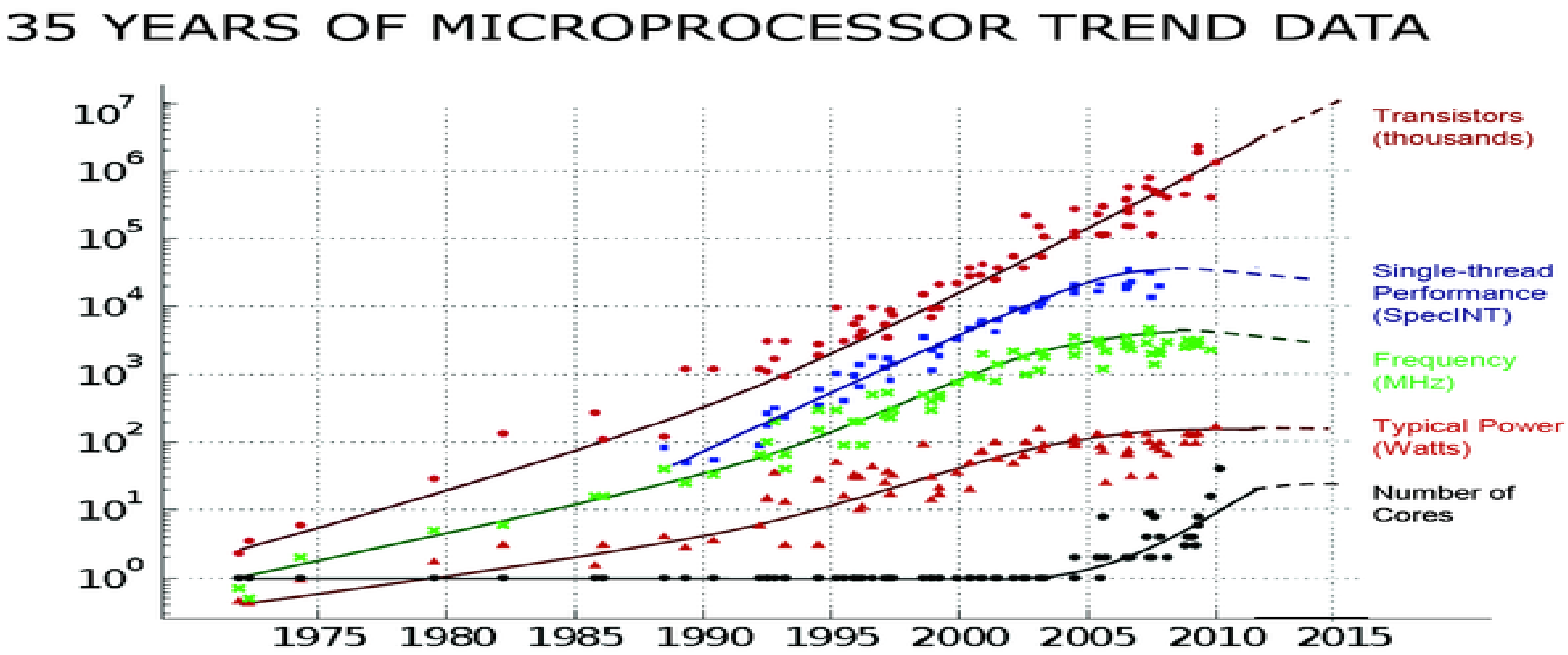
Exploiting Task-Parallelism on GPU Clusters via OmpSs and rCUDA Virtualization

Adrián Castelló, Rafael Mayo, Judit Planas, Enrique S. Quintana-Ortí

RePara 2015, August – Helsinki, Finland

Power/energy/utilization walls!

- End of Dennard's scaling
- Moore Law in place
- Dark silicon



Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

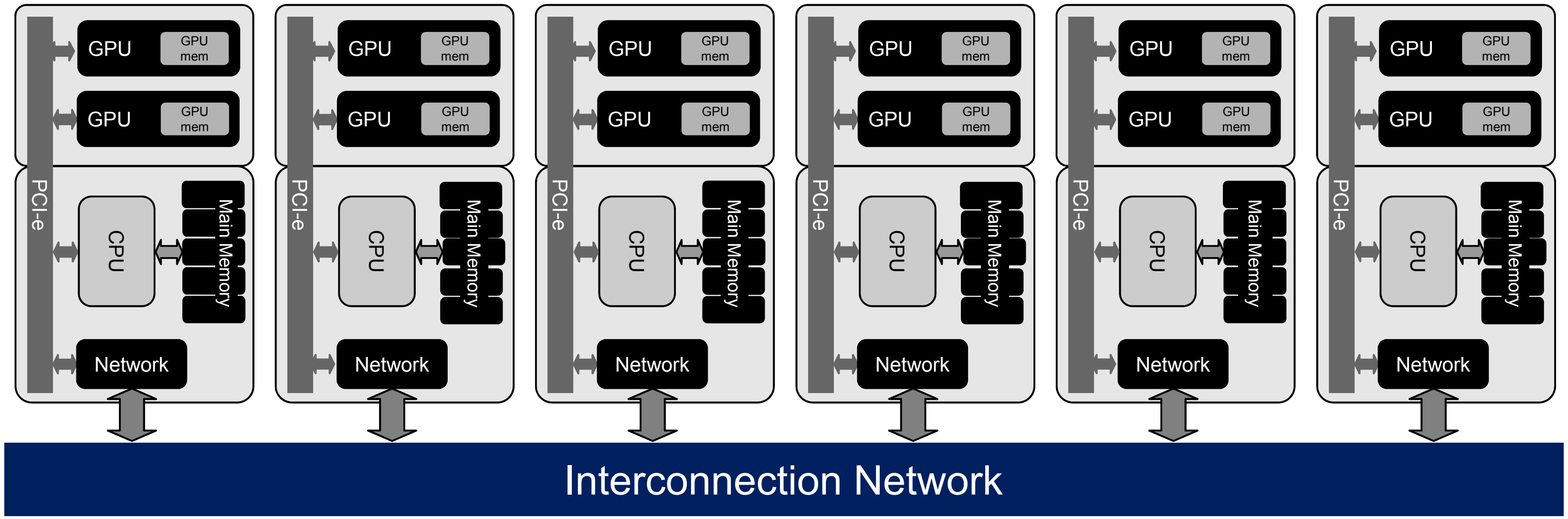

NVIDIA TESLA
NVIDIA TESLA

GPU computing: Why?

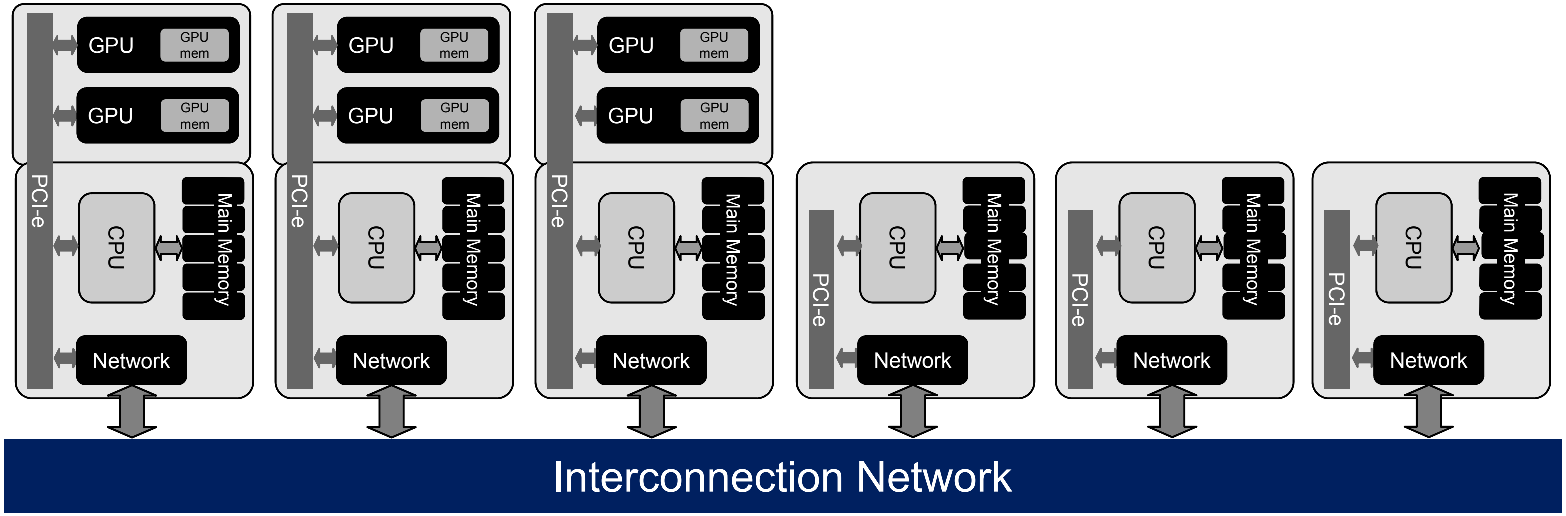
- Moderate Price
- High performance
- Favorable throughput-per-Watt
- Powerful and simple APIs (remember Cell B.E.)
  - OpenACC
  - CUDA
  - OpenCL

From the programming point of view:

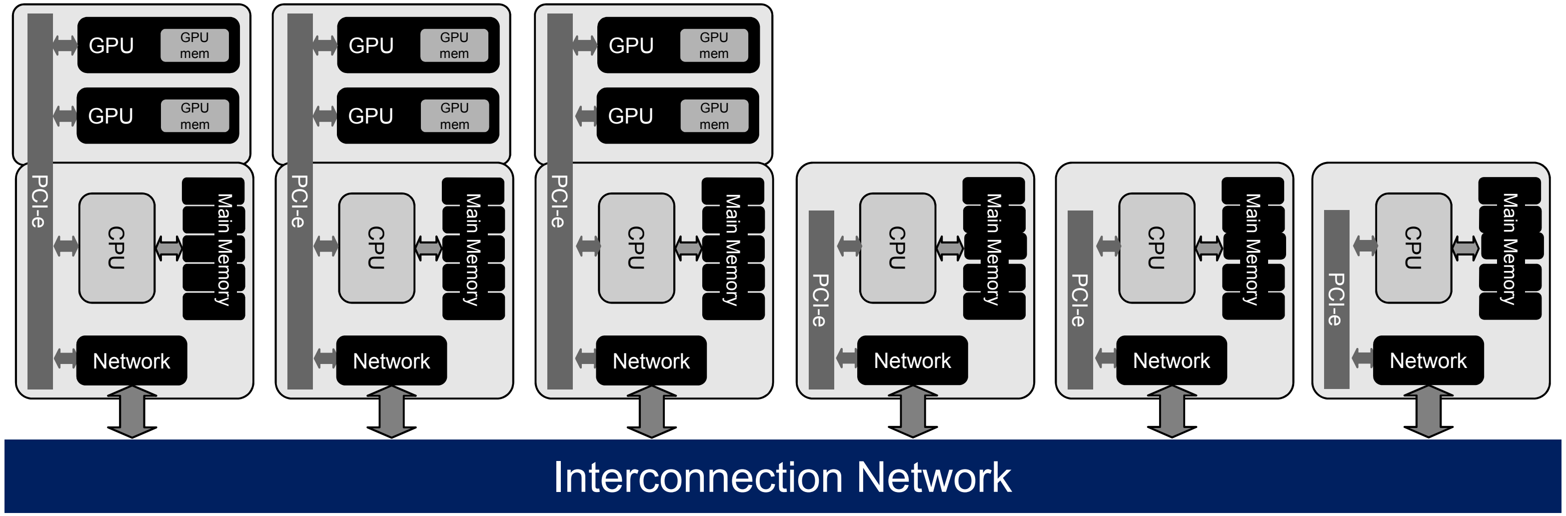
- A collection of nodes, each with:
    - one or more CPUs (with several cores per CPU)
    - one or more GPUs (1-4)
- An interconnection network



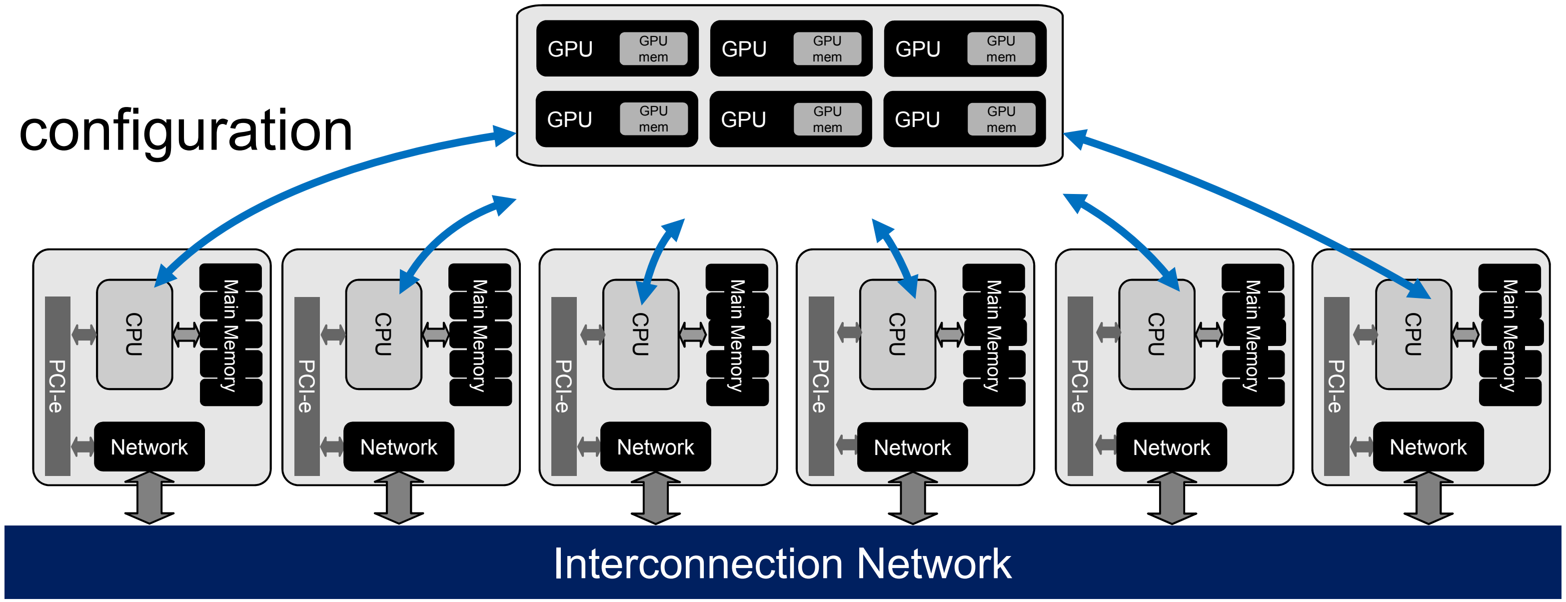
Physical configuration



Physical configuration

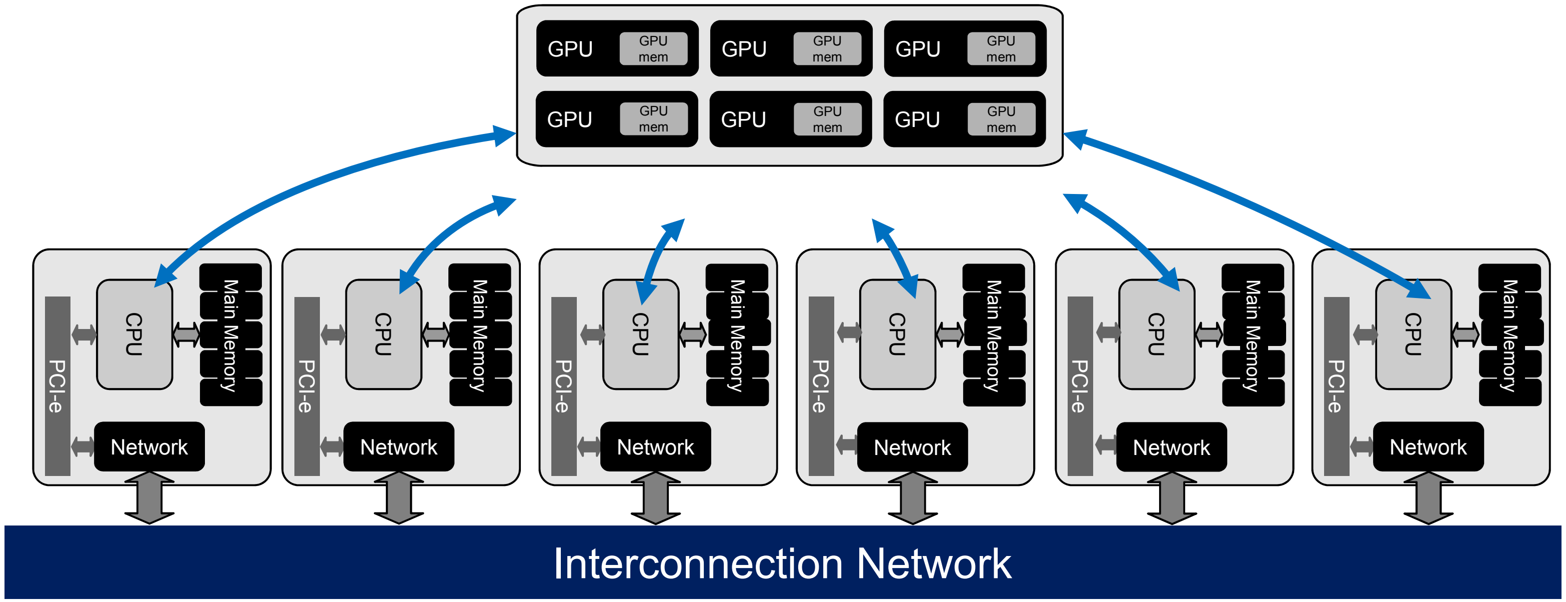


Logical configuration



Remote GPU virtualization

- All cluster nodes can use all the GPUs
- A single node can use more GPUs than it has installed
- A GPU can be shared between nodes



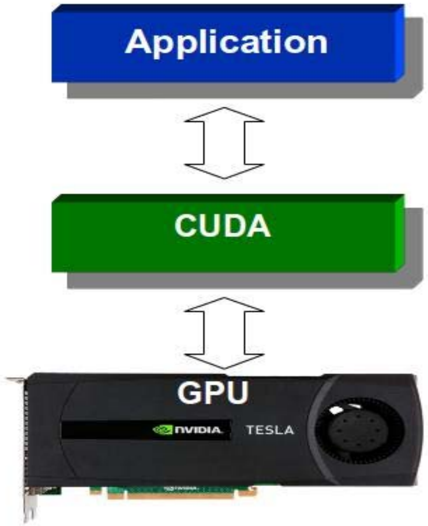
Outline

- Software
- Integration
- Systems
- Experimental Evaluation
- Conclusions and Future Work

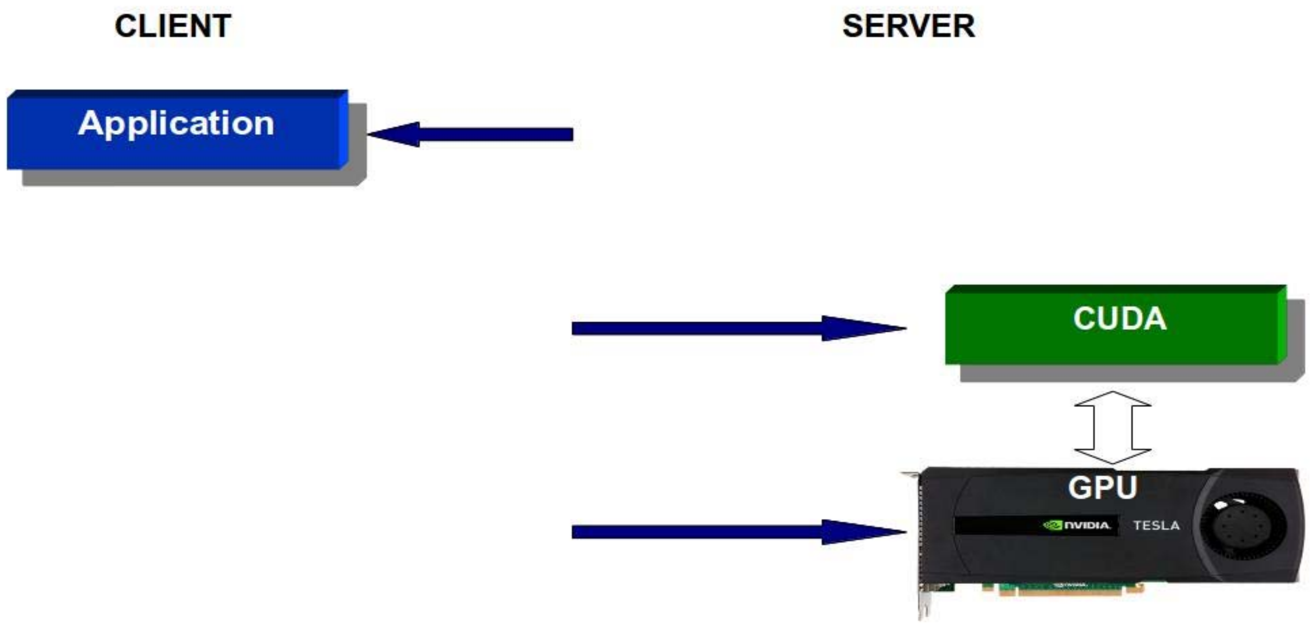
Developed in collaboration with Universidad Politécnica de Valencia
(J. Duato, C. Reaño, F. Silla)

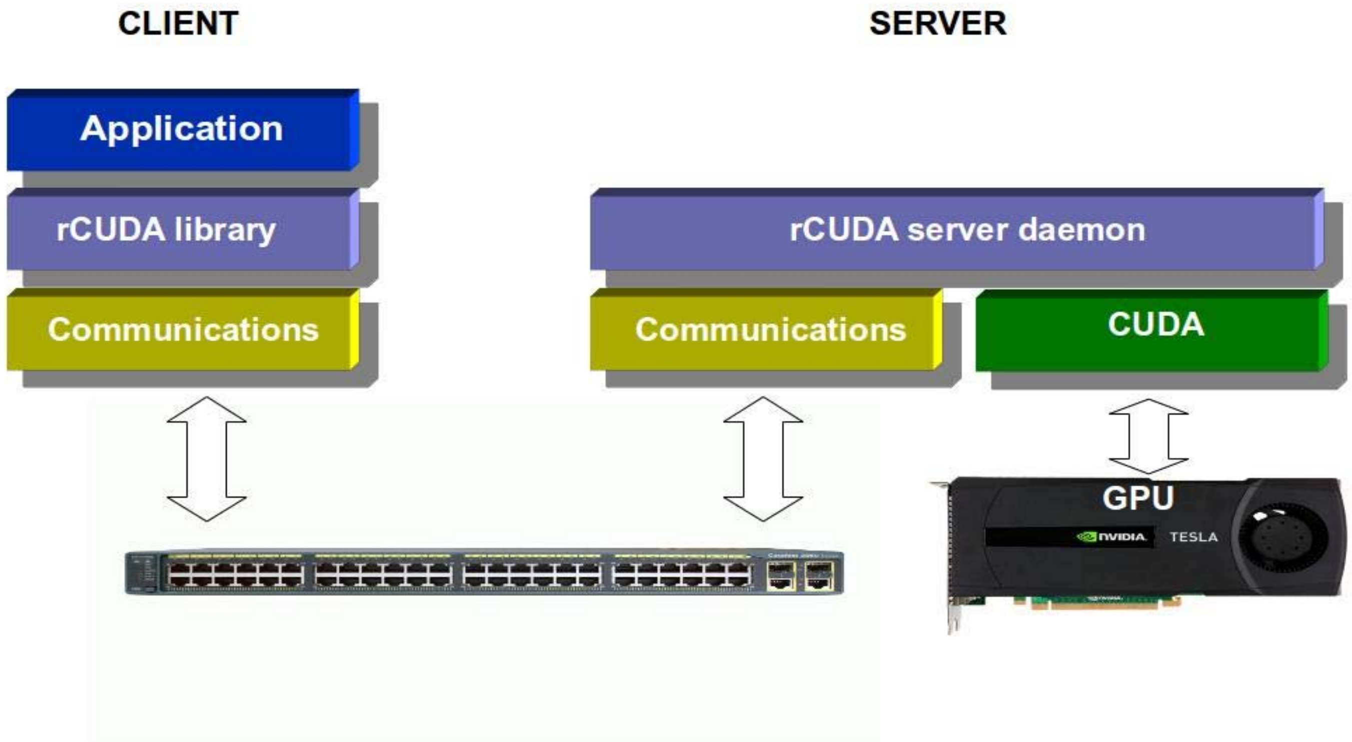
Grant a CUDA-based application running in one node access GPUs in other nodes:

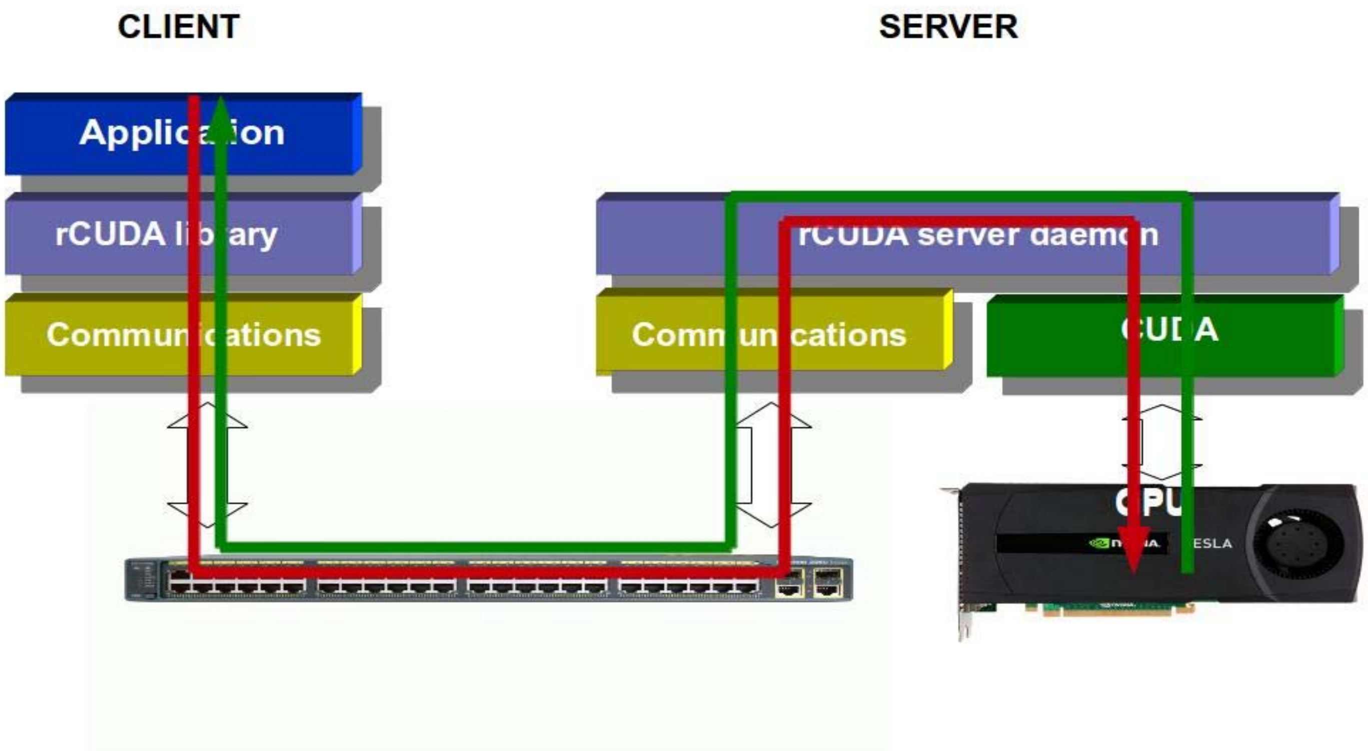
- Moderate level of data parallelism
- Applications for multi GPU computing


Application
CUDA
GPU
NVIDIA
TESLA

rCUDA remote CUDA




CLIENT
SERVER
Application
rCUDA library
Communications
rCUDA server daemon
Communications
CUDA
GPU


CLIENT
SERVER

OmpSs programming model

Developed at Barcelona Supercomputing Center

Task-oriented programming model

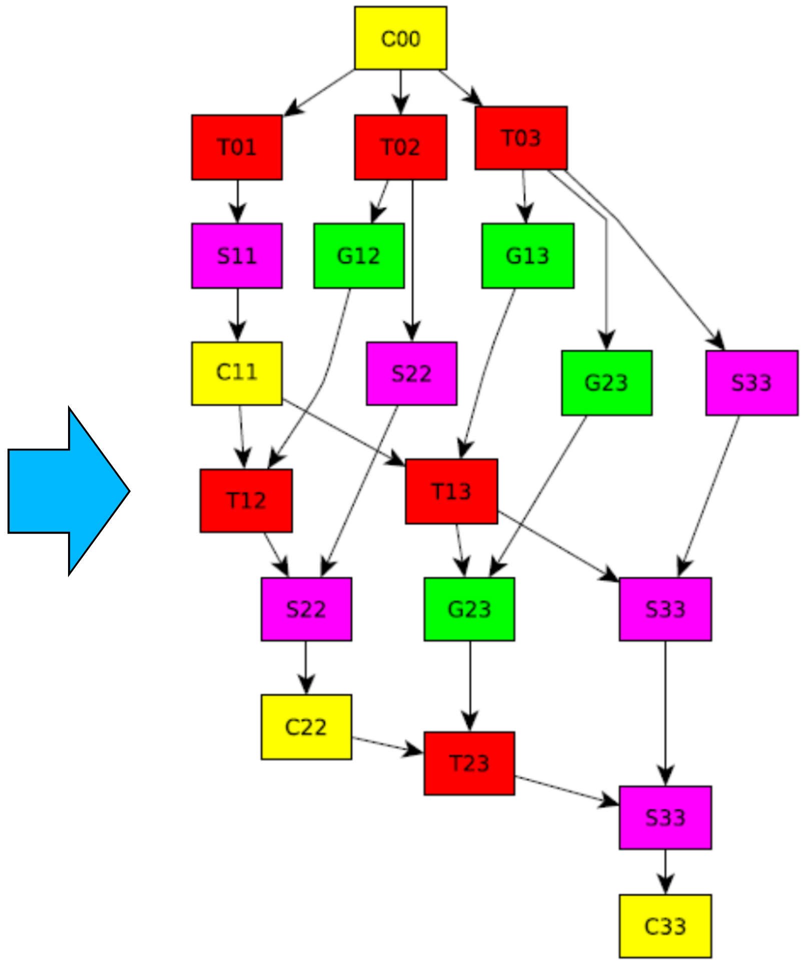
Based on OpenMP-like directives

Support from Nanos++ RT Library and Mercurium compiler

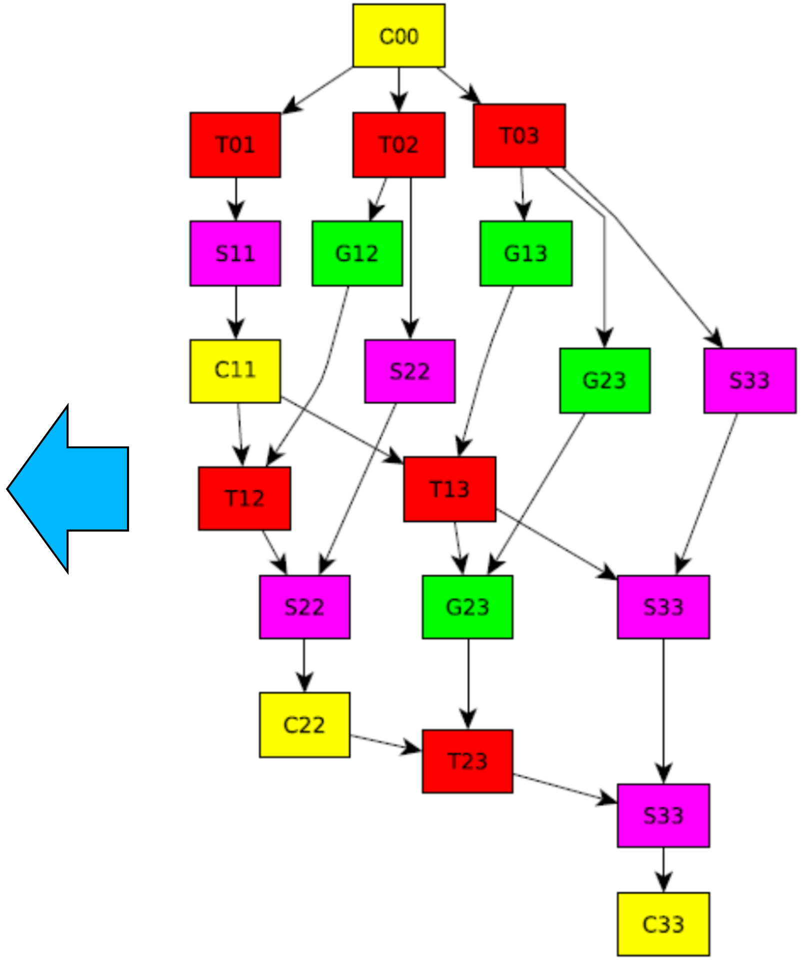
OmpSs programming model

```
void cholesky (double *A[s][s], int b, int s)
{
   for (int k = 0; k < s; k++) {

      po_cholesky (A[k][k], b, b);

      for (int j = k + 1; j < s; j++)
         tr_solve (A[k][k], A[k][j], b, b);

      for (int i = k + 1; i < s; i++) {
         for (int j = i + 1; j < s; j++)
            ge_multiply (A[k][i], A[k][j],
                         A[i][j], b, b);
         sy_update (A[k][i], A[i][i], b, b);
      }

   }
}
```

```
#pragma omp task inout([b][b]A)
void po_cholesky (double *A, int b, int ld)
{
 static int          INFO = 0;
 static const char UP    = 'U';
 dpotrf (&UP, &b, A, &ld, &INFO);   // LAPACK
}
```



OmpSs
programming model



Outline

- Introduction
- Software
- Systems
- Integration
- Experimental Evaluation
- Conclusions and Future Work

Integration

Initialization:

- CUDA loads functions/kernels upon beginning of the execution
- OmpSs loads them the first time a setup function is called
- Original rCUDA mimics CUDA → delayed load

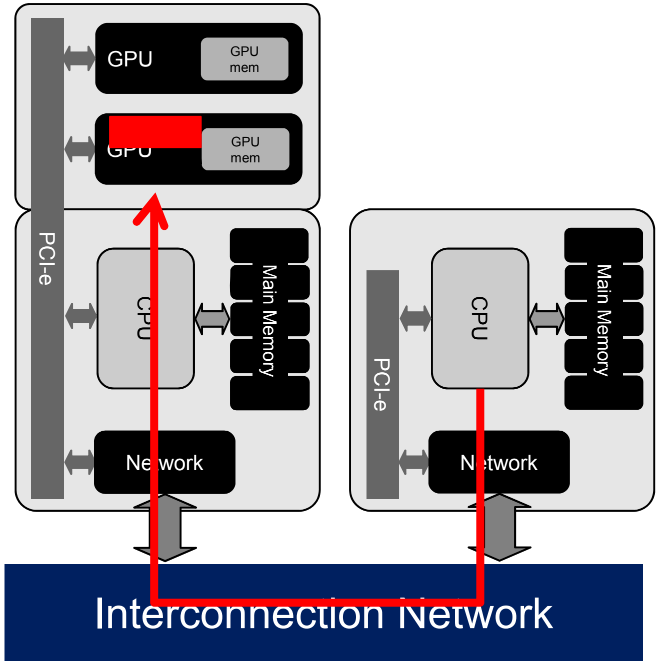
Avoid communication Overhead

- OmpSs performs regular cudaFree calls to prevent deep C-state
- rCUDA daemon maintains the GPU active
- rCUDA client does not use the network for this mechanism

On-going work

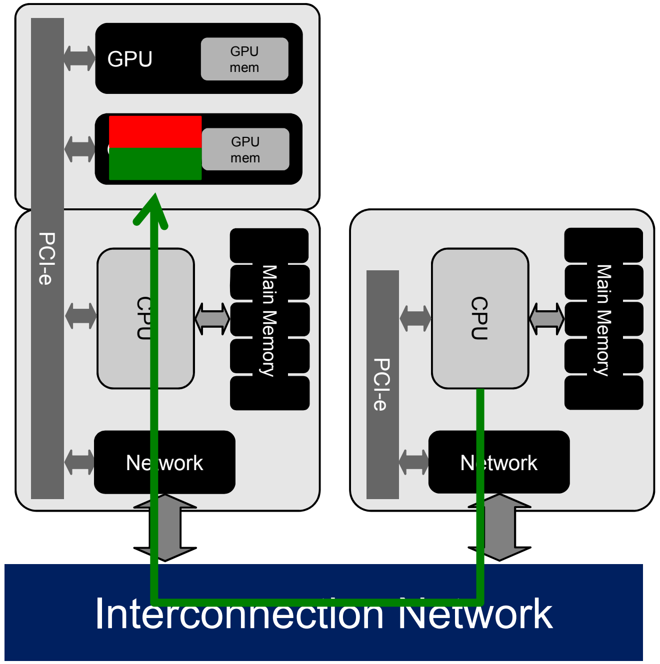
- OmpSs implements work stealing using cudaMemcpyPeer for copying data between GPU memories
- Current rCUDA does not allow cudaMemcpyPeer calls as each thread in the client side is a process in the server side

Scenarios: Intra-node memory copy I



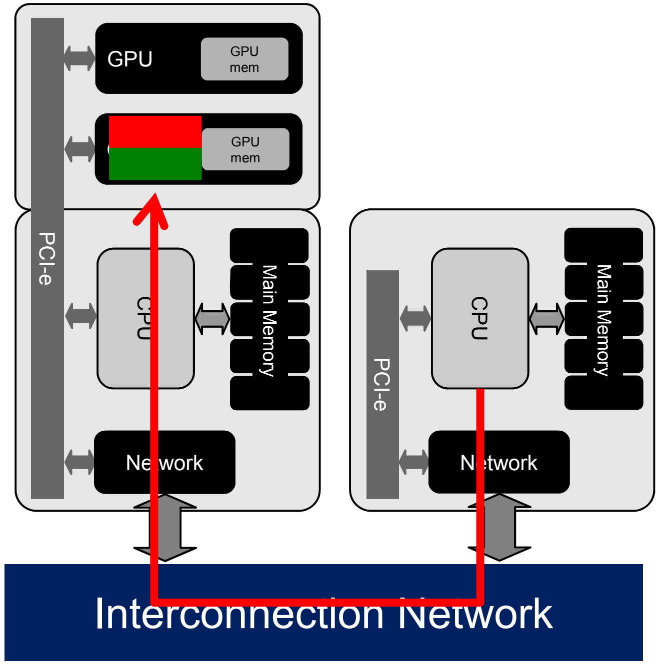
1. Thread 1 allocates GPU memory 0

Scenarios: Intra-node memory copy I



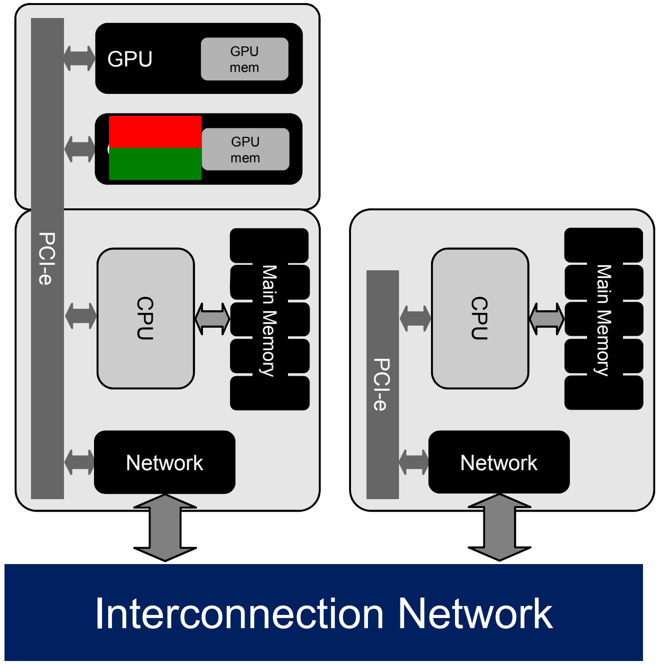
1. Thread 1 allocates GPU memory 0
2. Thread 2 allocates GPU memory 0

Scenarios: Intra-node memory copy I



1. Thread 1 allocates GPU memory 0
2. Thread 2 allocates GPU memory 0
3. Thread 1 tries to move data from thread 2 allocated memory to thread 1 allocated memory

Scenarios: Intra-node memory copy I

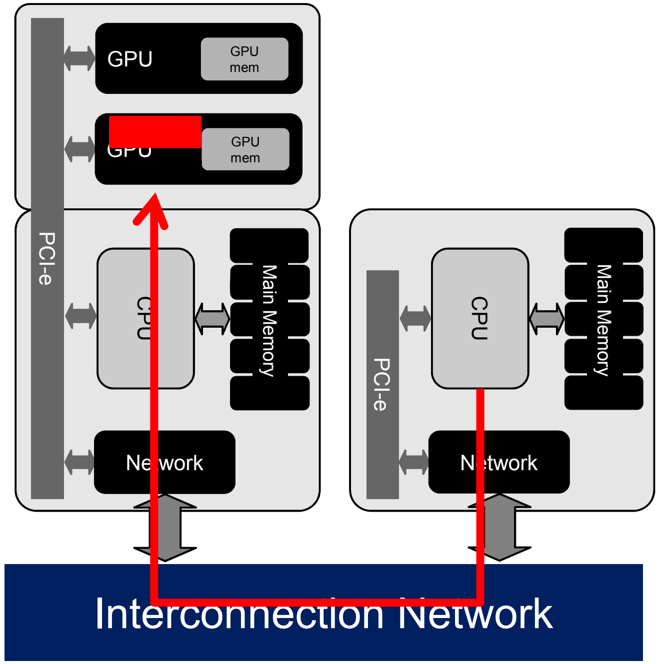


1. Thread 1 allocates GPU memory 0
2. Thread 2 allocates GPU memory 0
3. Thread 1 tries to move data from thread 2 allocated memory to thread 1 allocated memory

Not possible because memory allocated by a process cannot be accessed directly by other process!

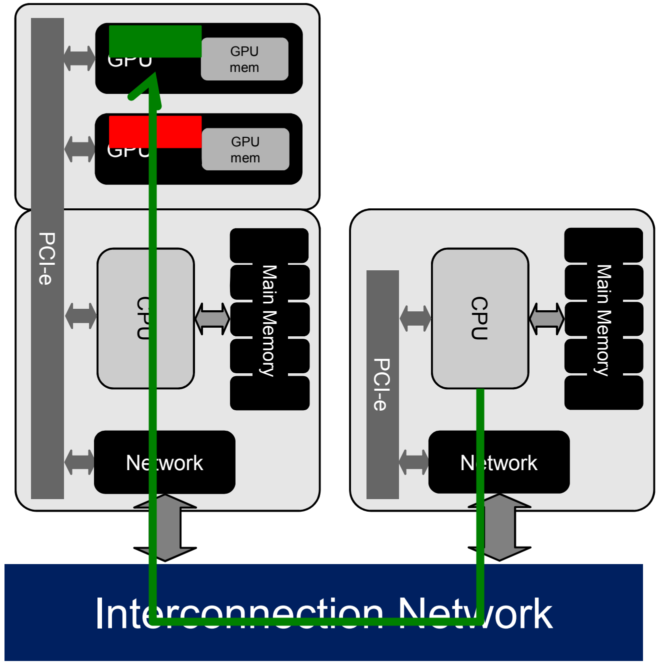
Intra-process communication is needed

Scenarios: Intra-node memory copy II



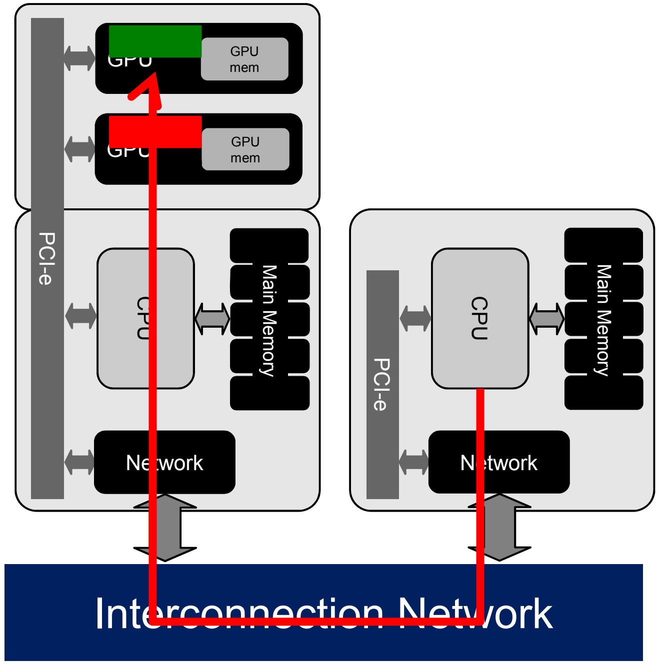
1. Thread 1 allocates GPU memory 0

Scenarios: Intra-node memory copy II



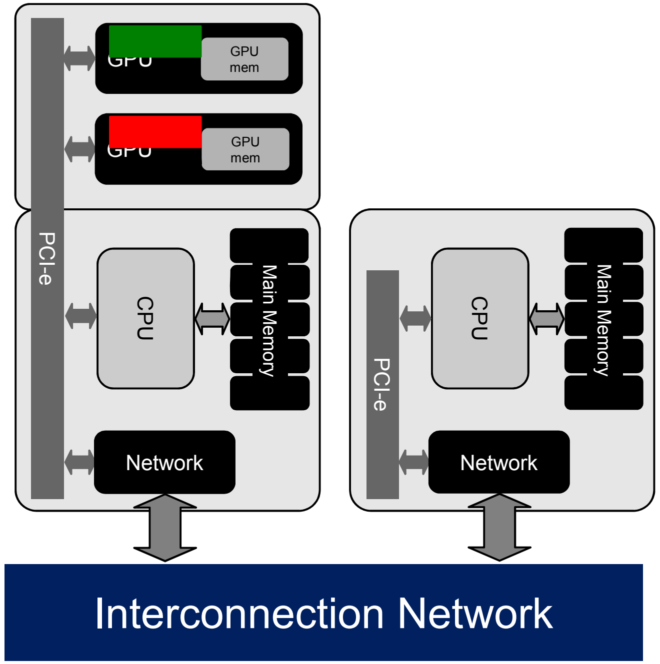
1. Thread 1 allocates GPU memory 0
2. Thread 2 allocates GPU memory 1

Scenarios: Intra-node memory copy II



1. Thread 1 allocates GPU memory 0
2. Thread 2 allocates GPU memory 1
3. Thread 1 tries to move data from thread 2 allocated memory to thread 1 allocated memory

Scenarios: Intra-node memory copy II

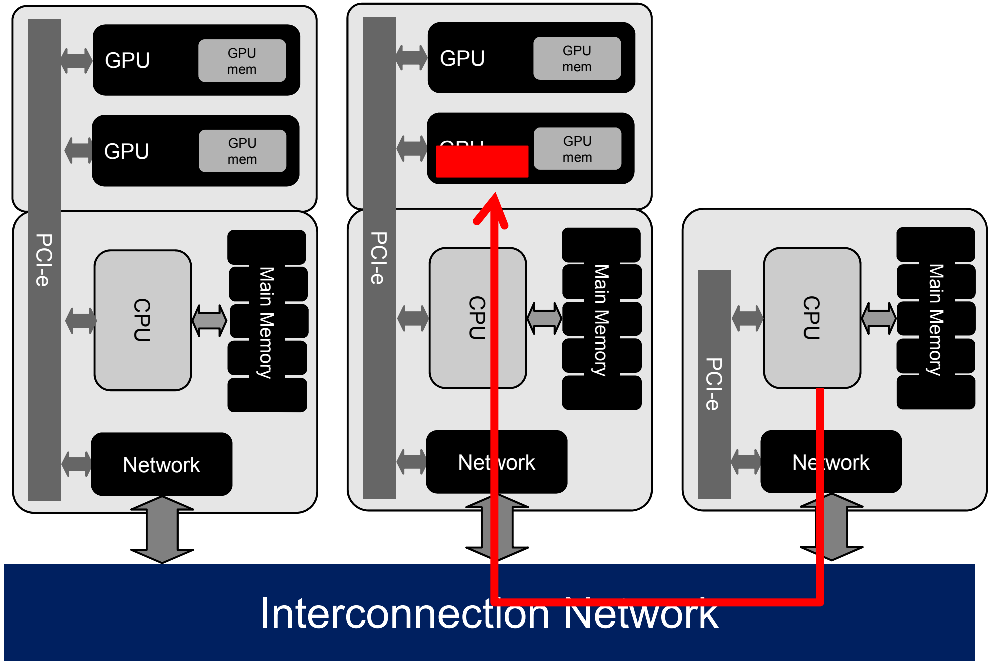


1. Thread 1 allocates GPU memory 0
2. Thread 2 allocates GPU memory 1
3. Thread 1 tries to move data from thread 2 allocated memory to thread 1 allocated memory

Not possible because memory allocated by a process cannot be accessed directly by other process!

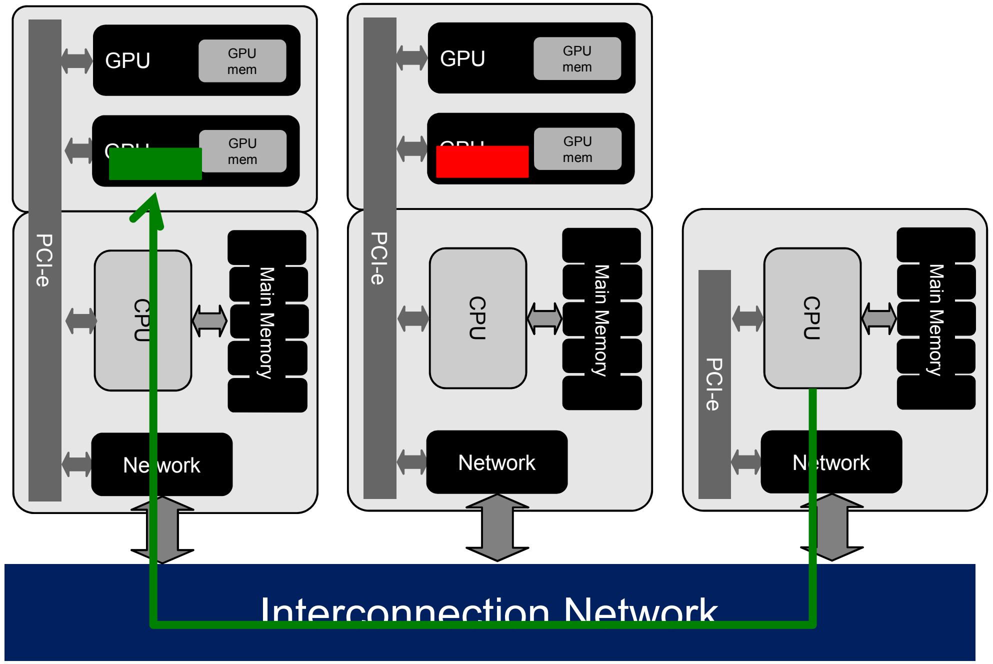
Intra-process communication is needed

Scenarios: Inter-node memory copy



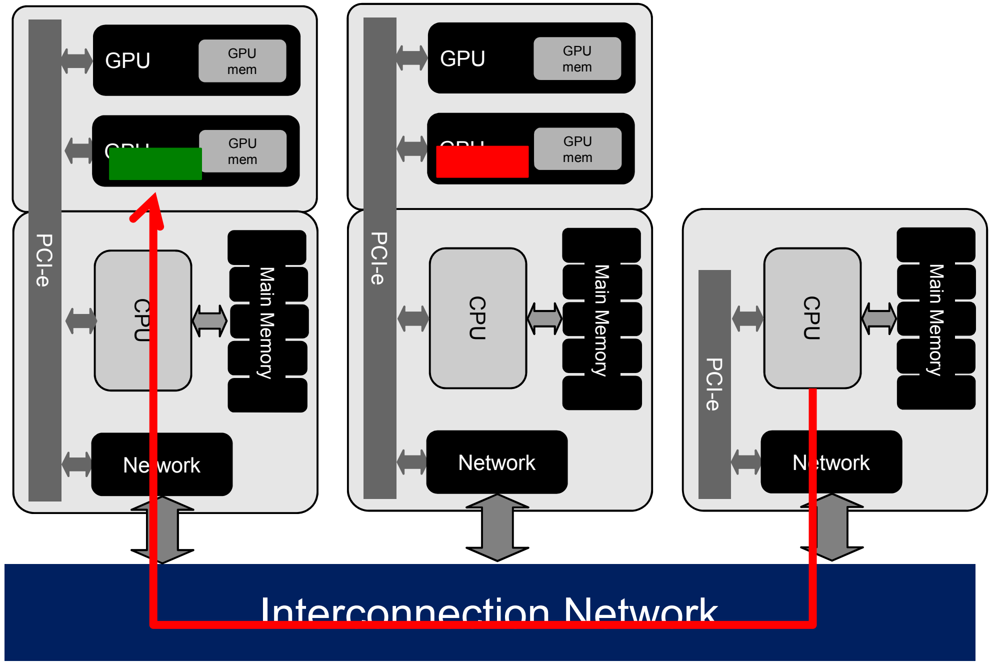
1. Thread 1 allocates GPU memory 0

Scenarios: Inter-node memory copy



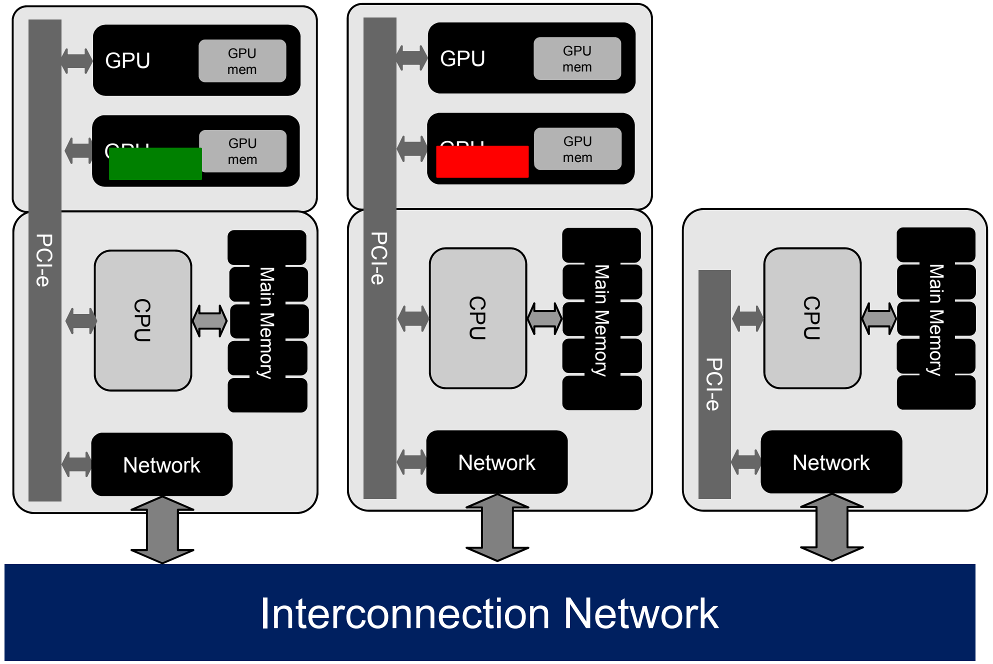
1. Thread 1 allocates GPU memory 0
2. Thread 2 allocates GPU memory 2

Scenarios: Inter-node memory copy



1. Thread 1 allocates GPU memory 0
2. Thread 2 allocates GPU memory 2
3. Thread 1 tries to move data from thread 2 allocated memory to thread 1 allocated memory

Scenarios: Inter-node memory copy



1. Thread 1 allocates GPU memory 0
2. Thread 2 allocates GPU memory 2
3. Thread 1 tries to move data from thread 2 allocated memory to thread 1 allocated memory

Not possible because memory allocated by a process cannot be accessed directly by other process!

GPU Direct RDMA on top of MPI is needed

Outline

- Introduction
- Software
- Integration
- Systems
- Experimental Evaluation
- Conclusions and Future Work

Hardware

- Tintorrum: 2-node system
  - 2 x Intel Xeon E5520 (quad-core) at 2.27 GHz
  - 24 GB of DDR3-1866 RAM memory.
  - 2 x NVIDIA C2050 boards, and 4 x NVIDIA C2050 GPUs.
  - Inter- node communications employ an InfiniBand (IB) QDR fabric.

- Minotauro: 126 nodes cluster (BSC)
  - 2 x Intel Xeon E5649 (6 cores) at 2.53 GHz
  - 24 GB od DDR3-1333 RAM
  - 2 x NVIDIA M2050 GPUs
  - Infniband QDR cluster network

Software

- rCUDA and OmpSs 14.10
- Tintorrum: CUDA 6.5 and gcc 4.4.7
- Minotauro: CUDA 5.0 and gcc 4.4.4

Applications

N-Body:

- Classical simulation of a dynamical system of particles
- Used in physics and astronomy
- Number of particles to 57,600
- No transfers between GPU memories
- Up to 4 local GPUs and up to 6 remote GPUs

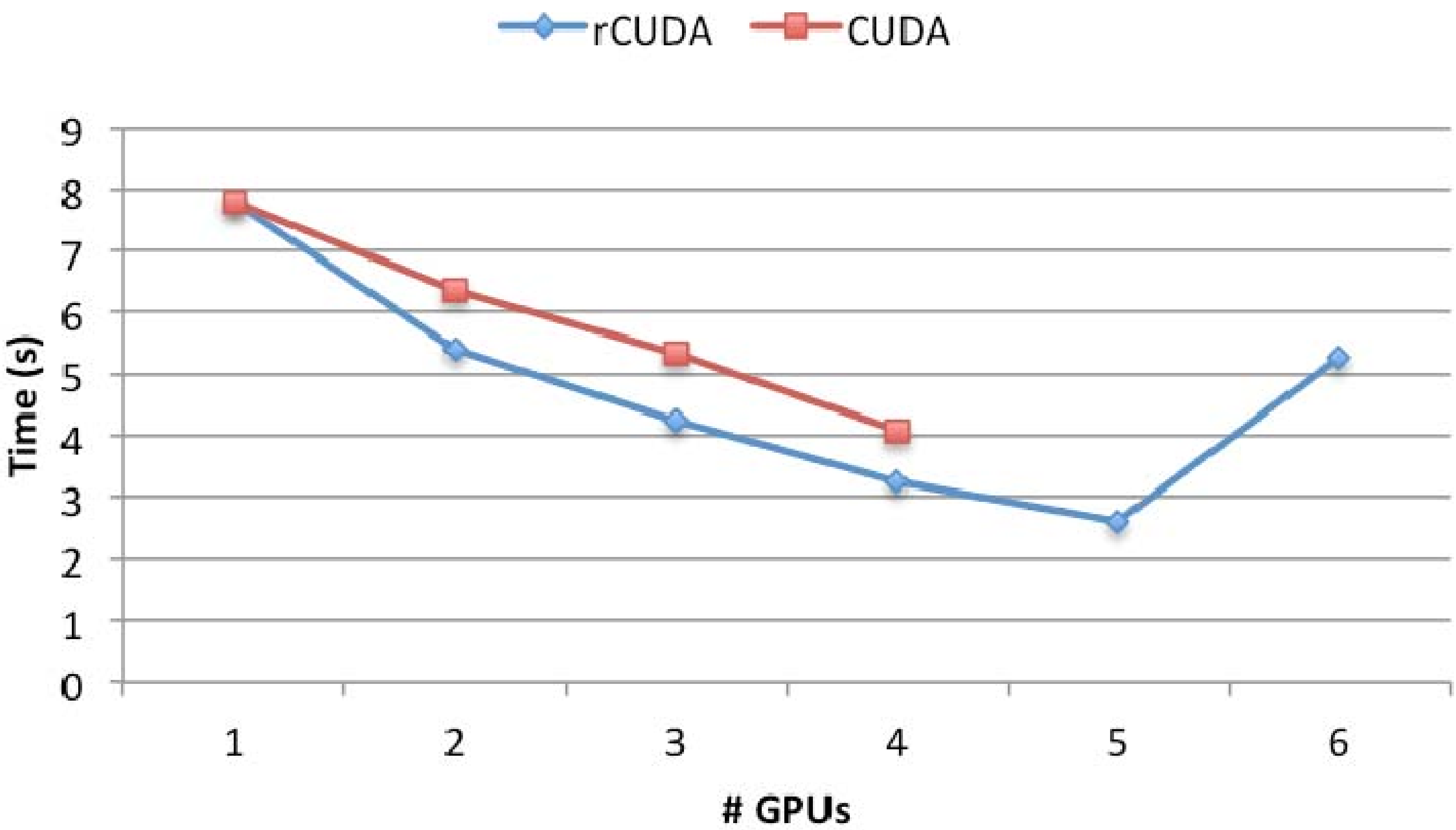
Cholesky factorization

- Solution of dense systems of linear equations
- 45,056x45,056 float elements in Minotauro
- 32,768x32,768 float elements in Tintorrum
- Up to 2 or 4 local GPUs and up to 4 remote GPUs (OmpSs cuBLAS limitation)

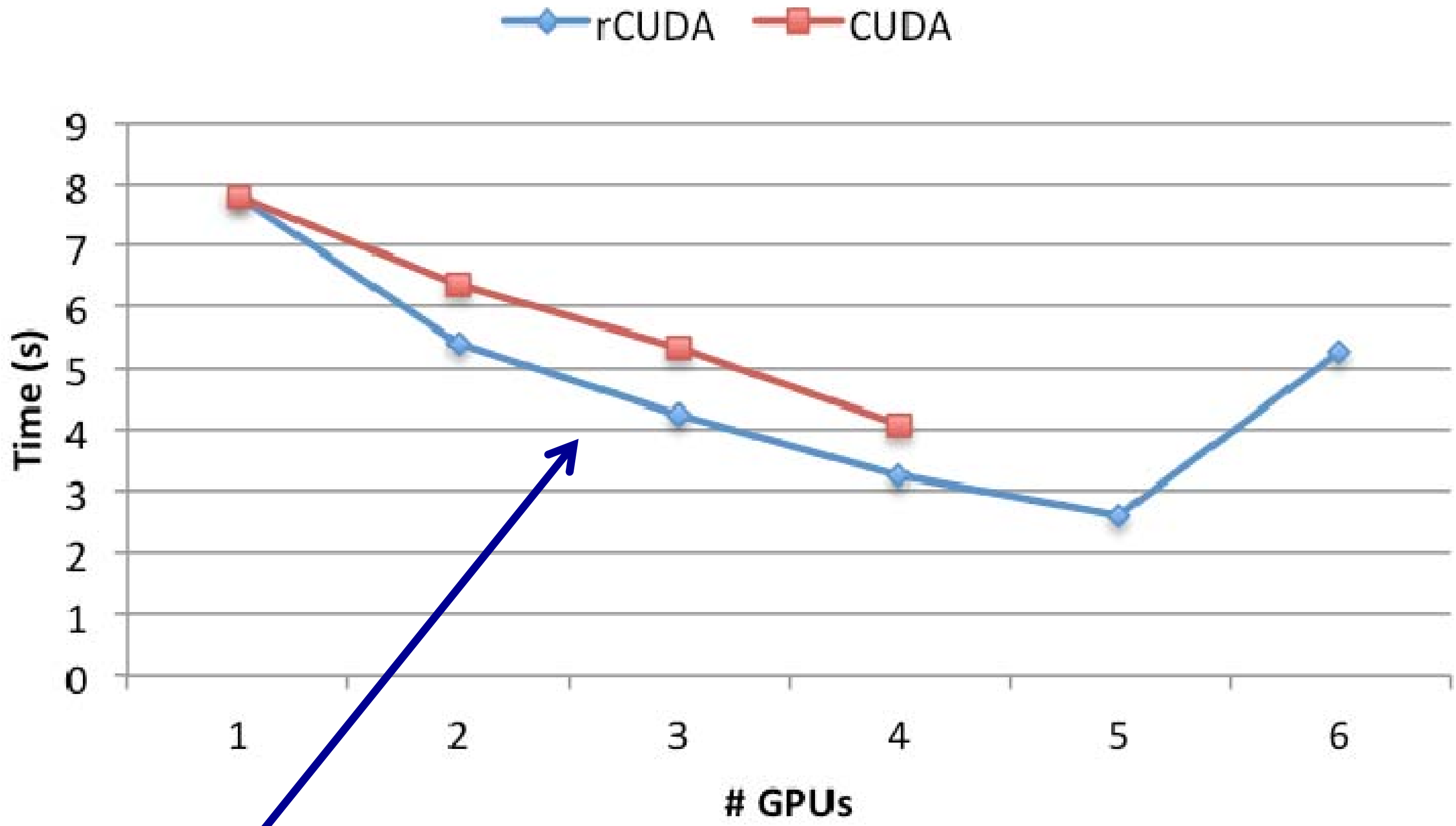
Outline

- Introduction
- Software
- Integration
- Systems
- Experimental Evaluation
- Conclusions and Future Work

N-Body (Tintorrum)

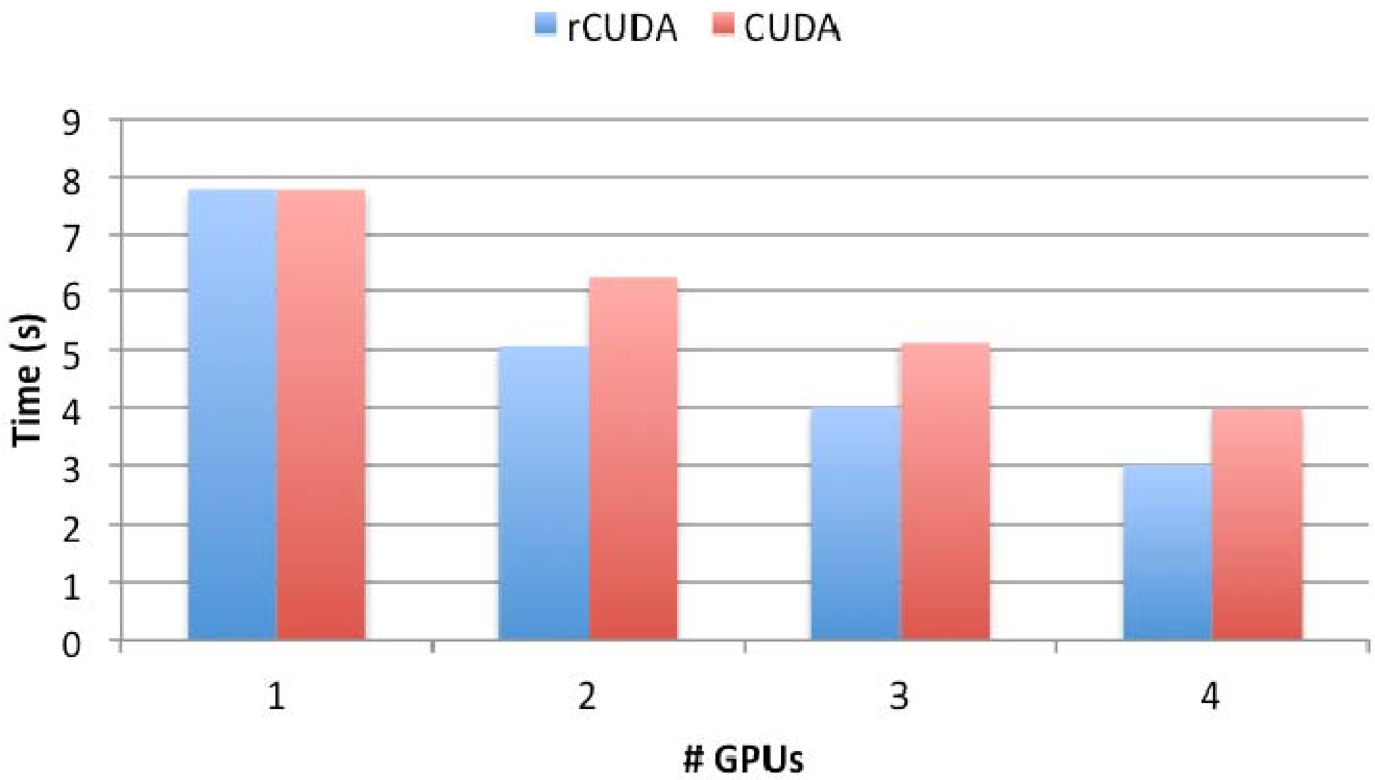


N-Body (Tintorrum)



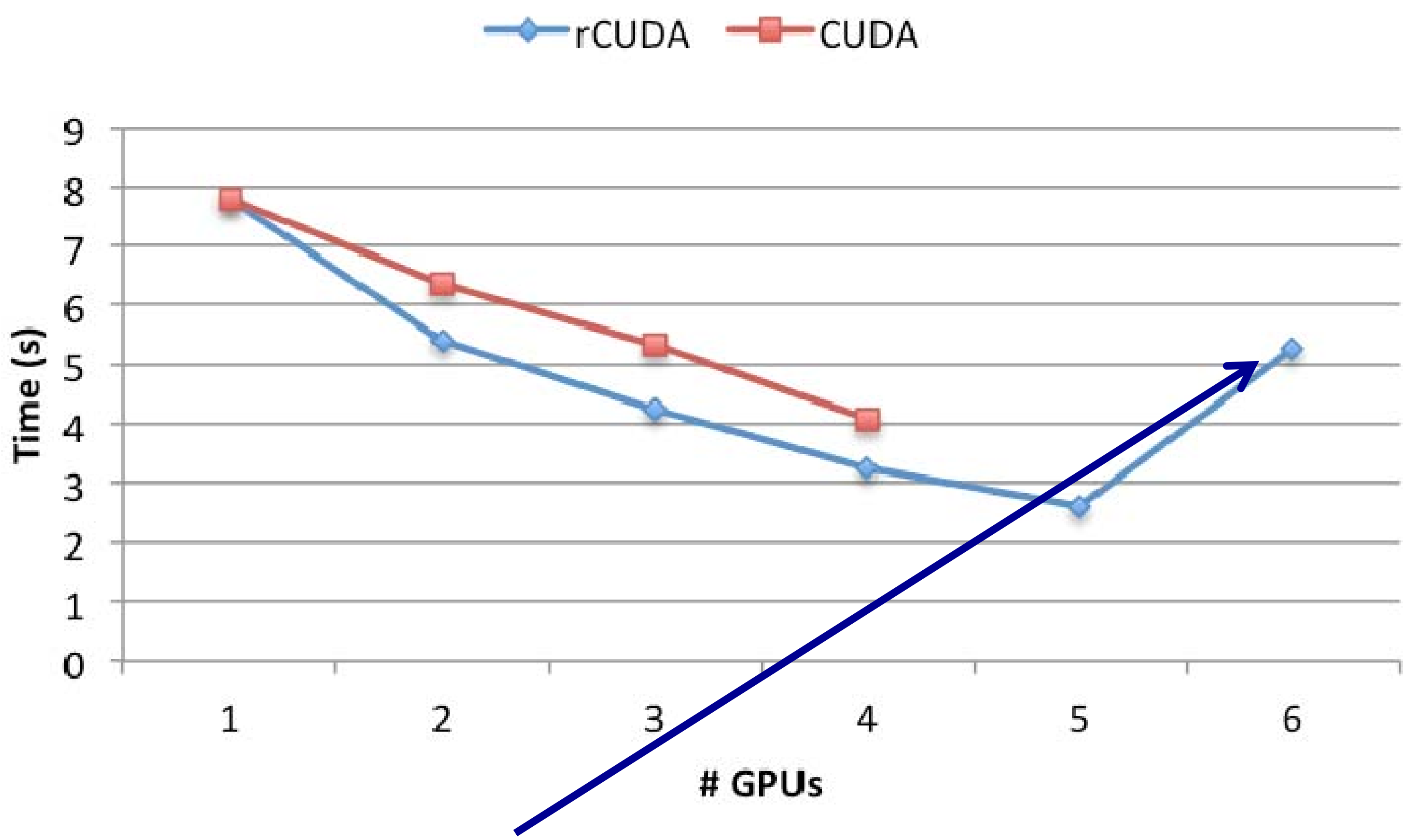
rCUDA performs better than CUDA

N-Body (Tintorrum)



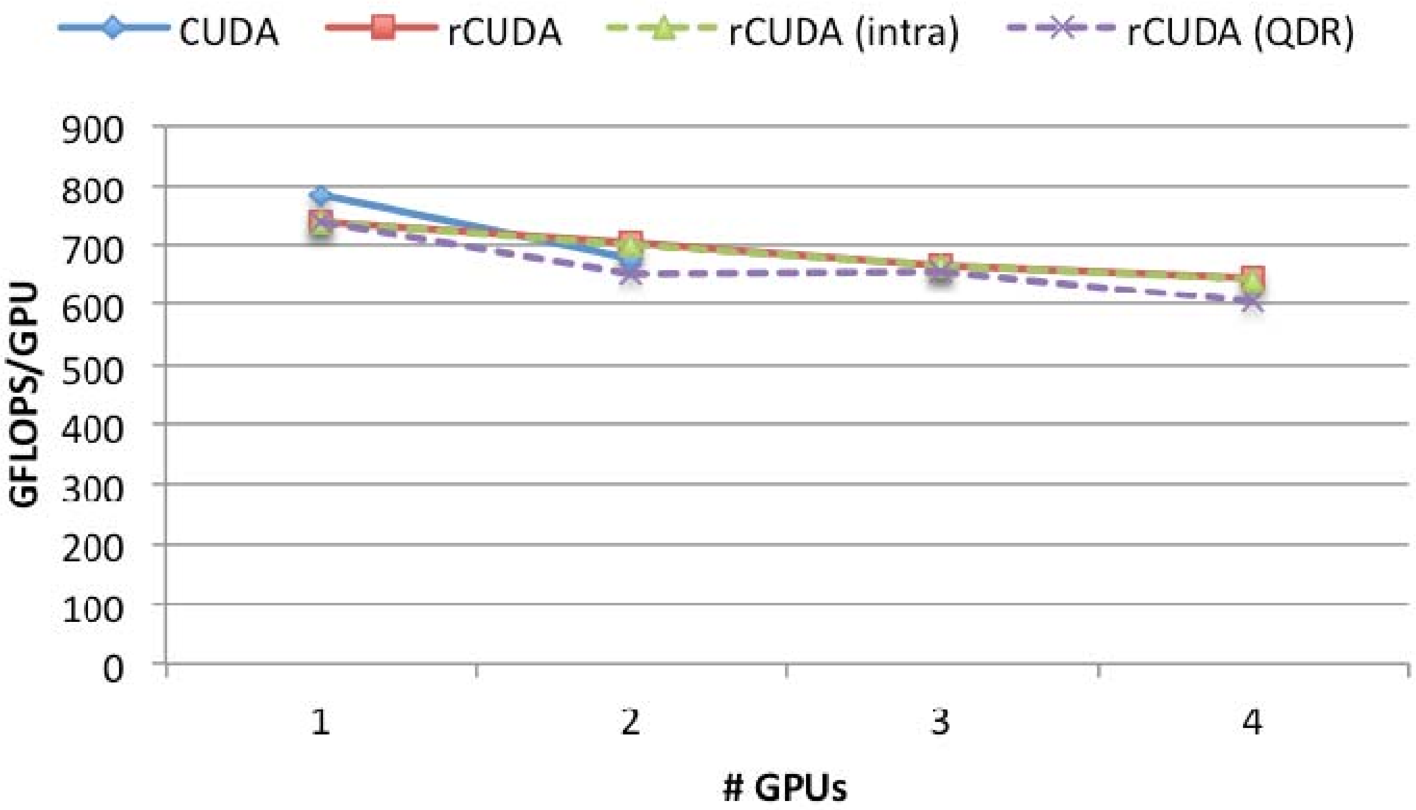
rCUDA synchronization mechanism is more agressive than that in CUDA

N-Body (Tintorrum)



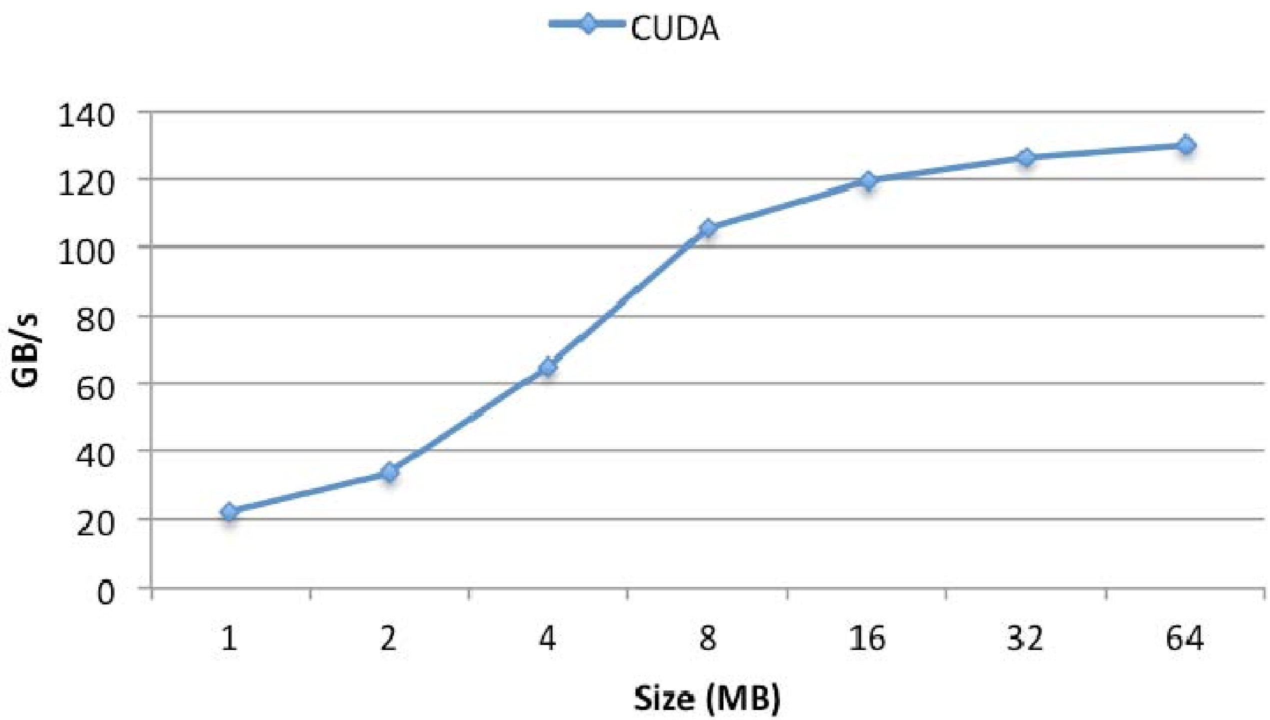
A bad hardware installation can seriously harm performance: Here, the IB card and the GPU are in different sockets and the data transfer occurs across the QPI bus

Cholesky (Minotauro)



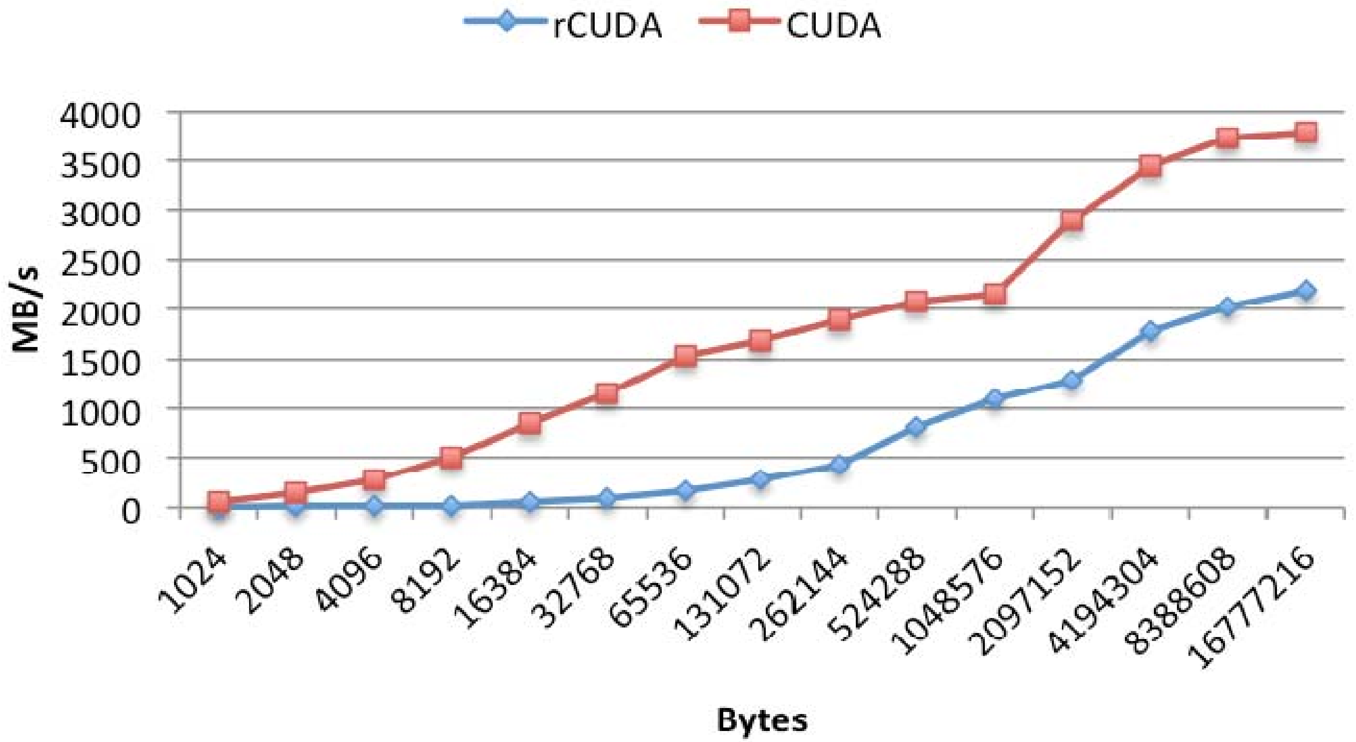
Cholesky (Minotauro)

rCUDA intra represents the rCUDA execution time and the overhead introduced by the several data transfers between GPU memories. Used to simulate the intra node time transfers

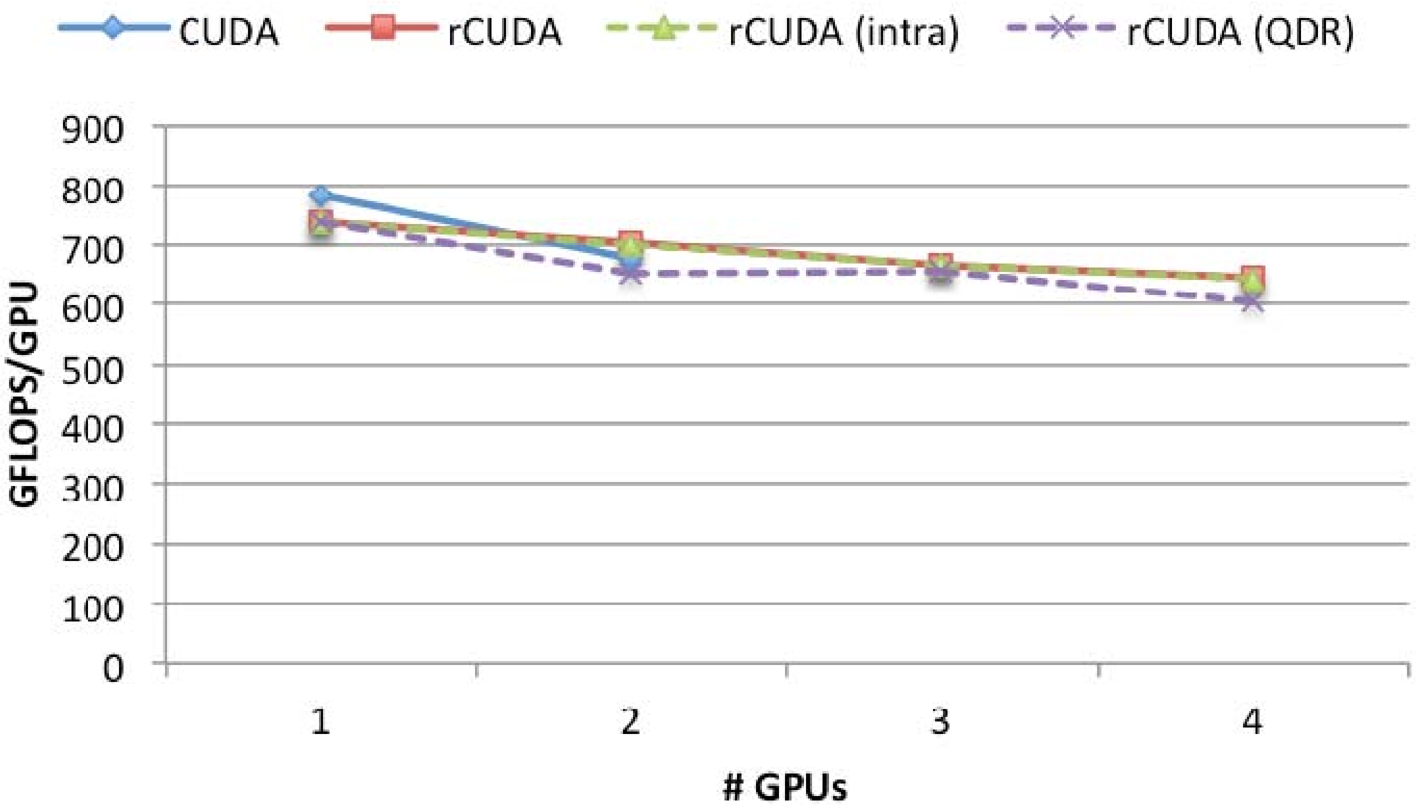


Cholesky (Minotauro)

rCUDA QDR represents the rCUDA execution time and the overhead introduced by the several data transfers between GPU memories each one in a different node using a IB QDR interconnection
Used to simulate the extra node time transfers



Cholesky (Minotauro)



rCUDA performs close to CUDA
For more than 1 GPU, only the QDR line performances worst
With a FDR interconnection it will be better

Outline

- Introduction
- Software
- Systems
- Integration
- Experimental Evaluation
- Conclusions and Future Work

Conclusions

- Combination of a virtualization framework and a task-based programming model is possible
- Most work done in rCUDA, but still far from complete
- First results of performance and scalability are promising

Conclusions

- Combination of a virtualization framework and a task-based programming model is possible
- Most work done in rCUDA, but still far from complete
- First results of performance and scalability are promising

Future work

- Implement rCUDA inter GPU memory transfers
- Analyze the scalability using CUBLAS

Thanks!