A parallelization exercise

Scenario

We have decided to start evaluating some parallel computing environment with «an exercise»: parallelize a well known algorithm.

One of the selected tasks was sorting, to be developed with a serial paradigm and then parallelized via OpenMP and on GPU HW with CUDA

Performance measurements of the developed code allow to make some considerations on different technologies and also to gain a better understanding of technology internal mechanisms

Sorting Algorithm

Algorithm selection is obviously the first step in this exercice. The algorithm have to

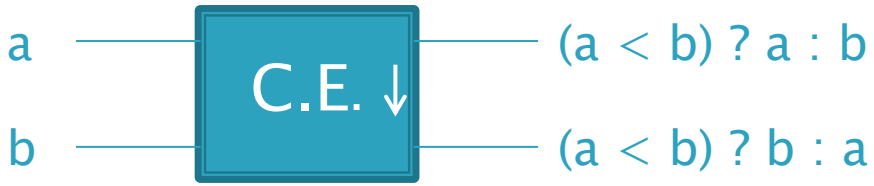
- Be parallelizable
- Be scalable to a large number of nodes

It should also have good performances even if not necessary the best on the market

The choice was to employ a sorting network implementing the «Bitonic Sorting» algorithm
[Knuth – The Art of Computer Programming Vol. 3]

Bitonic Sorting [1/3]

Fundamental element of the sorting network is the comparator-exchanger:



The network is composed by indipendent C.E., sorting is due to connections among its elements.

C.E. at the same depth can run in parallel

Bitonic Sorting [2/3]

The sorting network is built iteratively (i.e. by induction) with a tipical divide-and-conquer approach.

Base of the induction is a k-order bitonic sorter: a network able to sort a k-length bitonic sequence.

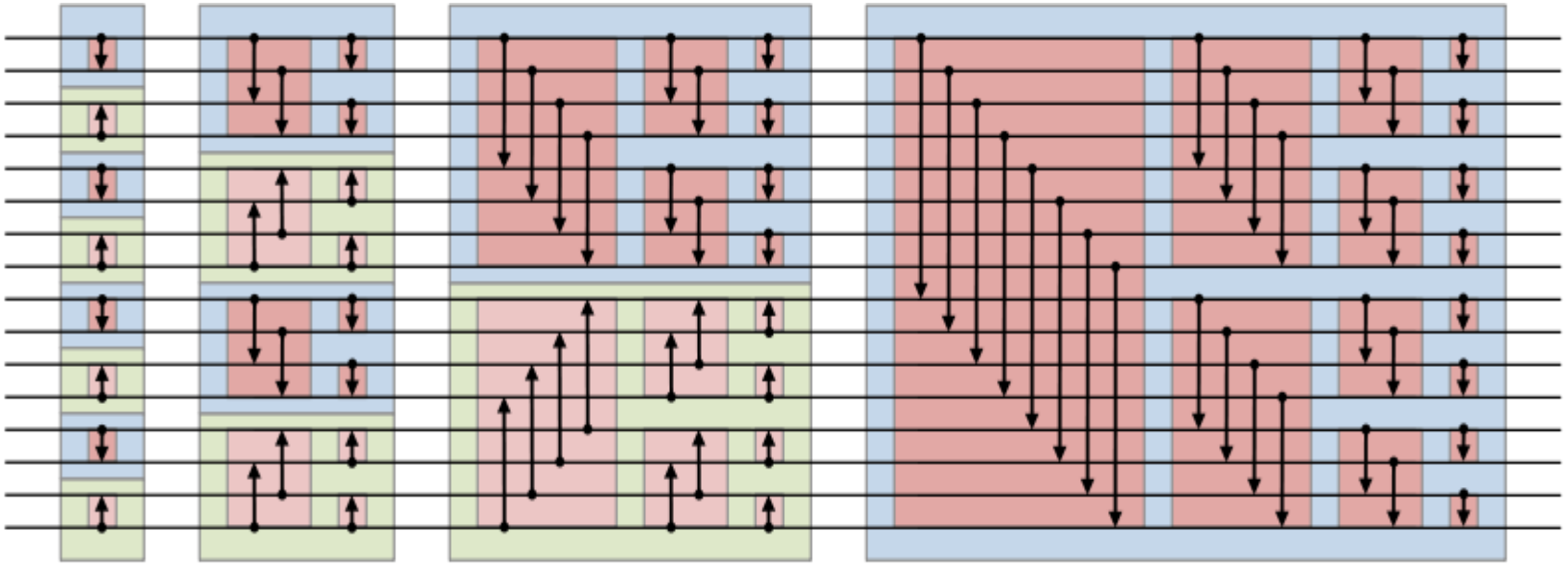
Two k-lenght ordered sequences can be merged to form a new bitonic sequence that can be sorted by a (2k)-order bitonic sorter.

Proceeding with the iteraction it's possible to construct a network able to sort a sequence of any length (power of two)

[Proof: use 0-1 principle and proceed by induction]

Bitonic Sorting [3/3]

An example of sorting network for 16 inputs



Number of C.E. $= O\left(N \log_2^2 N\right)$ – Network delay $= O\left(\log_2^2 N\right)$

Test system setup

Measurements were done with the following system:

SuperMicro GPGPU SuperServer

- MB: SuperMicro X8DTG-DF (Intel 5520 chipset)
- CPU: 1xIntel Xeon E5630 (4 cores @ 2.53GHz, 2 HT per core)
- RAM: 12 GB DDR3@ 1333MHz

- GPU: NVidia Tesla M2050 (448 CUDA cores, 3GB RAM @ 1.55GHz, 1.03TFlops/515GFlops)

- S.O.: RHEL 6 (gcc 4.4.5, OpenMP 3.0, CUDA 3.2)

Serial Bitonic Sorting

The following is a pseudo-implementation of the sorting network

```
bitonicSortSerial(theArray)
    for step from 0 to MaxStep // Outer block loop
       for substep from 0 to step // Inner block loop
           // Calculate Indexes for comparison
           // and block parameters
           for  i from 0 to theArray.length / 2
               // C.E. execution loop
               if (i is part of the ascending network)
                  CompareAndExchange(Ascending Order)
               else
                  CompareAndExchange(Descending Order)
```

OpenMP Bitonic Sorting [1/4]

Three nested loops allow to fork parallel threads at different depths of the code.

1. At the C.E. level – inside the inner loop: a thread is forked for each C.E.. Each thread execute a C.E. code and then exit (interesting for comparison with CUDA code but in general a very bad idea)
2. At the sub-block level – middle loop: in this case a thread executes several C.E. inside sub-blocks at the same depth (threads are «recycled» – cpu cycles used for effective computing, not only to span threads – C.E. executed are independent)
3. At the block level – in the outer loop: OpenMP manage the spanning of several threads (one per HT unit) to execute portions of the code inside the three loops (OpenMP can «recycle» threads, but C.E. aren't always indipendent: require explicit synchronization)

OpenMP Bitonic Sorting [2/4]

The 3 solutions correspond to instruct OpenMP to parallelize (with omp parallel directive) the following sections of the pseudo-code

```
bitonicSortSerial(theArray)
    for step from 0 to MaxStep // Outer block loop
        for substep from 0 to step // Inner block loop
            // Calculate Indexes for comparison
            // and block parameters
            for  i from 0 to theArray.length / 2
                // C.E. execution loop
                if (i is part of the ascending network)
                    CompareAndExchange(Ascending Order)
                else
                    CompareAndExchange(Descending Order)
```

Solution 3

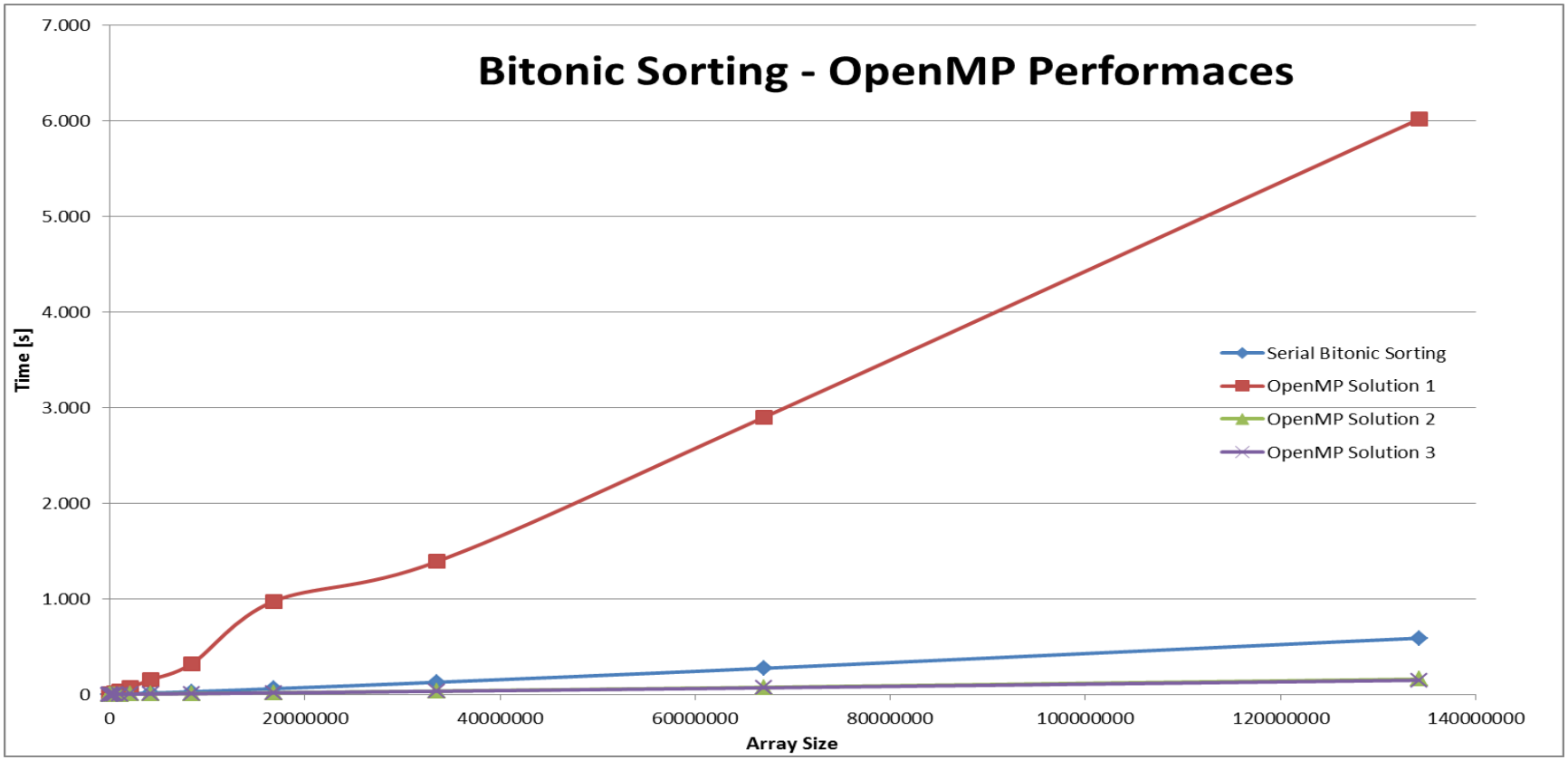
Solution 2

Solution 1

Solution 3 requires explicit synchronization here (omp barrier)

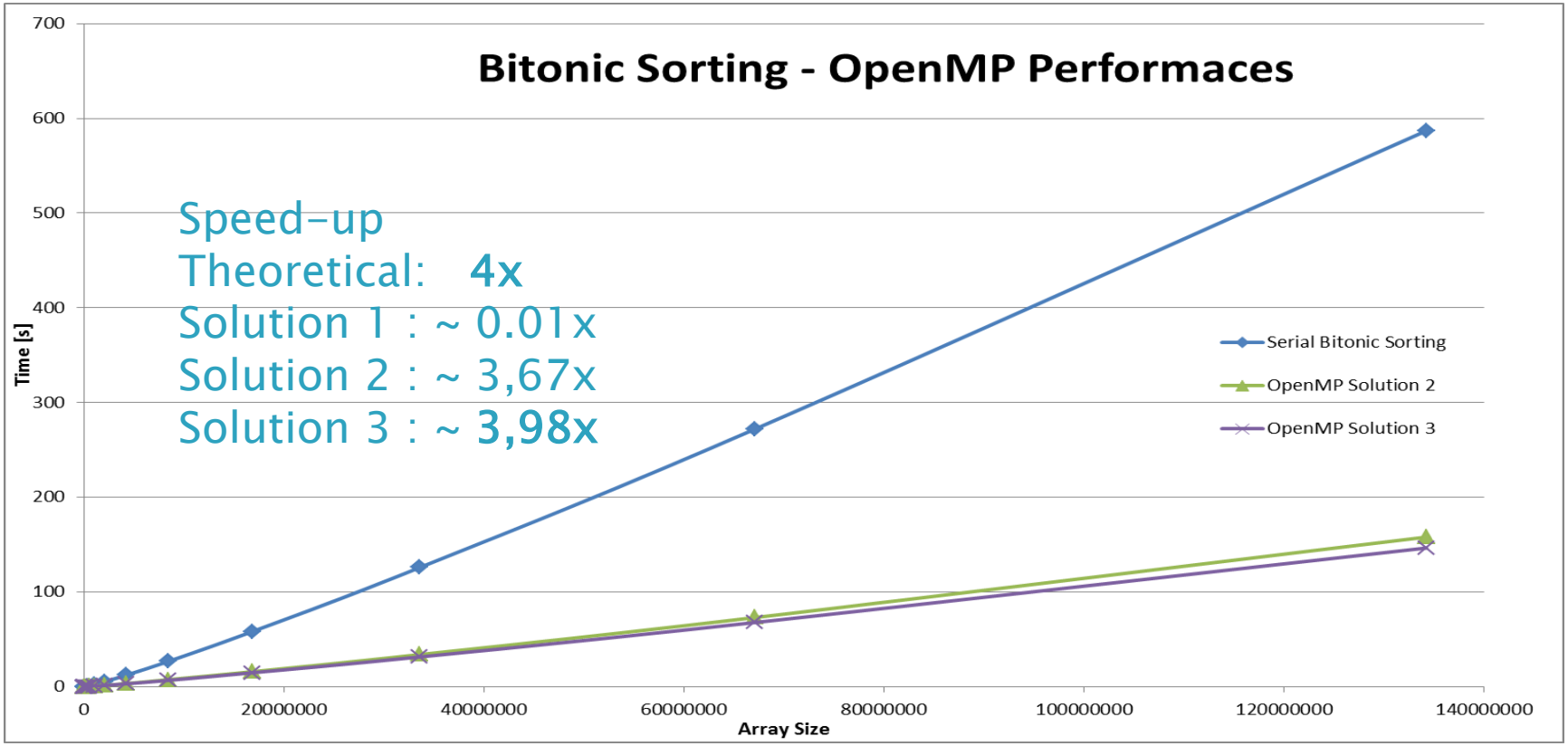
OpenMP Bitonic Sorting [3/4]

Performances measured on the test system:



OpenMP Bitonic Sorting [4/4]

Focusing on the two last solutions



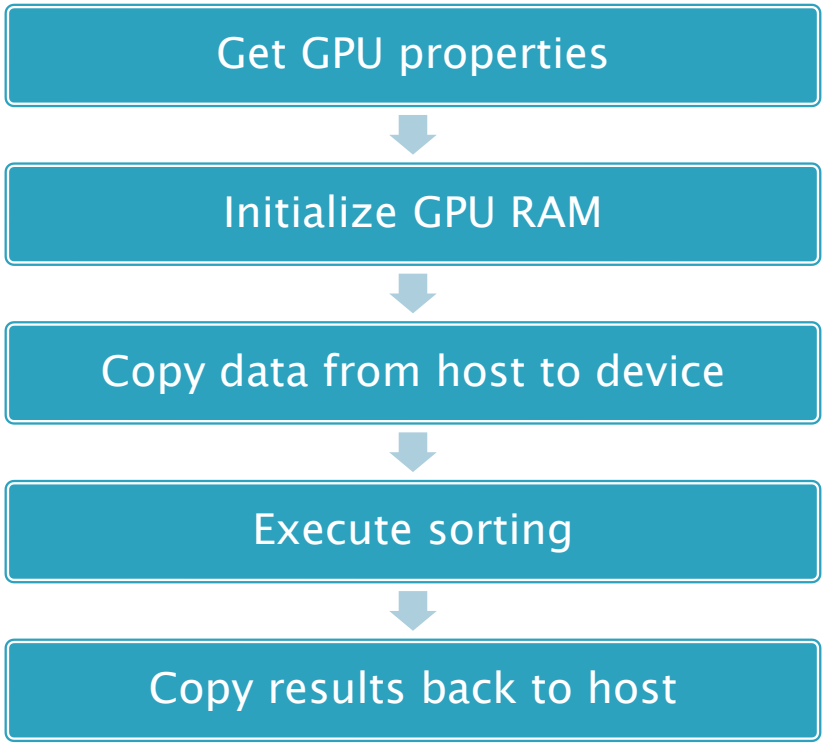
CUDA Bitonic Sorting [1/7]

Testing was done employing a NVidia Tesla M2050 GPU board

- Array of streaming processor with 448 CUDA cores
- SIMT (Single Instruction Multiple Thread) technology
- Threads are divided in warps (blocks of 32 threads). A processor executes the same instruction on each thread of a warp.
- Execution is in order. No branch prediction or speculative execution. When a processor encounters a branch inside a warp, it serializes the execution of each path of the branch.
- Warp scheduling is done in hardware with two warp schedulers able to issue one instruction per cycle.
- Execution context (IP, status registers, etc.) is maintained on-chip during the execution of a warp: no context switching cost.

CUDA Bitonic Sorting [2/7]

All the CUDA implementations of the Bitonic Sorting algorithm have the following structure:



CUDA Bitonic Sorting [3/7]

Several types of implementation were developed to investigate :

- The impact of thread size/lifetime (parallelization at different depths of the network)
- How thread grouping affects performances
- Execution time of part of the code on host and part on the GPU v.s. execution of the code completely on the GPU
- Impact of different synchronizations techniques (thread explicit synchronization on the GPU, submission of kernels inside streams, etc.)
- Branching impact on the streaming processor architecture

CUDA Bitonic Sorting [4/7]

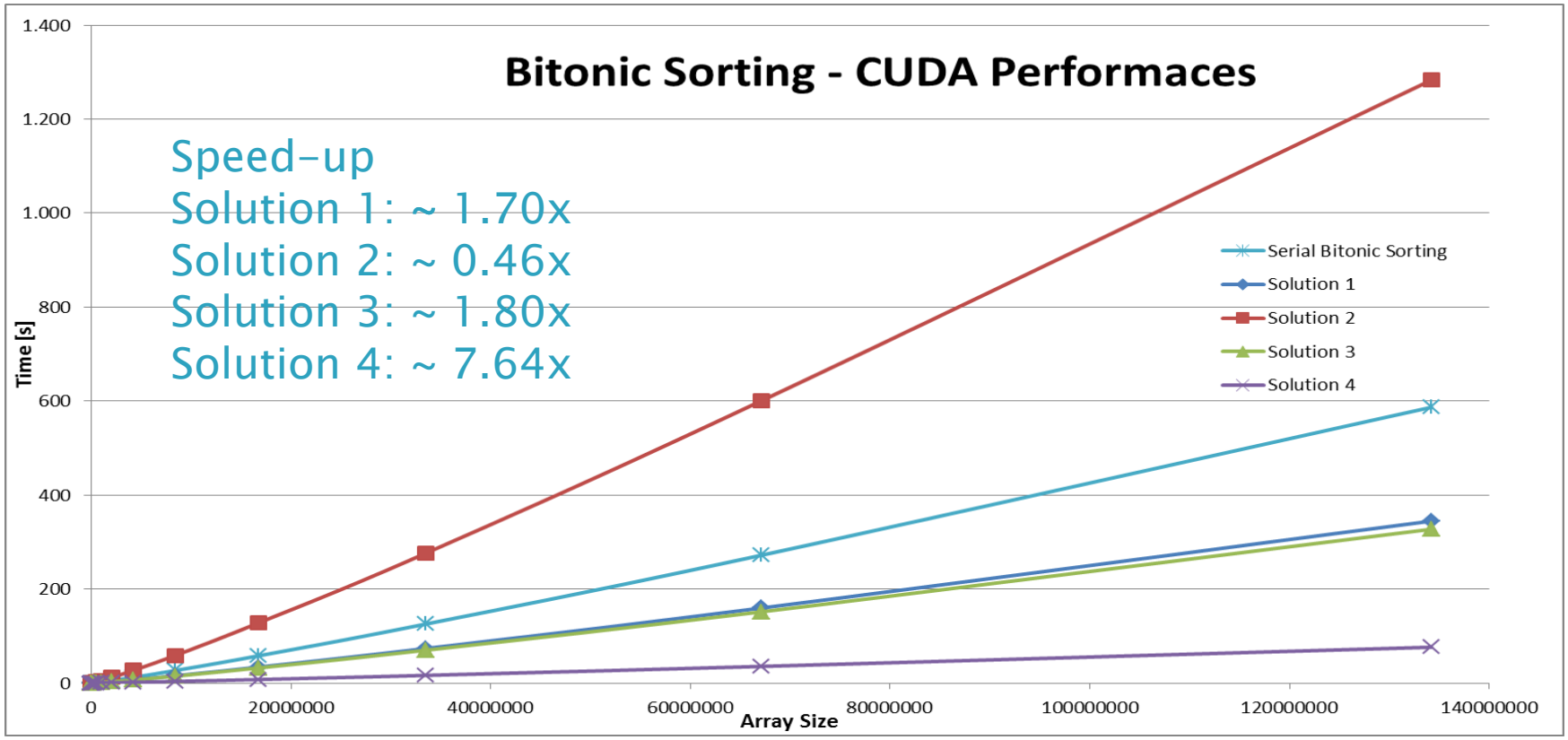
In particular we have developed 3 versions with light kernels implementing a single C.E. each. Auxiliary code (index computation, etc.) was executed on the host. Measurement were done with the following setup:

1. C.E. kernels grouped as N blocks of 1 thread (<<<N, 1>>>). Implicit synchronization done on the host
2. C.E. kernel grouped as 1 block of N thread (<<<1, N>>>). Implicit synchronization done on the host.
3. Kernel grouping as solution 2. Submission of all the threads at once. Synchronization obtained with streams

We have also developed a version of the algorithm with an 'heavier' kernel, that executes more step of the sorting network (less threads with longer lifetime – reduced scheduling overhead). In this case all the network was executed on GPU (Solution 4)

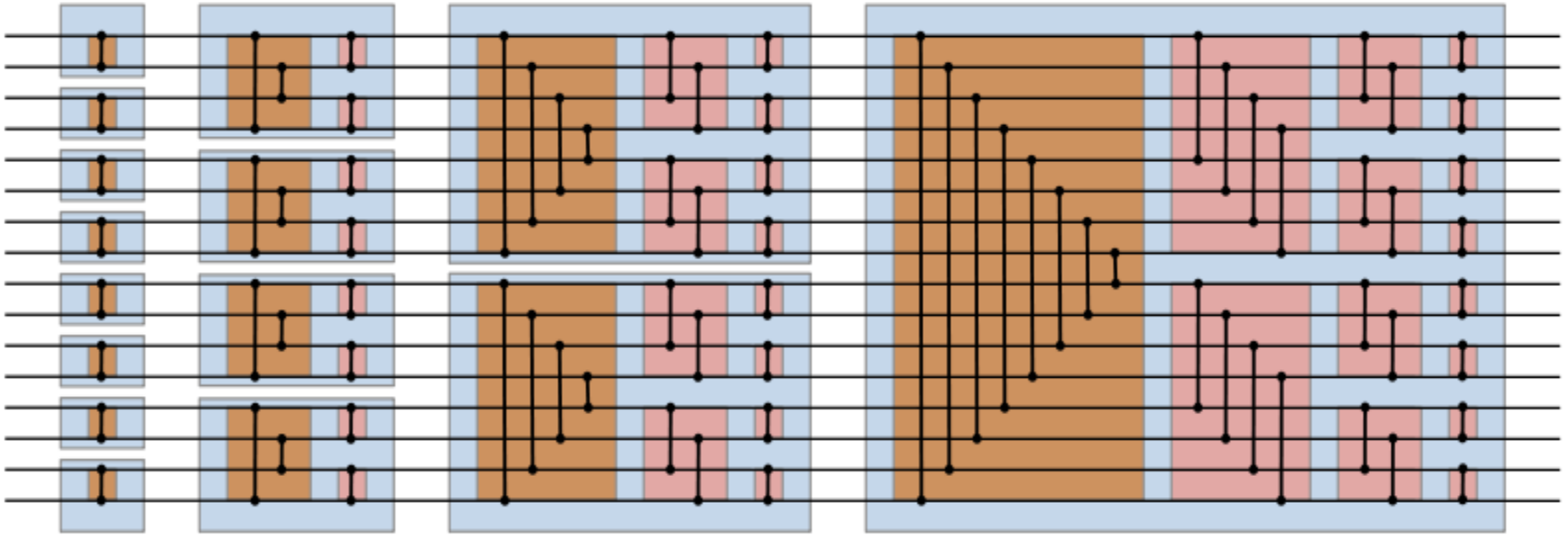
CUDA Bitonic Sorting [5/7]

Performances measured on the test system:



CUDA Bitonic Sorting [6/7]

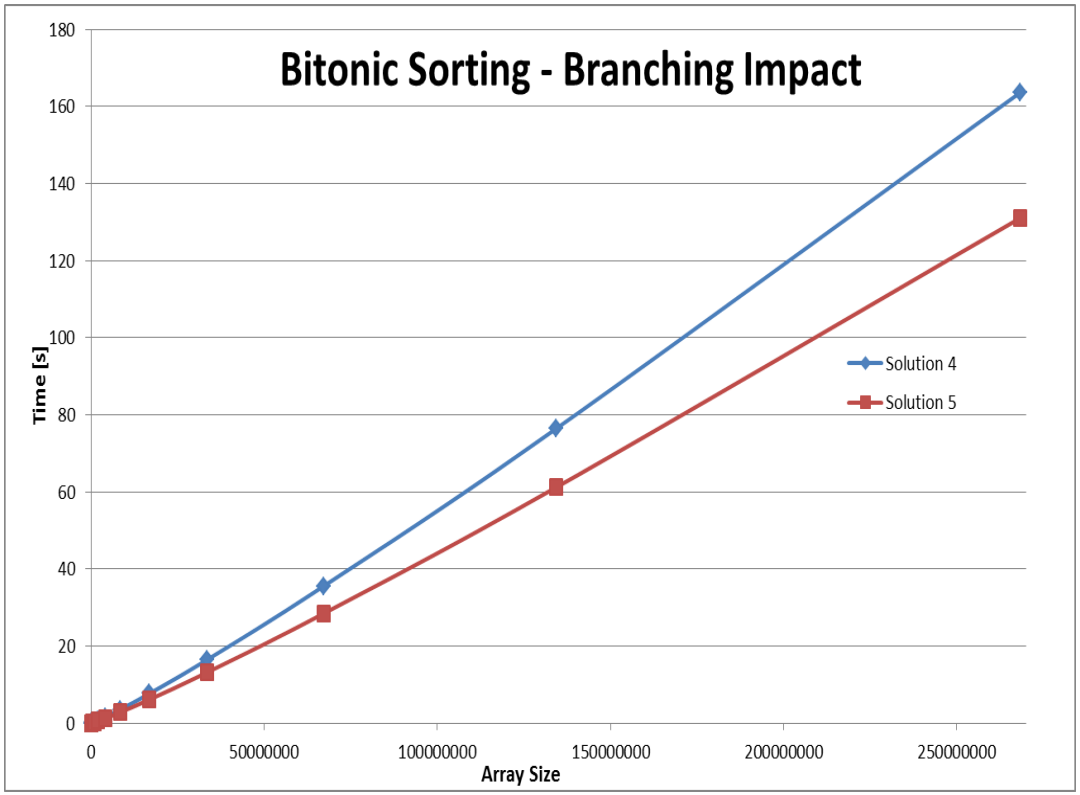
Solution 4 (full execution of the network on GPU with 'persistent' threads) was also employed to study the impact of branching, implementing the following network



that with a different routing execute the same sorting algorithm but employing only ascending C.E.

CUDA Bitonic Sorting [7/7]

The following is the comparison of the code with ascending and descending networks (Sol. 4) and with ascending networks only (Sol. 5)



Speed-up
Solution 4: ~ 7.64x
Solution 5: ~ 9.53x

Removing the need to distinguish the comparison direction of C.E. translates in a ~25% speed-up gain

Conclusions

Exploiting the power of multi/many cores machines requires a carefull design of the code.

Parallelization with OpenMP is quite straightforward. Good performances can be obtained with a careful weighting of the synchronization conditions and thread lifetime.

CUDA gives more options to the developer, on the other hand different structures of the code may yield high variations in the execution time (thread grouping, synchronization, etc.)

At the end the hardest part of the design is the Algorithm, but is also the phase where the code can gain the higher speed up.