Bandwidth and Contention

Felice Pantaleo
PH-CMG-CO

felice@cern.ch

Parallelism is all about symmetry

SPMD Phases

- *Initialize*
  - *Establish localized data structure and communication channels*
- *Obtain a unique identifier*
  - *Each thread acquires a unique identifier, typically range from 0 to N-1, where N is the number of threads*
- *Distribute Data*
  - *Decompose global data into chunks and localize them, or*
  - *Sharing/replicating major data structure using thread ID to associate subset of the data to threads*
- *Run the core computation*
- *Finalize*
  - *Reconcile global data structure, prepare for the next major iteration*

SPMD Phases

Thread IDs are used to differentiate behavior of threads

- Use thread ID in loop index calculations to split loop iterations among threads
  - Potential for memory/data divergence
- Use thread ID or conditions based on thread ID to branch to their specific actions
  - Potential for instruction/execution divergence

Both can have very different performance results and code complexity depending on the way they are done!

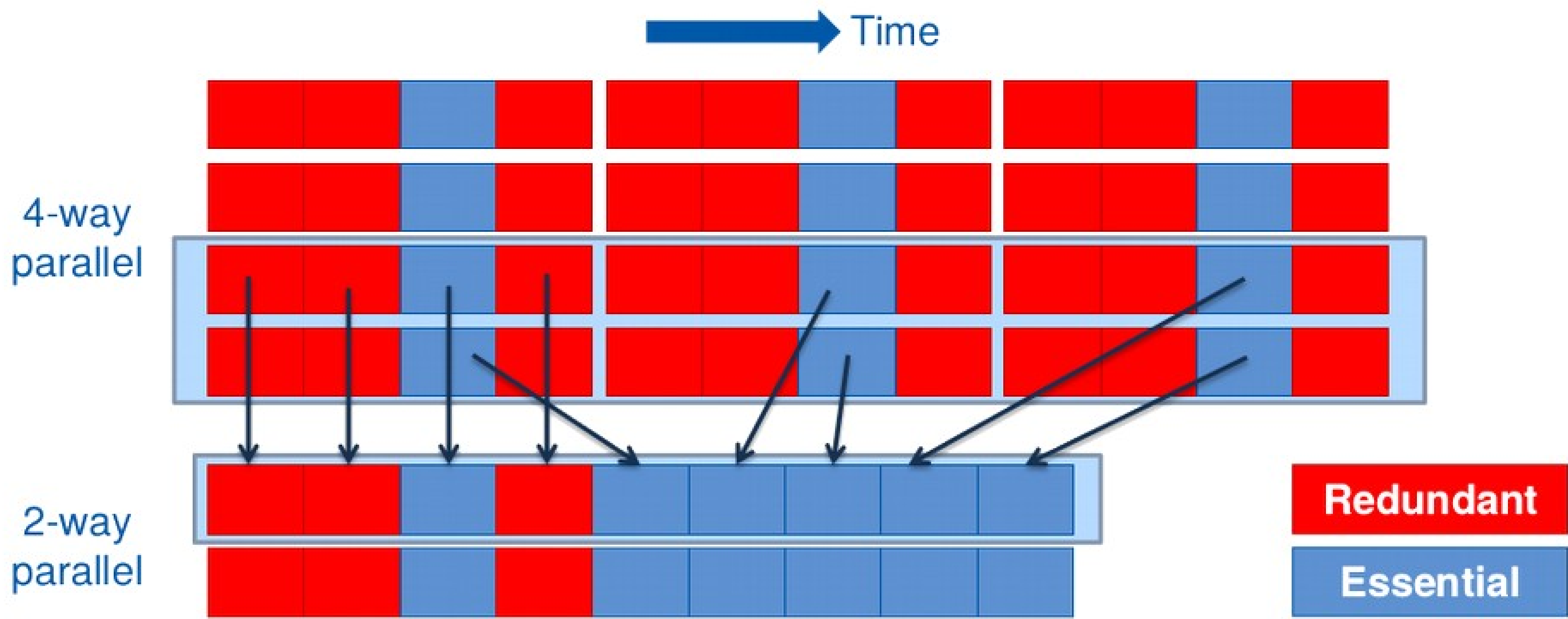
Conflicting data

Conflicting Data Updates Cause Serialization and Delays:

- *Massively parallel execution cannot afford serialization*
- *Contentions in updating critical data causes serialization*

Redundancy

- *Parallel execution sometimes requires doing redundant work*
  - *may result in too much redundant work and longer execution*



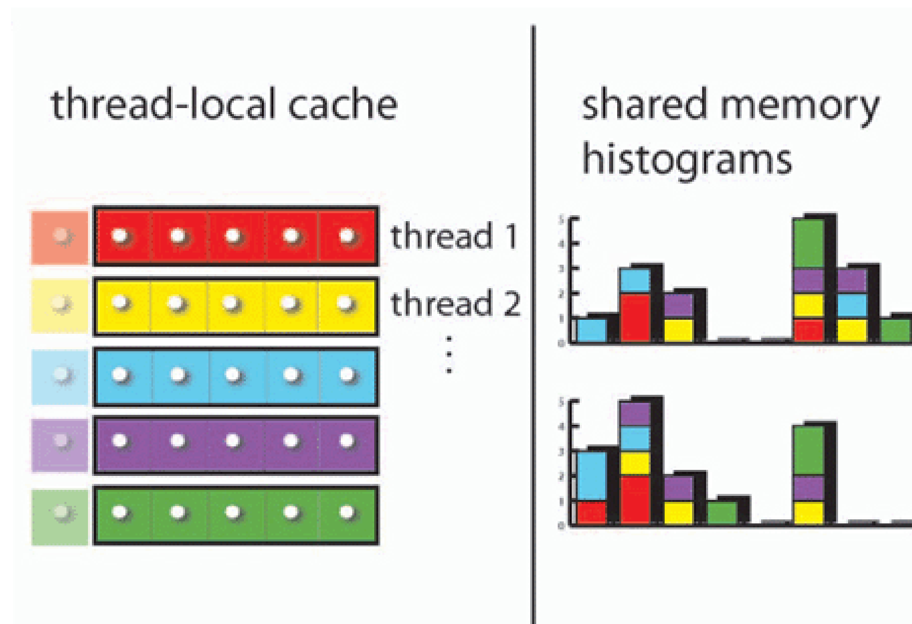
Mitigating contention

Contention can be mitigated with:

- Privatization
- Transformation of the access pattern

Privatization

- *Avoid frequent "phone calls" to the global memory and read/write the data locally as much as possible before updating the global value*
- *Make use of registers and shared memory for aggregating partial results*
- *Requires storage resources to keep copies of data structures*



Local and Global Queues

- *It's often useful to build queues or histograms in a parallel program*
- *Atomic functions cannot be interrupted while running.*
  - *On the GPU an atomic function is converted to one intrinsic*
- `atomicAdd(int* a, int value):`
  - *reads* `a` *at some address in global or shared memory*
  - *Increases* `a` *by* `value`
  - *Returns the old value*
- *Update the information about the number of elements in a queue atomically*
- *Keep the queue in shared memory*
- *At the end of the execution, update the global memory queue in a one-time copy*

Access pattern transformation

- Contention can also be mitigated with a transformation of the algorithm
- Example scatter to gather transformation

```
 1 int x[N]; //input
 2 int y[M]; //output
 3
 4 //parallelize here
 5 //1 thread per input element
 6 for(int i=0; i<N; ++i) {
 7
 8 ...
 9         for(int j=0; j<M; ++j) {
10
11                 x[i].foo(y[j]);
12                 ...
13         }
14
15 }
```

Access pattern transformation

- It can be read like: for each input update each output
- This is called Scatter approach
  - All threads have conflicting updates to the same output elements
  - Parallelization requires the use of atomics to update the output
  - If the number of threads is large, the program can become really slow

Access pattern transformation (ctd.)

CERN

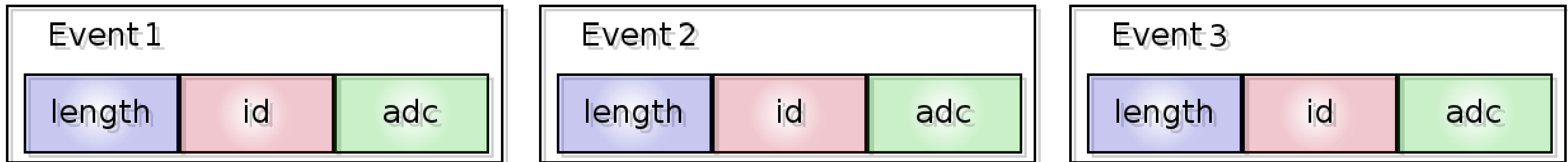
- *The transformation of the algorithm to "Owner Computes" (Gather), can make the parallel code much faster*
  - *All the threads can read the same input element at the same time*
  - *Does not introduce contention nor serialization*
  - *Can be consolidated through caches or local memories*

Owner Computes (ctd.)

```
 1 int x[N]; //input
 2 int y[M]; //output
 3
 4 //parallelize here!
 5 //1 thread per output element
 6 for(int j=0; j<M; ++j) {
 7
 8 ...
 9         for(int i=0; i<N; ++i) {
10
11                 y[j].foo(x[i]);
12                 ...
13         }
14
15 }
```

AoS to SoA conversion

Data come in Arrays of Structures:



SIMD architectures benefit from the conversion of the input data from Array of Structures (AoS) to Structure of Arrays (SoA)