Programmazione di Sistemi ~~Embedded e~~ Multicore

Teacher: Daniele De Sensi

Exam Rules & Assignments

Exam Rules (pt. 1)

- Teams of at most 2 students. You will get a grade for the project. The grade will be assigned to the team, not to the student (i.e., it does not matter who does what, both students need to know all the details of the project).
- There are three different projects you can choose from (you can find details in the project/ folder of the Github repo)
- You can propose a new project (send me an email)
- You must have three separate implementations (one for CUDA, one for Pthreads/OpenMP, one for MPI), **AND**
- You must have at least one implementation combining MPI+CUDA or MPI+Pthreads or MPI+OpenMP
  - (We will see how to do MPI+CUDA tomorrow)
- The parallel implementation **MUST** scale. If it does not, you must motivate why
- Verify the correctness of the parallel versions by comparing the results with those obtained by the sequential version (fix the seed if there is randomness involved)

Exam Rules (pt. 2)

What systems shall we use for the experiments?

- My suggestion is to use your laptop/workstation at least in the preliminary stages while debugging/implementing the code.
  - If you have an AMD rather than an NVIDIA GPU, you can still write CUDA code, and then convert it to HIP code (the AMD's equivalent of CUDA), using their **hipify** tool
  - If you do not have a laptop/workstation with a GPU, you can use Google's Colab. You can find instructions on how to run CUDA on Colab here: https://medium.com/@zubair09/running-cuda-on-google-colab-d8992b12f767 (just tried and it works).

- After the implementation/debug phase, I suggest you using the cluster to do performance tuning and to gather the data for the report. You can find a Google form link in one of the previous lectures' slides for requesting access to the cluster. If I do not create you an account after 3/4 days from when you filled the form, please write me an email.
  - I'll update the instructions on how to use the cluster for MPI+CUDA

Exam Rules (pt. 3)

- I will publish a Google form by the end of the week. You can pick a date out of a set of dates for the oral exam.
- One week before the oral exam, you must send me a zip file with the code, plus a PDF report. The submission of the code+report will be done through another Google form. If this is delayed, you must book again in a later slot.

Exam Rules (pt. 4)

- The PDF report must be no longer than 5 pages, and must contain the following:
  - Description of the implemented solution. Discuss the main implementation choice, as well as alternative choices that you tried and did not work and/or did not perform well. The trial and error process is important and shows maturity.
  - Experimental evaluation. Report weak/strong scaling, and use all the good practices we discussed (repeat the experiment multiple times, report variability, etc...)
  - Discussion on the limitations and ideas on how to address them (e.g., «we might need a system with a faster network», «we were bottlenecked by the GPU memory bandwidth», etc...)
- If you did not pick one of the three «default» projects I assigned, you must have a section describing the problem you want to solve (you can use one extra page)

Exam Rules (pt. 5)

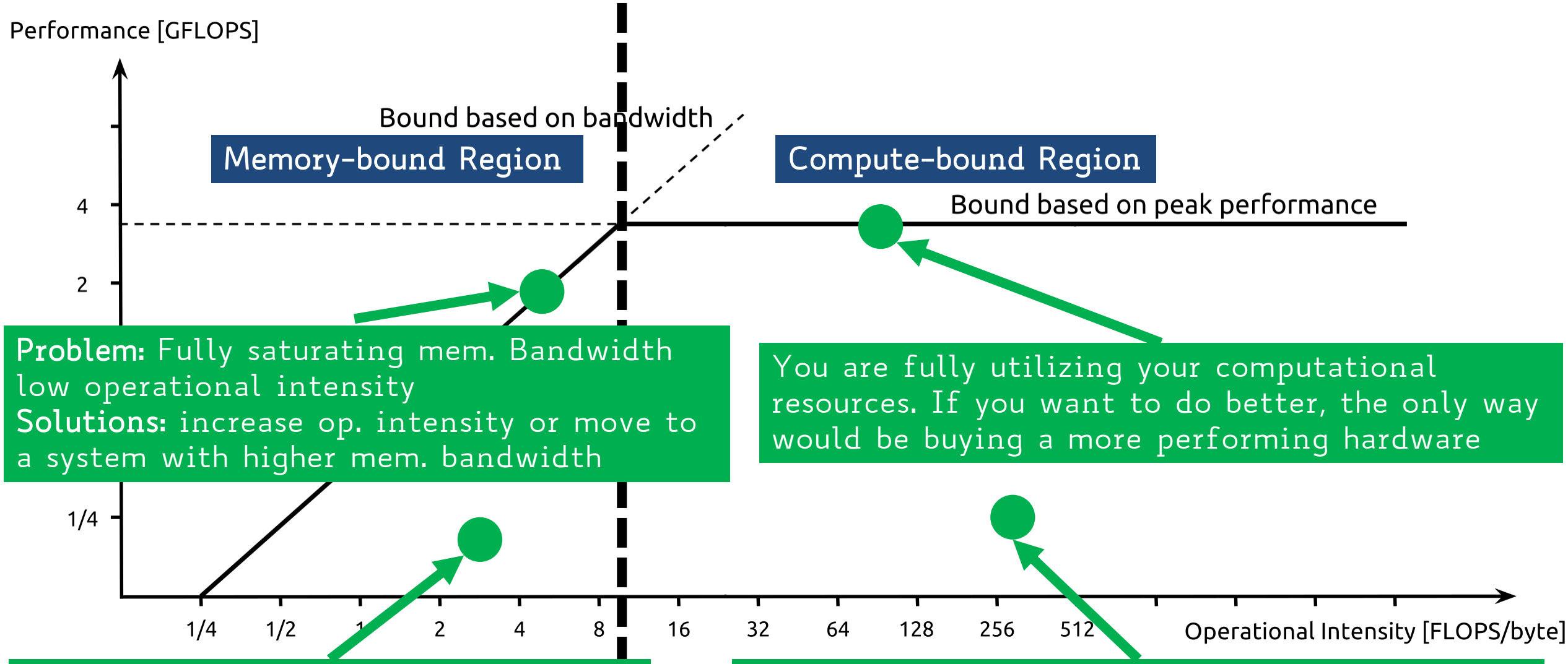
The oral exam is divided in two parts

- First, you will present your project. You can prepare a PowerPoint presentation if you prefer (is not mandatory) – max 10 minutes per group. After your presentation, I'll ask you questions about the project, about your design choices, whether you did or did not try a specific technique, etc... Grading approximatively as:
  - **Correctness:** (30%) The implementation must produce correct results for a range of test cases.
  - **Performance:** (30%) Speed-up and efficiency on parallel architectures.
  - **Code Quality:** (10%) Readability, comments, and adherence to coding standards.
  - **Report and Presentation:** (30%) Clarity, completeness, and quality of analysis.
- Then, I will move to questions not strictly related to the project, covering the topics we have seen during the class

Questions?

Recap

Roofline model



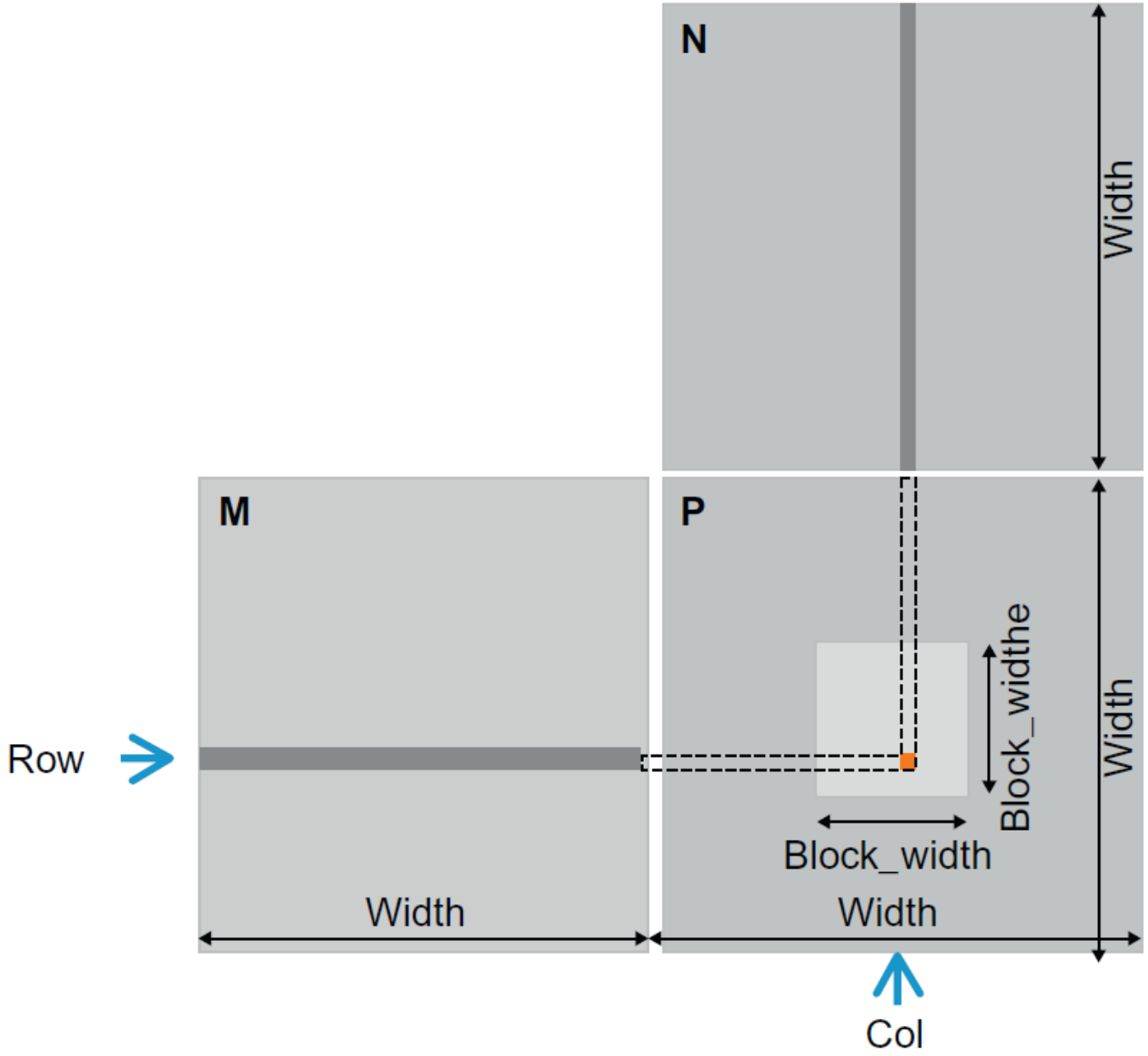
Problem: Fully saturating mem. Bandwidth low operational intensity
Solutions: increase op. intensity or move to a system with higher mem. bandwidth

You are fully utilizing your computational resources. If you want to do better, the only way would be buying a more performing hardware

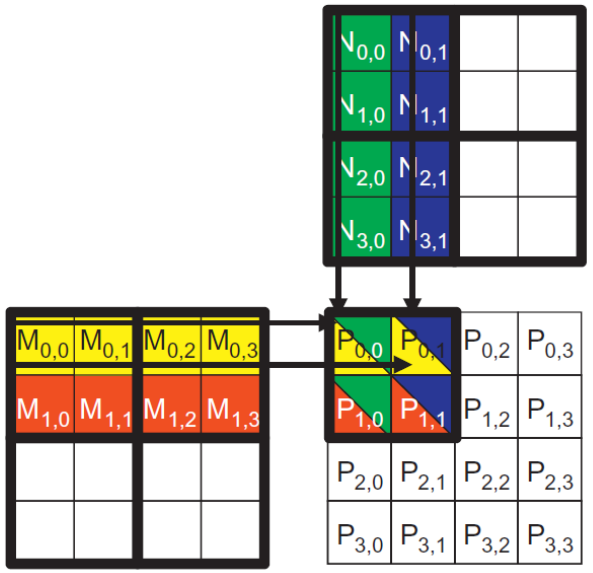
Problem: Suboptimal memory usage
Solutions: Optimize access patterns, reduce data movement, exploit locality

Problem: Compute inefficiencies (too many branches, load imbalance, etc...)
Solutions: Optimize compute, reduce overheads

Matrix Multiplication



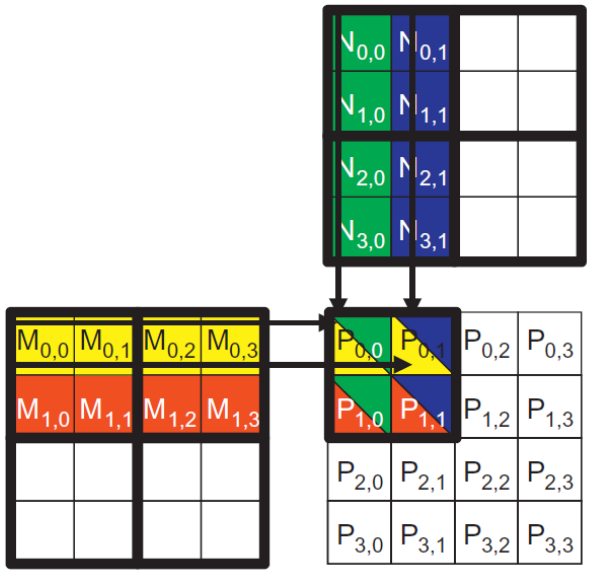
Tiling



	Phase 1		
$thread_{0,0}$	**$M_{0,0}$** ↓ $Mds_{0,0}$	**$N_{0,0}$** ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$
$thread_{0,1}$	**$M_{0,1}$** ↓ $Mds_{0,1}$	**$N_{0,1}$** ↓ $Nds_{1,0}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$
$thread_{1,0}$	**$M_{1,0}$** ↓ $Mds_{1,0}$	**$N_{1,0}$** ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$
$thread_{1,1}$	**$M_{1,1}$** ↓ $Mds_{1,1}$	**$N_{1,1}$** ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$

time

Tiling



	Phase 1			Phase 2		
$thread_{0,0}$	**$M_{0,0}$** ↓ $Mds_{0,0}$	**$N_{0,0}$** ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$	**$M_{0,2}$** ↓ $Mds_{0,0}$	**$N_{2,0}$** ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$
$thread_{0,1}$	**$M_{0,1}$** ↓ $Mds_{0,1}$	**$N_{0,1}$** ↓ $Nds_{1,0}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$	**$M_{0,3}$** ↓ $Mds_{0,1}$	**$N_{2,1}$** ↓ $Nds_{0,1}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$
$thread_{1,0}$	**$M_{1,0}$** ↓ $Mds_{1,0}$	**$N_{1,0}$** ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$	**$M_{1,2}$** ↓ $Mds_{1,0}$	**$N_{3,0}$** ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$
$thread_{1,1}$	**$M_{1,1}$** ↓ $Mds_{1,1}$	**$N_{1,1}$** ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$	**$M_{1,3}$** ↓ $Mds_{1,1}$	**$N_{3,1}$** ↓ $Nds_{1,1}$	$PValue_{1,1}$+= $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$

time →

- **Without tiling:** 4 threads, each loads one row (4 loads) and one column (4 loads) = 4x(4+4) = 32 accesses to global memory
- **With tiling:** 4 threads: each loads 4 elements from global memory = 16 accesses to global memory **(2x reduction in global accesses)**

Cluster Hands-On

Example: Jacobi 2D

Questions?

Jacobi Solution

diffCPU

```
double diffCPU(const double *phi, const double *phiPrev, int N)
{
    int i;
    double sum = 0;
    double diffsum = 0;

    for (i = 0; i < N*N; i++) {
        diffsum += (phi[i] - phiPrev[i]) * (phi[i] - phiPrev[i]);
        sum += phi[i] * phi[i];
    }

    return sqrt(diffsum/sum);
}
```

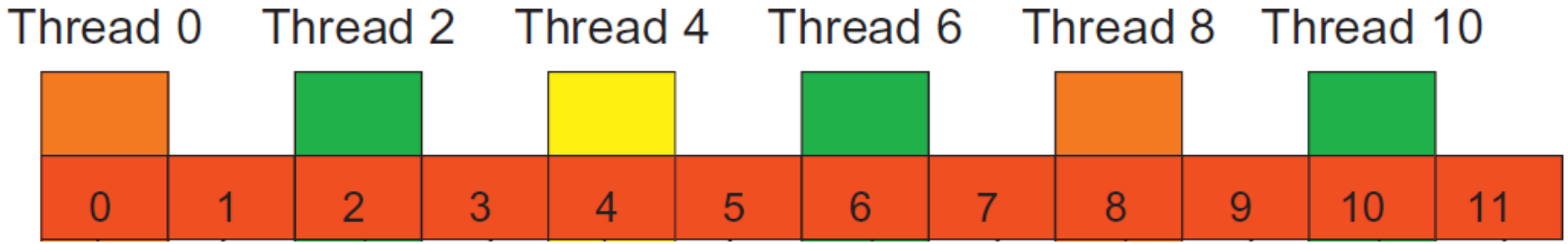
diffGPU

```
__global__
void diffGPU(const double *phi, const double *phiPrev, int N, double* sum, double* diffsum)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int index = i + j*N;
    atomicAdd(diffsum, (phi[index] - phiPrev[index]) * (phi[index] - phiPrev[index]));
    atomicAdd(sum, phi[index] * phi[index]);
}
```

```
if (iterations % 100 == 0) {
// Reinitialize sum_d and diffsum_d to 0
    sum_h = 0; diffsum_h = 0;
    CUDA_CHECK( cudaMemcpy(sum_d, &sum_h, sizeof(double), cudaMemcpyHostToDevice) );
    CUDA_CHECK( cudaMemcpy(diffsum_d, &diffsum_h, sizeof(double), cudaMemcpyHostToDevice) );

    diffGPU<<<dimGrid, dimBlock>>>(phiPrev_d, phi_d, N, sum_d, diffsum_d);
    CUDA_CHECK( cudaMemcpy(&sum_h, sum_d, sizeof(double), cudaMemcpyDeviceToHost) );
    CUDA_CHECK( cudaMemcpy(&diffsum_h, diffsum_d, sizeof(double), cudaMemcpyDeviceToHost) );
    diff = sqrt(diffsum_h/sum_h);
    CHECK_ERROR_MSG("Difference computation");
    printf("%d %g\n", iterations, diff);
}
```

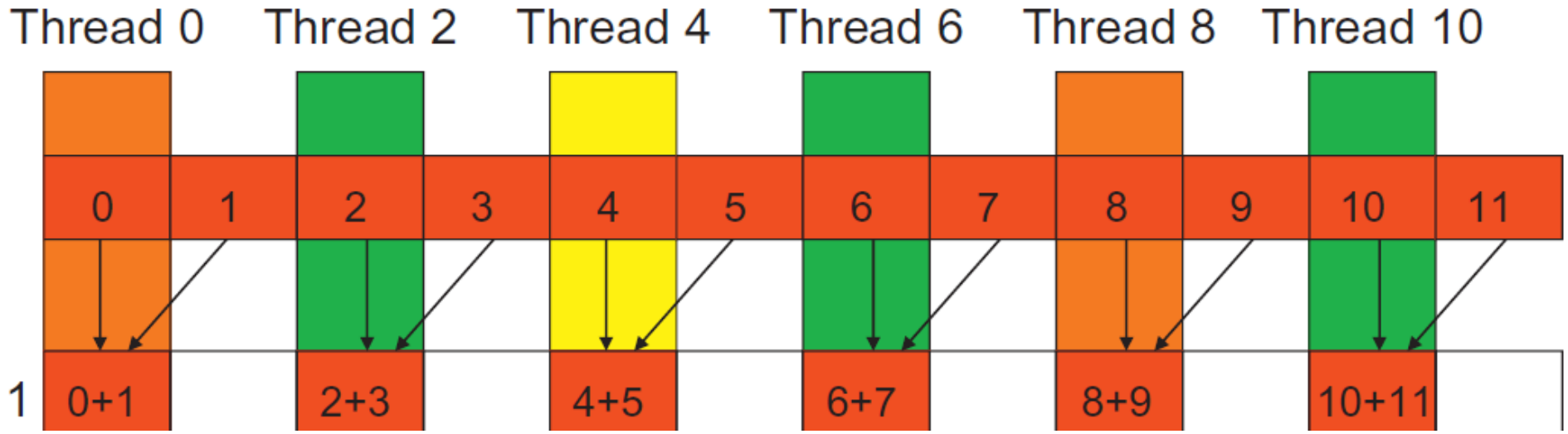
Better way of doing a reduce?



iterations

Array elements

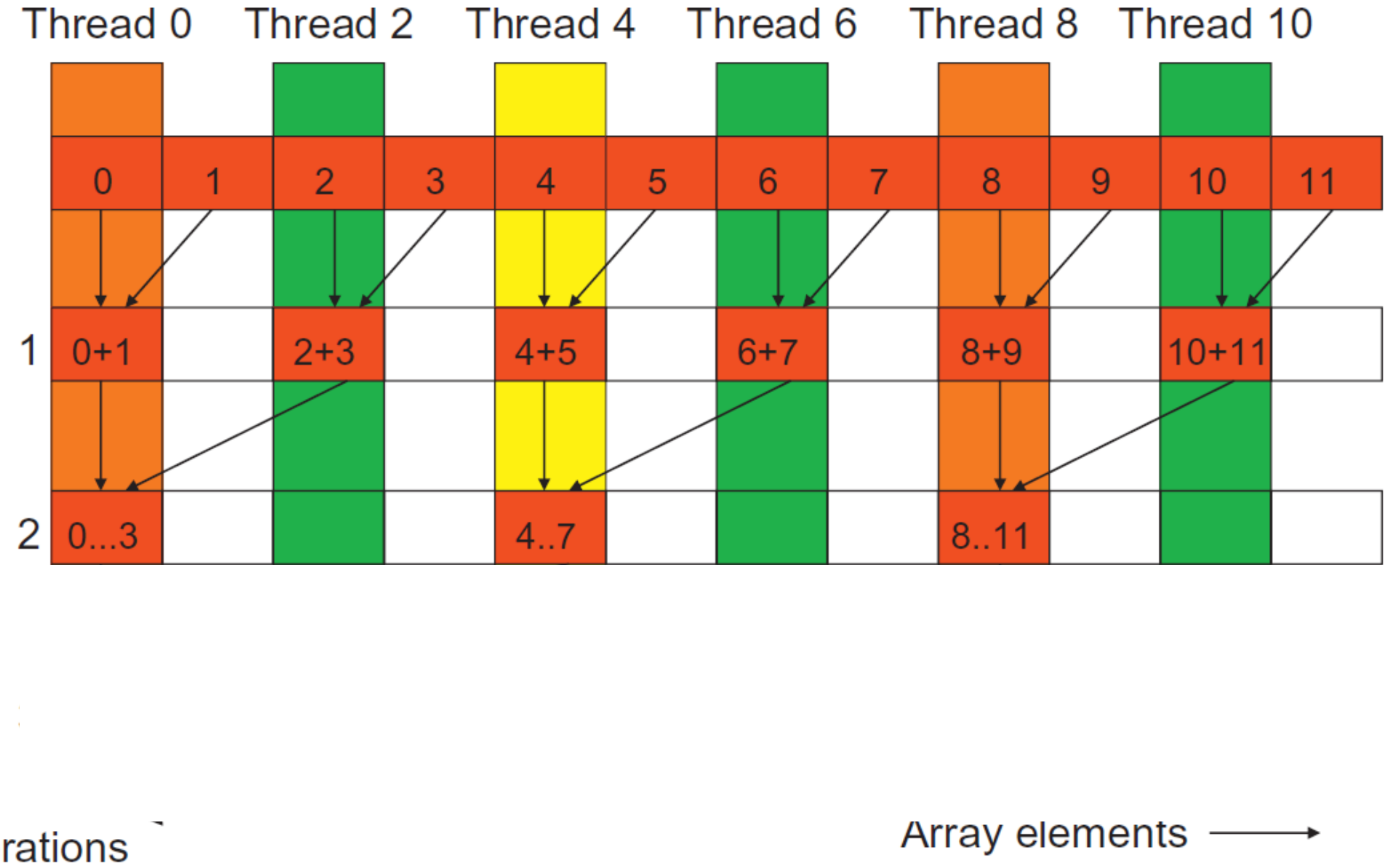
Better way of doing a reduce?



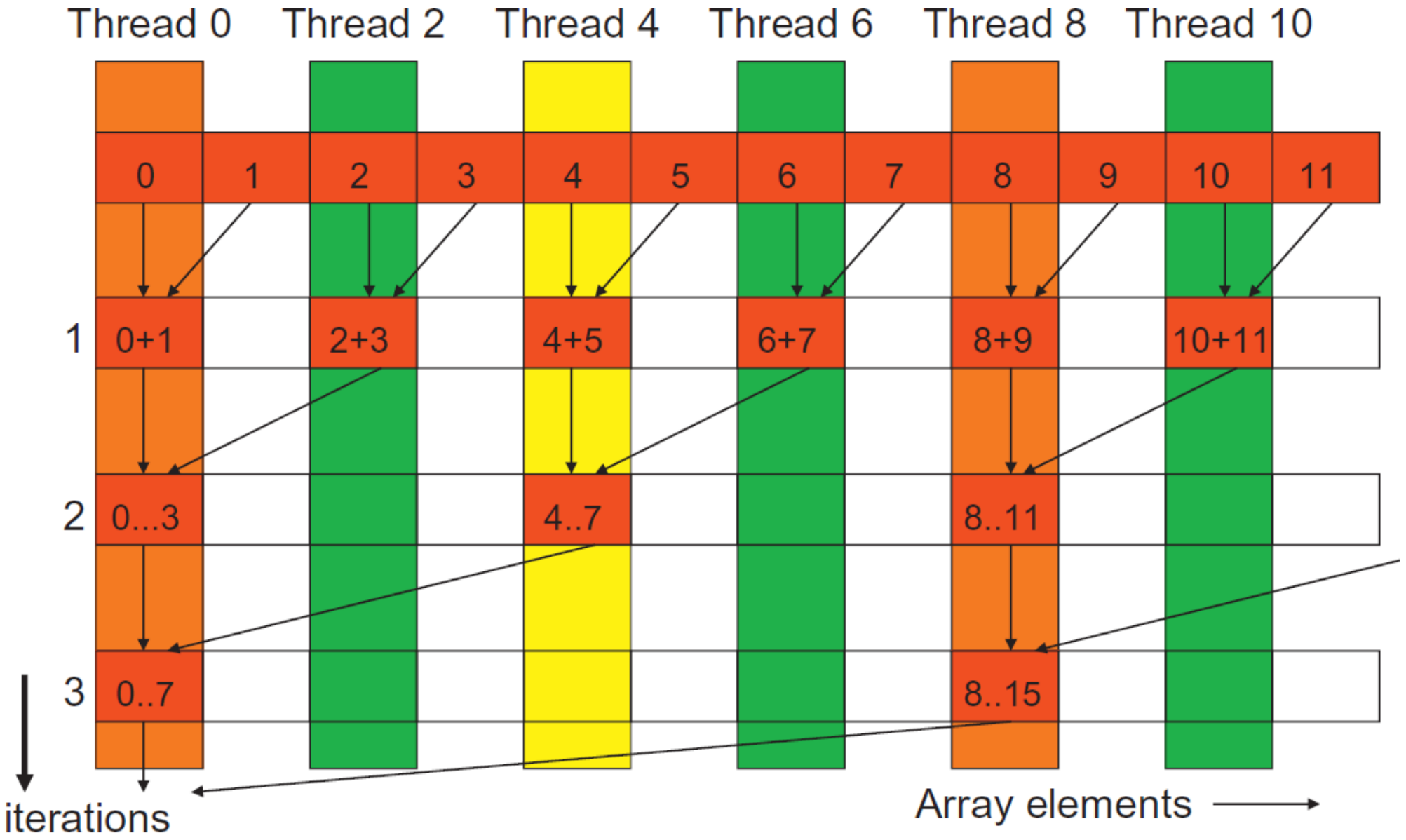
iterations

Array elements

Better way of doing a reduce?



Better way of doing a reduce?



Code (Reduce between threads in a block)

Assume we have already loaded array into **__shared__ float partialSum[]**

```
unsigned int t = threadIdx.x;
for (unsigned int stride = 1;  stride < blockDim.x;  stride *= 2)
{
  __syncthreads();
  if (t % (2*stride) == 0)
     partialSum[t] += partialSum[t+stride];
}
```

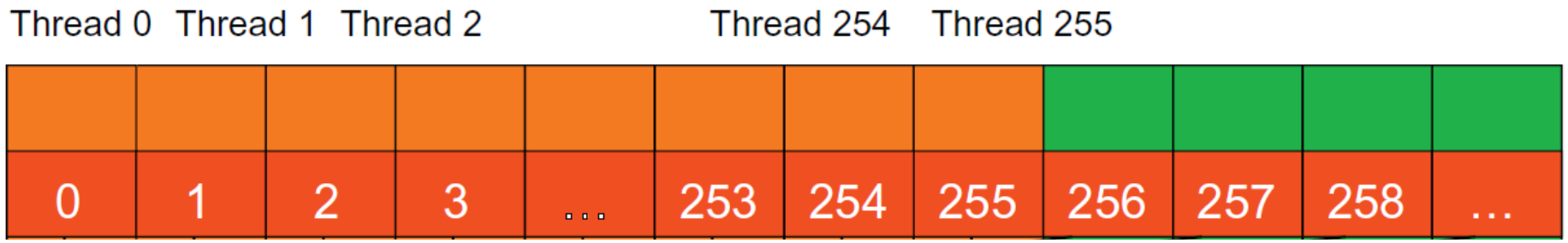
Questions?

Issues

- Branch!
- Some threads perform addition while others do not
- Threads that do not perform addition will do nothing
- No more than half of threads will be executing at any time
- All odd index threads are disabled right from the beginning!
- After the 5^{th} iteration, stride > 32, and entire warps in each block will be disabled, poor resource utilization but no divergence.

- How to mitigate it?

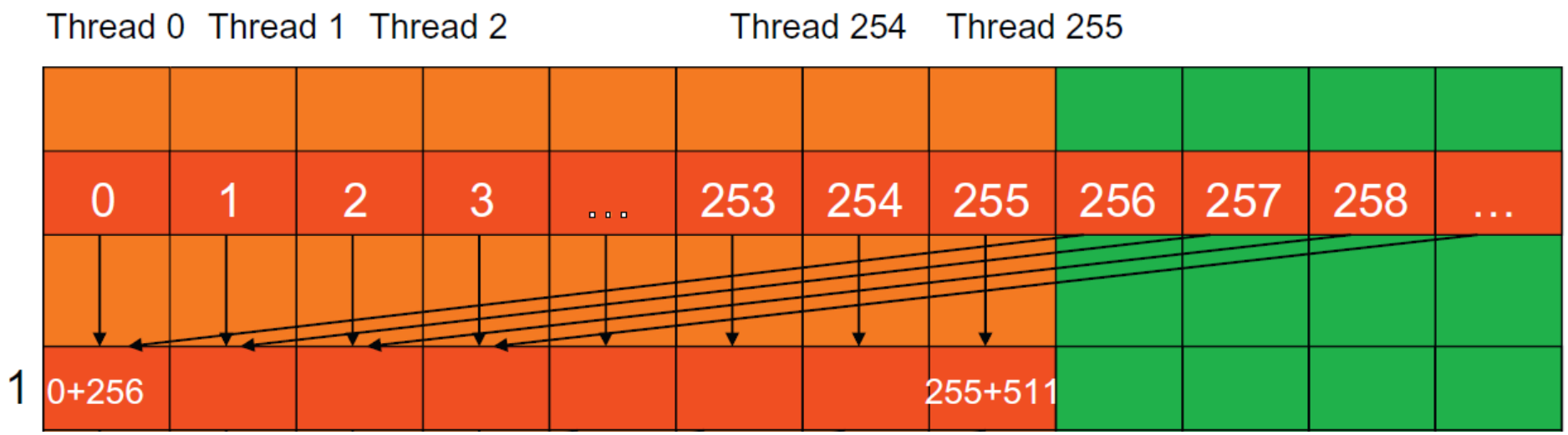
Better way of doing a reduce



iterations

Array elements

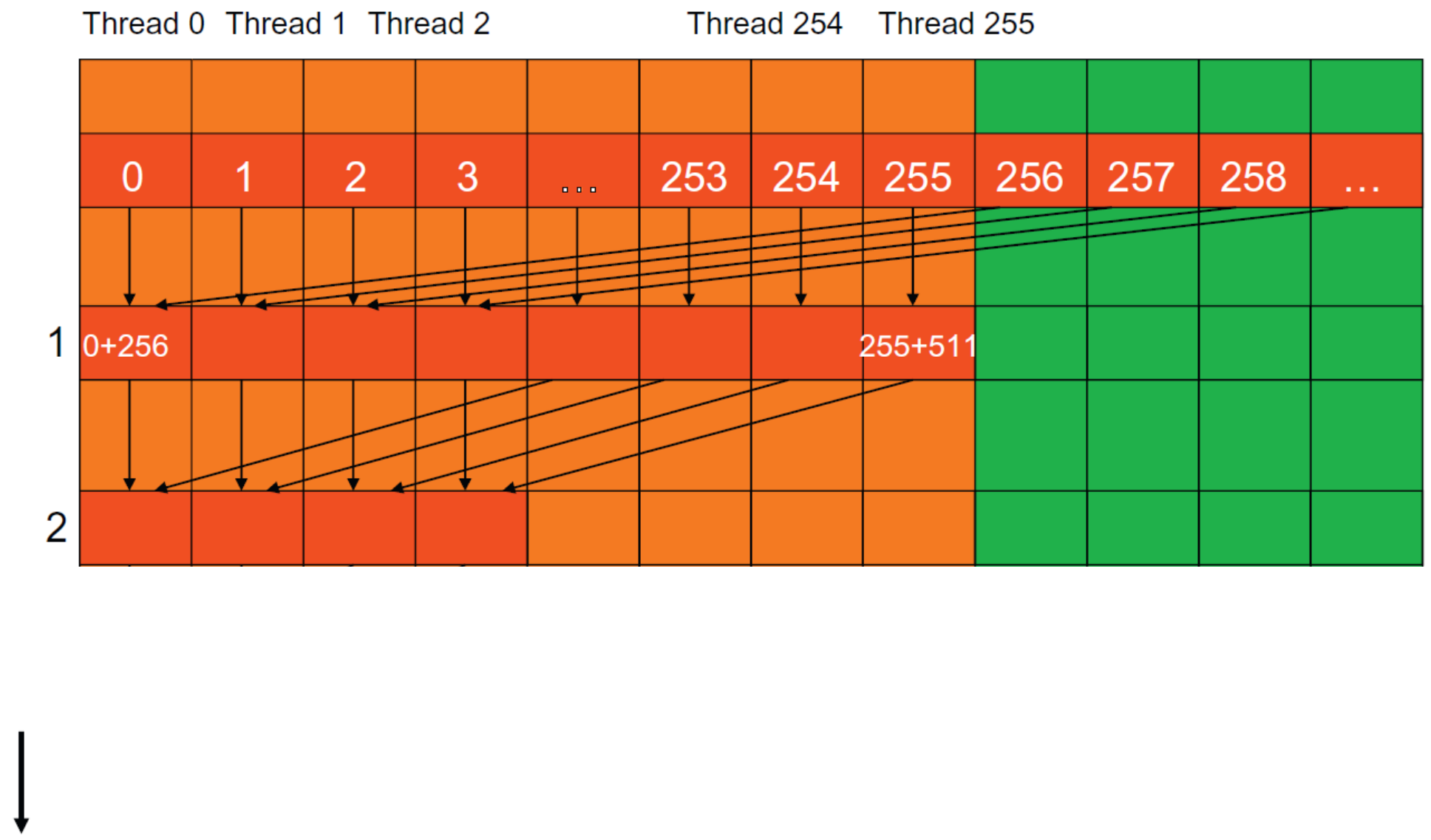
Better way of doing a reduce



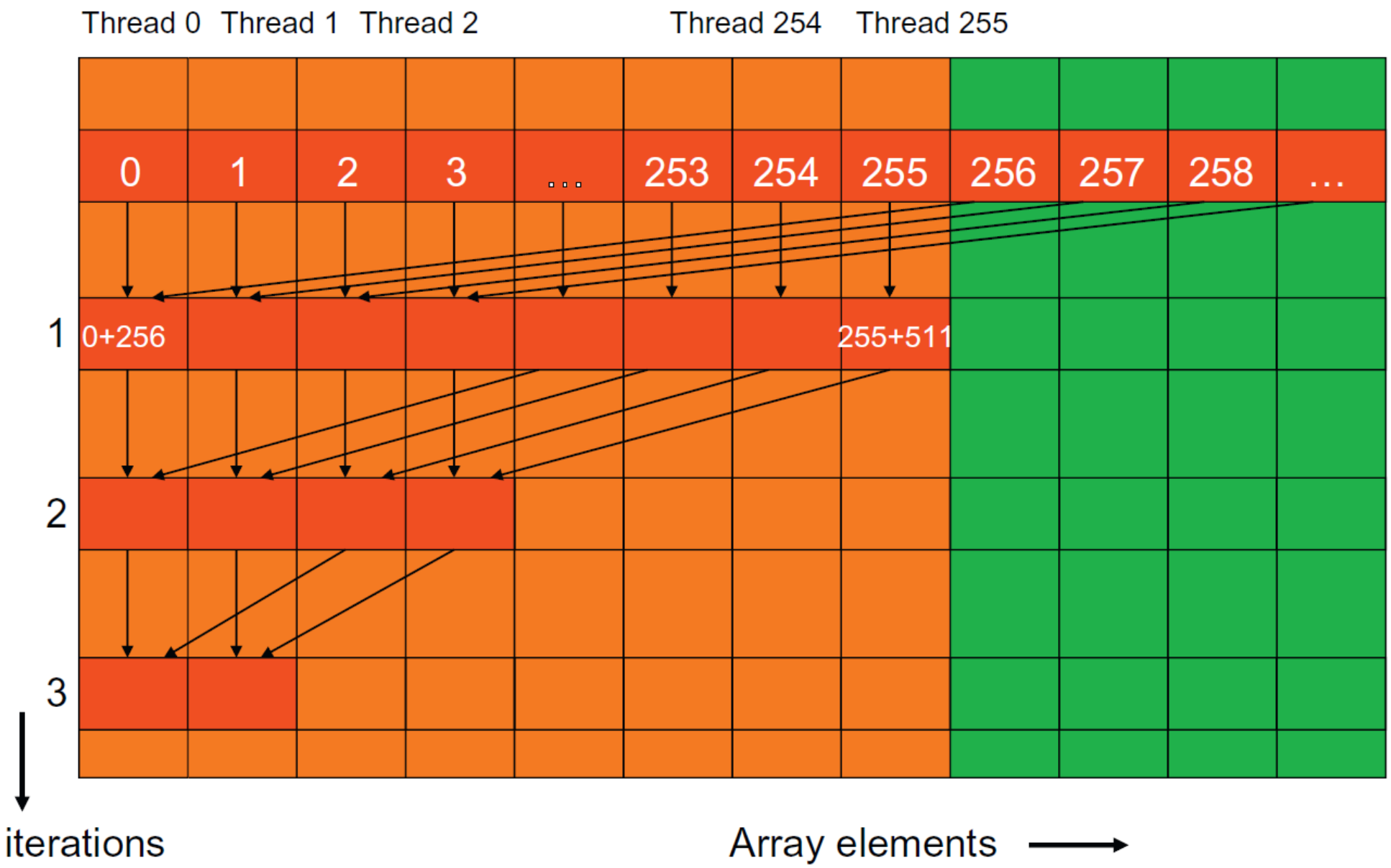
iterations

Array elements

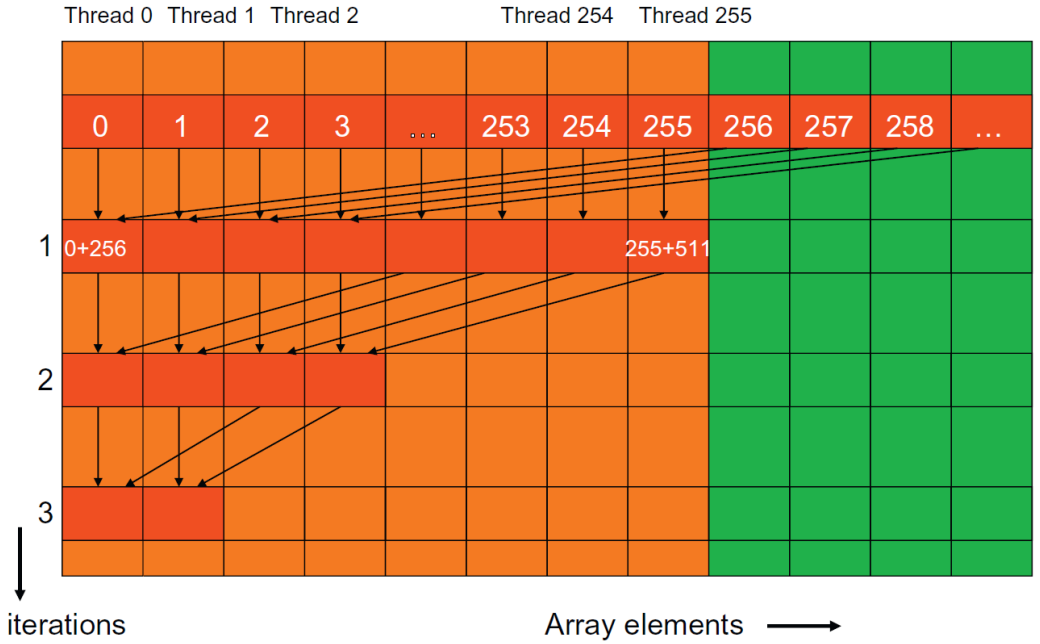
Better way of doing a reduce



Better way of doing a reduce



Better way of doing a reduce



Intuition: For the first steps, all the threads in a warp will do the same thing (either sum, or do nothing)

E.g., for a 512 threads block:

- 1° step: the fist 512 threads read data from the other 512 threads – 512 active threads
- 2° step: threads [0, 255] read data from [256, 511]
- ...
- After the 5° step, the stride will < 32, so we will have divergence (but only for the last few steps)

Code (Reduce between threads in a block)

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

2.  unsigned int t = threadIdx.x;
3.  for (unsigned int stride = blockDim.x/2; stride >= 1; stride = stride>>1)
4.  {
5.    __syncthreads();
6.    if (t < stride)
7.      partialSum[t] += partialSum[t+stride];
8.  }
```

What if N > block_size ?

- Shared memory only shared between threads in the same block
- How to reduce a vector larger than the number of threads per block?
- Suppose you need to reduce an array of v[N] elements, with N = Nblock*Nthread. We need to:

1. Use a kernel **`reduce<<<Nblock,Nthread>>>(v,partial)`** to get an array of Nblock partial values (**`partial`** is an array of Nblock elements allocated in global memory)

2. Use a kernel **`reduce<<<1,Nblock>>>(partial,result)`** to get the final result

Questions?

How to move data between GPUs?

CUDA + MPI

There are two possible solutions, based on whether MPI is GPU-aware or not:

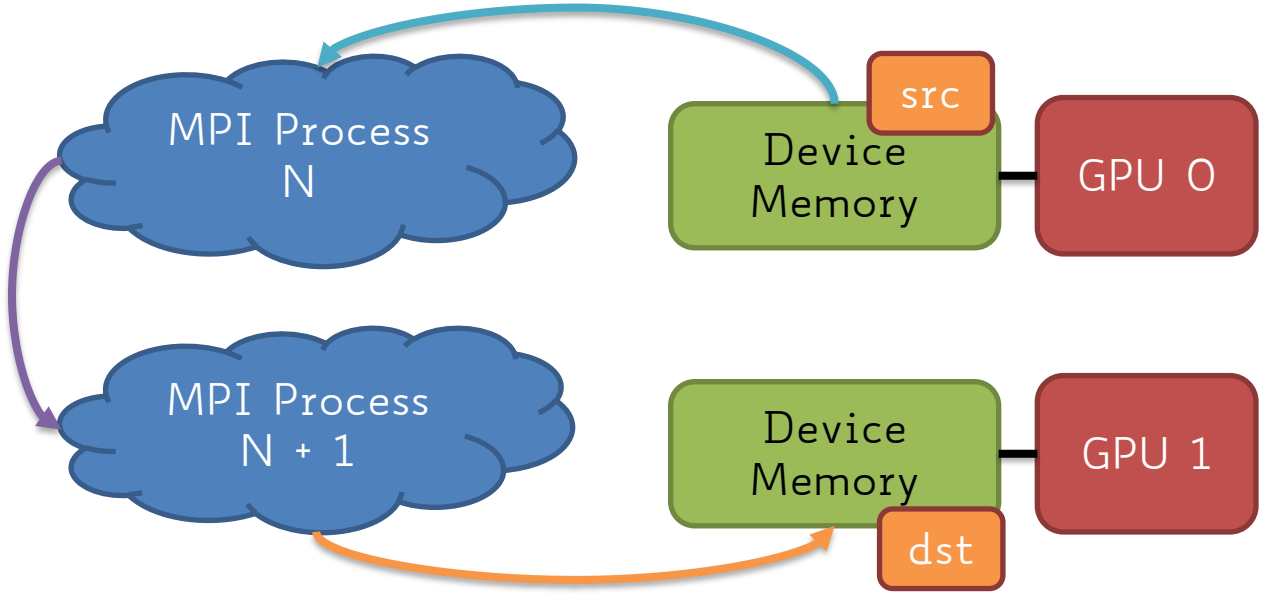
1. **MPI is not GPU-aware**: data must be explicitly transferred from the device to the host before making the desired MPI call. On the receiving side, these actions have to be repeated in reverse.

2. **MPI is GPU-aware**: MPI can access device buffers directly, hence pointers to device memory can be used in MPI calls.

At the time being, on the cluster MPI is **not** GPU-aware

MPI not GPU-Aware

- Source MPI process:
  - cudaMemcpy(tmp, src, cudaMemcpyDeviceToHost)
  - MPI_Send(tmp, …, N+1, …) // send tmp to N+1
- Destination MPI process:
  - MPI_Recv(tmp, …, N, …) // recv tmp from N+1
  - cudaMemcpy(dst, tmp, cudaMemcpyHostToDevice)



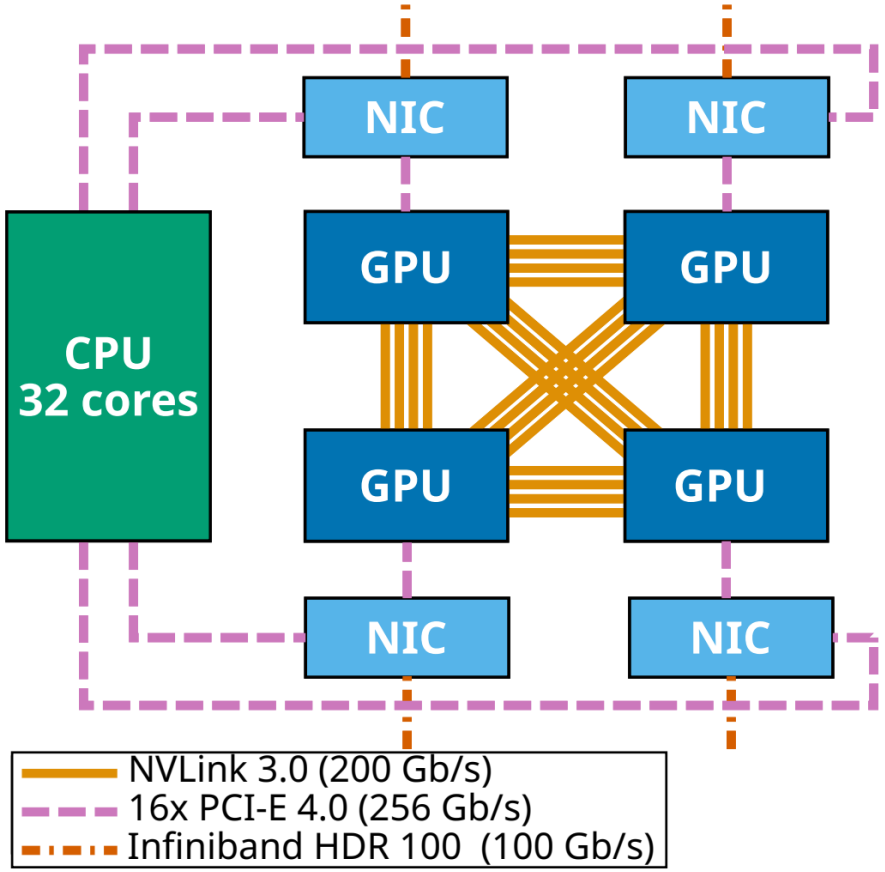
Beware!

- On the cluster, each GPU node has 2 GPUs
- Thus, you should run with 2 MPI process per node (each managing one GPU)
- You can specify the GPU you want to run on with **cudaSetDevice** e.g.:

```
if(rank % 2 == 0){
   cudaSetDevice(0);
}else{
   cudaSetDevice(1);
}
```

MPI GPU-Aware

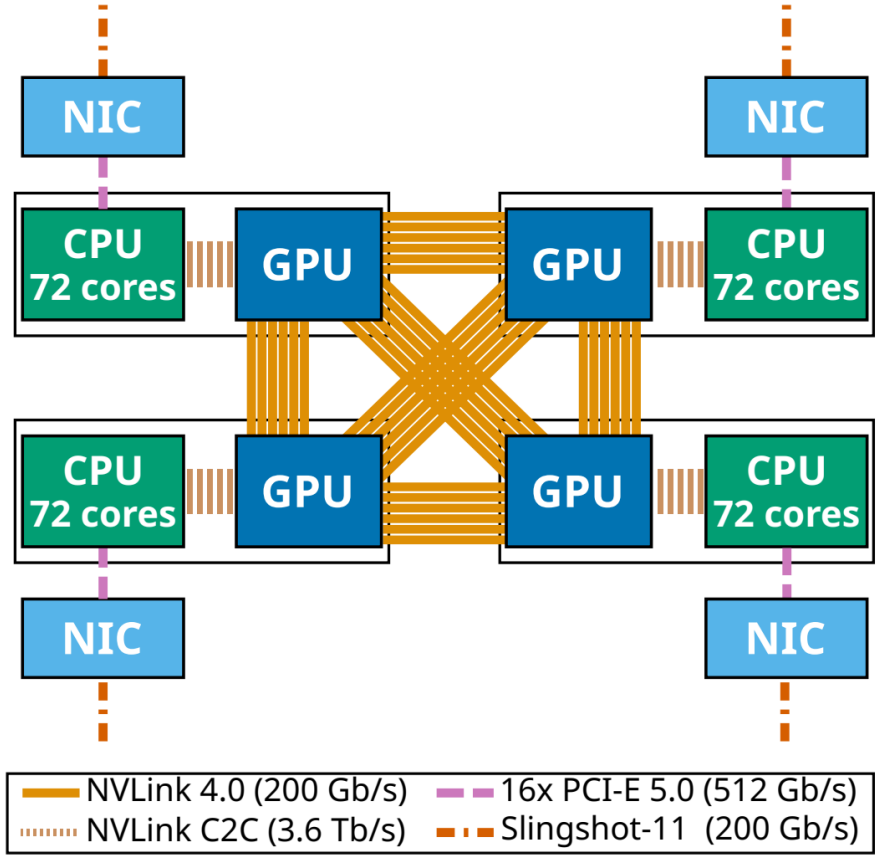
- No need to copy data between GPU and host memory
- This saves at least one memory copy on the sending and one memory copy on the receiving side.
- Useful for GPUs on the same node



(b) Leonardo.

MPI GPU-Aware

- No need to copy data between GPU and host memory
- This saves at least one memory copy on the sending and one memory copy on the receiving side.
- Useful for GPUs on the same node



(a) Alps.

MPI GPU-Aware

- No need to copy data between GPU and host memory
- This saves at least one memory copy on the sending and one memory copy on the receiving side.
- Useful for GPUs on the same node
- But also between different nodes
- For more details: https://arxiv.org/abs/2409.09874

*CCL: Beyond MPI

- *CCL libraries becoming more and more popular and developed by big companies (NCCL, RCCL, HCCL, etc...)
- Provide a few collectives (mostly those needed for ML training – allreduce, reduce-scatter, allgather, and a few others), and point-to-point
- They do not adhere to any standard, so they are more flexible and can innovate much faster
  - MPI is a huge standard and any change must ensure backward compatibility
  - MPI design is driven by the community through open discussion. Changes take years to be accepted and adopted by the implementations
- Still, there is value in having something standard. What if I have a system deploying both AMD and NVIDIA GPUs? (NCCL and RCCL do not talk to each other)

*CCL: Beyond MPI

In practice, *CCL often provide higher performance than MPI (at least for collectives)

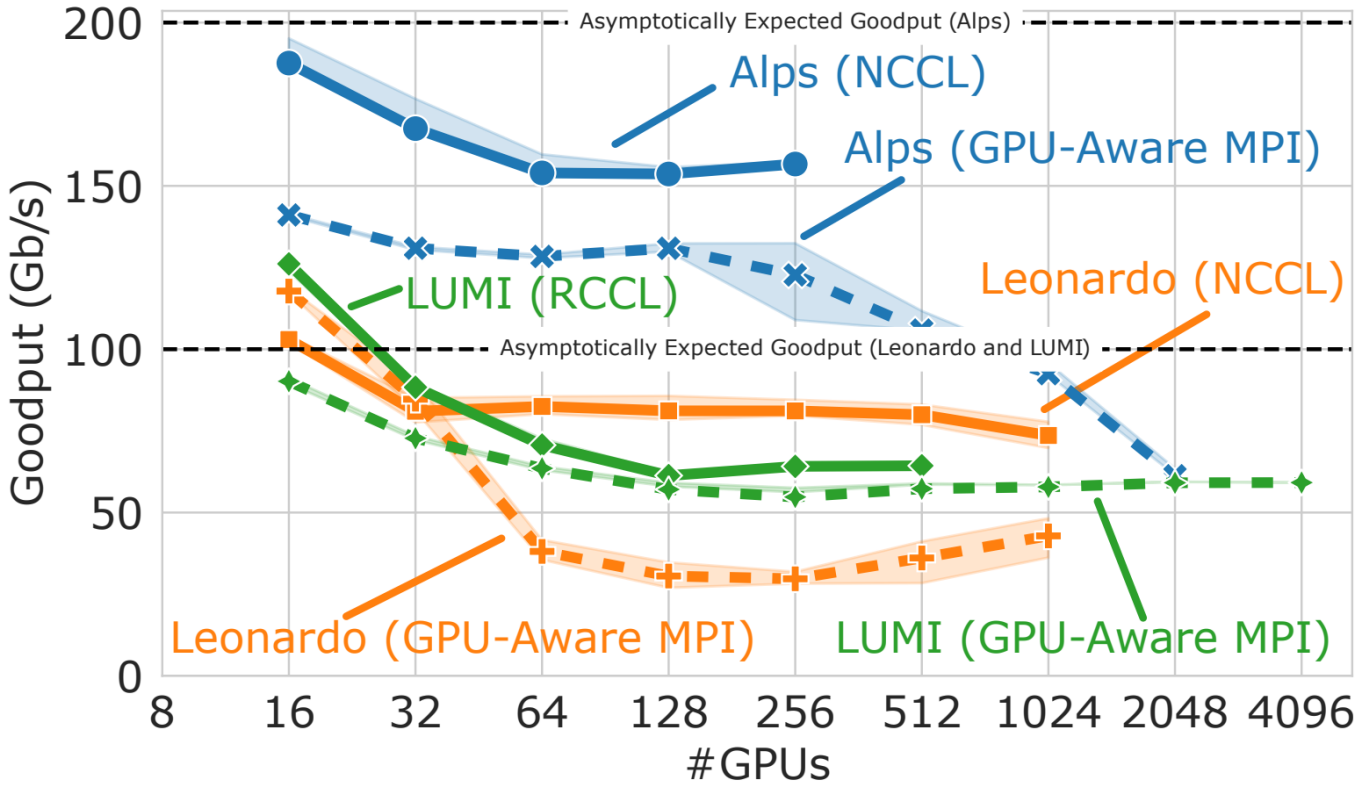


Fig. 9: 2 MiB alltoall scalability.

If you want to know more: https://danieledesensi.github.io/assets/pdf/2024_GPUGPU.pdf