phase Reduction:

1) Each Block computes the reduction on a subset of input using the shared memory and stores the result in global memory
2) One block computes the final reduction on partial block result of the first phase



Exercices

- Implement the two versions of Vector Reduction.
- Implement the Vector Reduction on large arrays computing the whole result.
- Compare the performance among different kernel configurations

References

Optimizing Parallel Reduction on CUDA (2007)

http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf

Optimizing Parallel Reduction in CUDA

Mark Harris
NVIDIA Developer Technology

References

- CUDA C Programming Guide

  http://docs.nvidia.com/cuda/cuda-c-programming-guide/

- CUDA Runtime API

  http://docs.nvidia.com/cuda/cuda-runtime-api/

  http://developer.download.nvidia.com/compute/cuda/4_1/rel/toolkit/docs/online/modules.html

  (Memory Management)

- CUDA Math API

  http://docs.nvidia.com/cuda/cuda-math-api/