rCUDA: towards energy-efficiency in GPU computing by leveraging low-power processors and InfiniBand interconnects

Federico Silla
Technical University of Valencia
Spain

Joint research effort
Outline

- Current GPU computing facilities
- How GPU virtualization adds value to your cluster
- The rCUDA framework: basics and performance
- Integrating rCUDA with SLURM
- rCUDA and low-power processors
Outline

- Current GPU computing facilities
- How GPU virtualization adds value to your cluster
- The rCUDA framework: basics and performance
- Integrating rCUDA with SLURM
- rCUDA and low-power processors
GPU computing defines all the technological issues for using the GPU computational power for executing general purpose applications.

GPU computing has experienced a remarkable growth in the last years.
Current GPU computing facilities

The basic building block is a node with one or more GPUs.

GPUs inside the CPU box

GPUs outside the CPU box

Current GPU computing facilities
Current GPU computing facilities

- From the programming point of view:
  - A set of nodes, each one with:
    - one or more CPUs (probably with several cores per CPU)
    - one or more GPUs (typically between 1 and 4)
    - disjoint memory spaces for each GPU and the CPUs
  - An interconnection network
For some kinds of code the use of GPUs brings huge benefits in terms of performance and energy.

- There must be data parallelism in the code: this is the only way to take benefit from the hundreds of processors within a GPU.

Different scenarios from the point of view of the application:

- Low level of data parallelism
- High level of data parallelism
- Moderate level of data parallelism
- Applications for multi-GPU computing
  - Leveraging up to the 4 GPUs inside the node.
Outline

- Current GPU computing facilities
- How GPU virtualization adds value to your cluster
- The rCUDA framework: basics and performance
- Integrating rCUDA with SLURM
- rCUDA and low-power processors
A GPU computing facility is usually a set of independent self-contained nodes that leverage the shared-nothing approach:

- Nothing is directly shared among nodes (MPI required for aggregating computing resources within the cluster)
- GPUs can only be used within the node they are attached to

How GPU virtualization adds value
CONCERNS:

- Multi-GPU applications running on a subset of nodes cannot make use of the tremendous GPU resources available at other cluster nodes (even if they are idle)
- For applications with moderate levels of data parallelism, many GPUs in the cluster may be idle for long periods of time
How GPU virtualization adds value

- A way of addressing the first concern is by **sharing** the GPUs present in the cluster among all the nodes. For the second concern, once GPUs are shared, their amount can be **reduced**.

- This would increase GPU utilization, also lowering power consumption, at the same time that initial acquisition costs are reduced.

- This new configuration requires:
  1) A way of seamlessly sharing GPUs across nodes in the cluster (GPU virtualization)
  2) Enhanced job schedulers that take into account the new GPU configuration
In summary, **GPU virtualization allows a new vision** of a GPU deployment, moving from the usual cluster configuration:

![Diagram showing GPU virtualization](image)

Interconnection Network

to the following one ....
How GPU virtualization adds value

Physical configuration

Interconnection Network

Logical configuration

Logical connections

Interconnection Network
How GPU virtualization adds value

- GPU virtualization is also useful for multi-GPU applications

Without GPU virtualization

Only the GPUs in the node can be provided to the application

With GPU virtualization

Many GPUs in the cluster can be provided to the application
How GPU virtualization adds value

- GPU virtualization allows all nodes to access all GPUs

**Physical configuration**

- Node
- Node
- Node
- Node

**Real local GPUs**

**Logical configuration**

- Node
- Node
- Node
- Node

**Virtualized remote GPUs**
How GPU virtualization adds value

- One step further:
  - enhancing the scheduling process so that GPU servers are put into low-power sleeping modes as soon as their acceleration features are not required
How GPU virtualization adds value

- Going even beyond:
  - consolidating GPUs into **dedicated GPU boxes** (no CPU power required)
  - allowing GPU task migration ???

**TRUE GREEN GPU COMPUTING**
Main GPU virtualization drawback is the increased latency and reduced bandwidth to the remote GPU.

Influence of data transfers for SGEMM

- Pinned Memory
- Non-Pinned Memory

Data from a matrix-matrix multiplication using a local GPU!!!
Several efforts have been made regarding GPU virtualization during the last years:

- **rCUDA** (CUDA 5.0)
- **GVirtuS** (CUDA 3.2)
- **DS-CUDA** (CUDA 4.1)
- **vCUDA** (CUDA 1.1)
- **GViM** (CUDA 1.1)
- **GridCUDA** (CUDA 2.3)
- **V-GPU** (CUDA 4.0)

**Publicly available**

**NOT publicly available**
How GPU virtualization adds value

- **Performance comparison** of GPU virtualization solutions:
  - Intel Xeon E5-2620 (6 cores) 2.0 GHz (SandyBridge architecture)
  - GPU NVIDIA Tesla K20
  - Mellanox ConnectX-3 single-port InfiniBand Adapter (FDR)
  - SX6025 Mellanox switch
  - CentOS 6.3 + Mellanox OFED 1.5.3

<table>
<thead>
<tr>
<th>Latency (measured by transferring 64 bytes)</th>
<th>Pageable H2D</th>
<th>Pinned H2D</th>
<th>Pageable D2H</th>
<th>Pinned D2H</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA</td>
<td>34,3</td>
<td>4,3</td>
<td>16,2</td>
<td>5,2</td>
</tr>
<tr>
<td>rCUDA</td>
<td>94,5</td>
<td>23,1</td>
<td>292,2</td>
<td>6,0</td>
</tr>
<tr>
<td>GVirtuS</td>
<td>184,2</td>
<td>200,3</td>
<td>168,4</td>
<td>182,8</td>
</tr>
<tr>
<td>DS-CUDA</td>
<td>45,9</td>
<td>-</td>
<td>26,5</td>
<td>-</td>
</tr>
</tbody>
</table>
How GPU virtualization adds value

- Bandwidth of a copy between GPU and CPU memories

### Host-to-Device
- pageable memory

- pinned memory

### Device-to-Host
- pageable memory

- pinned memory

**Graphs:**
- **Host-to-Device**
  - CUDA
  - rCUDA
  - GVirtuS
  - DS-CUDA

- **Device-to-Host**
  - CUDA
  - rCUDA
  - GVirtuS
  - DS-CUDA
How GPU virtualization adds value

- rCUDA has been successfully tested with the following applications:
  - NVIDIA CUDA SDK Samples
  - LAMMPS
  - WideLM
  - CUDASW++
  - OpenFOAM
  - HOOMDBlue
  - mCUDA-MEME
  - GPU-Blast
  - Gromacs
Outline

- Current GPU computing facilities
- How GPU virtualization adds value to your cluster
- The rCUDA framework: basics and performance
- Integrating rCUDA with SLURM
- rCUDA and low-power processors
A framework enabling a CUDA-based application running in one (or some) node(s) to access GPUs in other nodes

It is useful for:

- Moderate level of data parallelism
- Applications for multi-GPU computing
Basics of the rCUDA framework

CUDA application

Application

CUDA libraries

CUDA device
Basics of the rCUDA framework
Basics of the rCUDA framework

Client side
- Application
- rCUDA library
- Network device

CUDA application
- rCUDA daemon
- Network device

Server side
- CUDA libraries
- CUDA device
Basics of the rCUDA framework

Client side

CUDA application

Server side

Application

rCUDA library

Network device

rCUDA daemon

CUDA libraries

Network device

CUDA device
Basics of the rCUDA framework

Client side

- Application
- rCUDA library

CUDA application

- rCUDA daemon

Server side

- CUDA libraries
- CUDA device
- Network device
- Network device
rCUDA uses a proprietary communication protocol

Example:

1) initialization
2) memory allocation on the remote GPU
3) CPU to GPU memory transfer of the input data
4) kernel execution
5) GPU to CPU memory transfer of the results
6) GPU memory release
7) communication channel closing and server process finalization
Basics of the rCUDA framework

- rCUDA features a modular architecture
Basics of the rCUDA framework

- rCUDA features **optimized communications:**
  - Use of GPUDirect to avoid unnecessary memory copies
  - Pipeline to improve performance
  - Preallocated pinned memory buffers
  - Optimal pipeline block size
Basics of the rCUDA framework

- Pipeline block size for InfiniBand FDR

- NVIDIA Tesla K20; Mellanox ConnectX-3 + SX6025 Mellanox switch

It was 2MB with IB QDR
Basics of the rCUDA framework

- Bandwidth of a copy between GPU and CPU memories

```
Throughput (MB/s)

Copy Size (MB)

CUDA
rCUDA FDR
rCUDA QDR
rCUDA GbE
```

Host-to-Device pinned memory

Host-to-Device pageable memory
Basics of the rCUDA framework

- Latency study: copy a small dataset (64 bytes)
- Current GPU computing facilities
- How GPU virtualization adds value to your cluster
- The *rCUDA* framework: basics and performance
- Integrating *rCUDA* with SLURM
- *rCUDA* and low-power processors
Performance of the rCUDA framework

- Test system:
  - Intel Xeon E5-2620 (6 cores) 2.0 GHz (SandyBridge architecture)
  - GPU NVIDIA Tesla K20
  - Mellanox ConnectX-3 single-port InfiniBand Adapter (FDR)
  - SX6025 Mellanox switch
  - Cisco switch SLM2014 (1Gbps Ethernet)
  - CentOS 6.3 + Mellanox OFED 1.5.3
Performance of the rCUDA framework

- Test CUDA SDK examples
Performance of the rCUDA framework

- CUDASW++

Bioinformatics software for Smith-Waterman protein database searches

![Graph showing performance of rCUDA framework]
GPU-Blast

Accelerated version of the NCBI-BLAST (Basic Local Alignment Search Tool), a widely used bioinformatics tool
Test system for MultiGPU

- For CUDA tests one node with:
  - 2 Quad-Core Intel Xeon E5440 processors
  - Tesla S2050 (4 Tesla GPUs)
  - Each thread (1-4) uses one GPU

- For rCUDA tests 8 nodes with:
  - 2 Quad-Core Intel Xeon E5520 processors
  - 1 NVIDIA Tesla C2050
  - InfiniBand QDR
  - Test running in one node and using up to all the GPUs of the others
  - Each thread (1-8) uses one GPU
Performance of the rCUDA framework

- MonteCarlo MultiGPU (from NVIDIA SDK)
Performance of the rCUDA framework

- GEMM MultiGPU
  - using **libflame**: high performance dense linear algebra library
    http://www.cs.utexas.edu/~flame/web/

![Bar chart showing performance comparison between CUDA and rCUDA](chart.png)
Outline

- Current GPU computing facilities
- How GPU virtualization adds value to your cluster
- The rCUDA framework: basics and performance
- Integrating rCUDA with SLURM
- rCUDA and low-power processors
Integrating rCUDA with SLURM

- SLURM (Simple Linux Utility for Resource Management) job scheduler
- Does not understand about virtualized GPUs
- Add a new GRES (general resource) in order to manage virtualized GPUs
- Where the GPUs are in the system is completely transparent to the user
- In the job script or in the submission command, the user specifies the number of rGPUs (remote GPUs) required by the job
Integrating rCUDA with SLURM
Integrating rCUDA with SLURM

**slurm.conf**

```
ClusterName=rcu
...
GresTypes=gpu,rgpu
...
NodeName=rcu16 NodeAddr=rcu16 CPUs=8 Sockets=1
    CoresPerSocket=4 ThreadsPerCore=2
RealMemory=7990 Gres=rgpu:4,gpu:4
```
### Integrating rCUDA with SLURM

```
scontrol show node
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NodeName</td>
<td>rcu16</td>
</tr>
<tr>
<td>Arch</td>
<td>x86_64</td>
</tr>
<tr>
<td>CoresPerSocket</td>
<td>4</td>
</tr>
<tr>
<td>CPUAlloc</td>
<td>0</td>
</tr>
<tr>
<td>CPUErr</td>
<td>0</td>
</tr>
<tr>
<td>CPUTot</td>
<td>8</td>
</tr>
<tr>
<td>Features</td>
<td>(null)</td>
</tr>
<tr>
<td>Gres</td>
<td>rgpu:4, gpu:4</td>
</tr>
<tr>
<td>NodeAddr</td>
<td>rcu16</td>
</tr>
<tr>
<td>NodeHostName</td>
<td>rcu16</td>
</tr>
<tr>
<td>OS</td>
<td>Linux</td>
</tr>
<tr>
<td>RealMemory</td>
<td>7682</td>
</tr>
<tr>
<td>Sockets</td>
<td>1</td>
</tr>
<tr>
<td>State</td>
<td>IDLE</td>
</tr>
<tr>
<td>ThreadsPerCore</td>
<td>2</td>
</tr>
<tr>
<td>TmpDisk</td>
<td>0</td>
</tr>
<tr>
<td>Weight</td>
<td>1</td>
</tr>
<tr>
<td>BootTime</td>
<td>2013-04-24T18:45:35</td>
</tr>
<tr>
<td>SlurmdStartTime</td>
<td>2013-04-30T10:02:04</td>
</tr>
</tbody>
</table>
Integrating rCUDA with SLURM

gres.conf

<table>
<thead>
<tr>
<th>Name</th>
<th>File</th>
<th>Cuda</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>rgpu</td>
<td>/dev/nvidia0</td>
<td>2.1</td>
<td>1535m</td>
</tr>
<tr>
<td>rgpu</td>
<td>/dev/nvidia1</td>
<td>3.5</td>
<td>1535m</td>
</tr>
<tr>
<td>rgpu</td>
<td>/dev/nvidia2</td>
<td>1.2</td>
<td>1535m</td>
</tr>
<tr>
<td>rgpu</td>
<td>/dev/nvidia3</td>
<td>1.2</td>
<td>1535m</td>
</tr>
<tr>
<td>gpu</td>
<td>/dev/nvidia0</td>
<td>1.2</td>
<td>1535m</td>
</tr>
<tr>
<td>gpu</td>
<td>/dev/nvidia1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gpu</td>
<td>/dev/nvidia2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gpu</td>
<td>/dev/nvidia3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Integrating rCUDA with SLURM

Submit a job

```
$ srun -N1 --gres=rgpu:4:512M script.sh...
```

Environment variables are initialized by SLURM and used by the rCUDA client (transparently to the user)

- `RCUDA_DEVICE_COUNT=4`
- `RCUDA_DEVICE_0=rcu16:0`
- `RCUDA_DEVICE_1=rcu16:1`
- `RCUDA_DEVICE_2=rcu16:2`
- `RCUDA_DEVICE_3=rcu16:3`

Server name/IP address: GPU
Integrating rCUDA with SLURM

Resources per job:
- 1: Nodes: 2 GPUs: 3
- 2: Nodes: 2 GPUs: 1
- 3: Nodes: 1 GPUs: 0
- 4: Nodes: 2 GPUs: 1
- 5: Nodes: 1 GPUs: 2

Timeline:
- t0: NODE 0
  - GPU 0
  - GPU 1
- t1: NODE 1
  - GPU 0
- t2: NODE 2
  - Job 3
- t3: NODE 3
  - Job queue

Job queue:
1 2 3 4 5 5

srn: error: Unable to allocate resources
Integrating rCUDA with SLURM

<table>
<thead>
<tr>
<th>Resources per job</th>
<th>Nodes: 2 GPUs: 3</th>
<th>Nodes: 1 GPUs: 0</th>
<th>Nodes: 2 GPUs: 1</th>
<th>Nodes: 1 GPUs: 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

GPUs are decoupled from nodes

All jobs are executed in less time
Test bench for the SLURM experiments:

Old “heterogeneous” cluster with 28 nodes:

- 1 node without GPU, with 2 AMD Opteron 1.4GHz, 6GB RAM
- 23 nodes without GPU with 2 AMD Opteron 1.4GHz, 2GB RAM
- 1 node without GPU, QuadCore i7 3.5GHz, 8GB de RAM
- 1 node with GeForce GTX 670 2GB, QuadCore i7 3GHz, 8GB RAM
- 1 node with GeForce GTX 670 2GB, Core2Duo 2.4GHz, 2GB RAM
- 1 node with 2 GeForce GTX 590 3GB, Quad Core i7 3.5GHz, 8GB RAM

- 1Gbps Ethernet
- Scientific Linux release 6.1 and also Open Suse 11.2 (dual boot)
Integrating rCUDA with SLURM

- matrixMul from NVIDIA’s SDK
  - matrixA 320x320
  - matrixB 640x320

Total execution time including SLURM
Integrating rCUDA with SLURM

- matrixMul from NVIDIA’s SDK

The higher performance of rCUDA is due to a pre-initialized CUDA environment in the rCUDA server.

Total execution time including SLURM
Integrating rCUDA with SLURM

- matrixMul from NVIDIA’s SDK

The time for initializing the CUDA environment is about 0.9 seconds

Total execution time including SLURM
Integrating rCUDA with SLURM

- matrixMul from NVIDIA’s SDK

Scheduling the use of remote GPUs takes less time than finding a local available GPU.
Integrating rCUDA with SLURM

- Sharing remote GPUs among processes (matrixMul)

![Graph 1: GFlops/node vs Nodes](image1)

*Total execution time including SLURM*

![Graph 2: Time(s) vs Nodes](image2)
Integrating rCUDA with SLURM

- Sharing remote GPUs among processes
  - sortingNetworks from NVIDIA’s SDK
Integrating rCUDA with SLURM

- Sharing remote GPUs among processes (CUDASW++)

The GPU memory gets full with 5 concurrent executions
Outline

- Current GPU computing facilities
- How GPU virtualization adds value to your cluster
- The rCUDA framework: basics and performance
- Integrating rCUDA with SLURM
- rCUDA and low-power processors
**rCUDA and low-power processors**

- There is a clear trend to improve the performance-power ratio in HPC facilities:
  - By leveraging accelerators (Tesla K20 by NVIDIA, FirePro by AMD, Xeon Phi by Intel, …)
  - More recently, by using low-power processors (Intel Atom, ARM, …)
- rCUDA allows to attach a “pool of accelerators” to a computing facility:
  - Less accelerators than nodes → accelerators shared among nodes
  - Performance-power ratio probably improved further
- How interesting is a heterogeneous platform leveraging powerful processors, low-power ones, and remote GPUs?
  - Which is the best configuration?
  - Is energy effectively saved?
rCUDA and low-power processors

- The **computing platforms** analyzed in this study are:
  - **KAYLA**: NVIDIA Tegra 3 ARM Cortex A9 quad-core CPU (1.4 GHz), 2GB DDR3 RAM and Intel 82574L Gigabit Ethernet controller
  - **ATOM**: Intel Atom quad-core CPU s1260 (2.0 GHz), 8GB DDR3 RAM and Intel I350 Gigabit Ethernet controller. *No PCIe connector for a GPU*
  - **XEON**: Intel Xeon X3440 quad-core CPU (2.4GHz), 8GB DDR3 RAM and 82574L Gigabit Ethernet controller

- The **accelerators** analyzed are:
  - **CARMA**: NVIDIA Quadro 1000M GPU (96 cores) with 2GB DDR5 RAM. The rest of the system is the same as for KAYLA
  - **FERMI**: NVIDIA GeForce GTX480 “Fermi” GPU (448 cores) with 1,280 MB DDR3/GDDR5 RAM
rCUDA and low-power processors

1st analysis: using local GPUs; CUDASW++ (top) LAMMPS (bottom)
rCUDA and low-power processors

- 1st analysis: using local GPUs; CUDASW++ (top) LAMMPS (bottom)

- The XEON+FERMI combination achieves the best performance
rCUDA and low-power processors

- 1st analysis: using local GPUs; CUDASW++ (top) LAMMPS (bottom)

- The XEON+FERMI combination achieves the best performance
- LAMMPS makes a more intensive use of the GPU than CUDASW++
rCUDA and low-power processors

- 1st analysis: using local GPUs; CUDASW++ (top) LAMMPS (bottom)

- The XEON+FERMI combination achieves the best performance
- LAMMPS makes a more intensive use of the GPU than CUDASW++
- The lower performance of the PCIe bus in Kayla reduces performance
**rCUDA and low-power processors**

- 1st analysis: using **local GPUs**; CUDASW++ (top) LAMMPS (bottom)

- The XEON+FERMI combination achieves the best performance

- LAMMPS makes a more intensive use of the GPU than CUDASW++

- The lower performance of the PCIe bus in Kayla reduces performance

- The 96 cores of CARMA provide a noticeably lower performance than a more powerful GPU
**rCUDA and low-power processors**

- 1\textsuperscript{st} analysis: using **local GPUs**; CUDASW++ (top) LAMMPS (bottom)

![Bar charts comparing time, average power, and energy consumption for CARMA, KAYLA+FERMI, and XEON+FERMI. CARMA requires less power.](image)
rCUDA and low-power processors

- 1st analysis: using local GPUs; CUDASW++ (top) LAMMPS (bottom)

When execution time and required power are combined into energy, XEON+FERMI is more efficient
rCUDA and low-power processors

- 1st analysis: using local GPUs; CUDASW++ (top) LAMMPS (bottom)

Summing up:
- If execution time is a concern, then use XEON+FERMI
- If power is to be minimized, CARMA is a good choice
- The lower performance of KAYLA’s PCIe decreases the interest of this option for single system local-GPU usage
- If energy must be minimized, XEON+FERMI is the right choice
rCUDA and low-power processors

• 2\textsuperscript{nd} analysis: use of \textbf{remote GPUs}

• The use of the CUADRO GPU within the CARMA system is discarded as rCUDA server due to its poor performance. \textbf{Only the FERMI GPU is considered}

• Six different combinations of rCUDA clients and servers are feasible:

<table>
<thead>
<tr>
<th>Combination</th>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>KAYLA</td>
<td>KAYLA+FERMI</td>
</tr>
<tr>
<td>2</td>
<td>ATOM</td>
<td>KAYLA+FERMI</td>
</tr>
<tr>
<td>3</td>
<td>XEON</td>
<td>KAYLA+FERMI</td>
</tr>
<tr>
<td>4</td>
<td>KAYLA</td>
<td>XEON+FERMI</td>
</tr>
<tr>
<td>5</td>
<td>ATOM</td>
<td>XEON+FERMI</td>
</tr>
<tr>
<td>6</td>
<td>XEON</td>
<td>XEON+FERMI</td>
</tr>
</tbody>
</table>

• Only one client system and one server system are used at each experiment
rCUDA and low-power processors

- 2\textsuperscript{nd} analysis: use of remote GPUs

![Graphs showing power consumption and energy usage for different configurations.](image)
rCUDA and low-power processors

- 2nd analysis: use of remote GPUs

The KAYLA-based server presents much higher execution time

CUDASW++
rCUDA and low-power processors

- 2nd analysis: use of remote GPUs

The network interface for KAYLA delivers much less than the expected 1Gbps bandwidth

The KAYLA-based server presents much higher execution time

<table>
<thead>
<tr>
<th></th>
<th>KAYLA</th>
<th>ATOM</th>
<th>XEON</th>
<th>KAYLA</th>
<th>ATOM</th>
<th>XEON</th>
</tr>
</thead>
<tbody>
<tr>
<td>KAYLA</td>
<td>70</td>
<td>60</td>
<td>80</td>
<td>50</td>
<td>40</td>
<td>60</td>
</tr>
<tr>
<td>ATOM</td>
<td>60</td>
<td>50</td>
<td>70</td>
<td>40</td>
<td>30</td>
<td>50</td>
</tr>
<tr>
<td>XEON</td>
<td>80</td>
<td>70</td>
<td>90</td>
<td>60</td>
<td>50</td>
<td>80</td>
</tr>
<tr>
<td>KAYLA + FERMI</td>
<td>50</td>
<td>40</td>
<td>60</td>
<td>30</td>
<td>20</td>
<td>40</td>
</tr>
<tr>
<td>XEON + FERMI</td>
<td>60</td>
<td>50</td>
<td>80</td>
<td>40</td>
<td>30</td>
<td>50</td>
</tr>
</tbody>
</table>

![Graph showing network bandwidth comparison](image-url)
rCUDA and low-power processors

• 2\textsuperscript{nd} analysis: use of remote GPUs

- The lower bandwidth of KAYLA (and ATOM) can also be noticed when using the XEON server.
• 2\textsuperscript{nd} analysis: use of remote GPUs

Most of the power and energy is required by the server side, as it hosts the GPU.
\textit{rCUDA and low-power processors}

- \textbf{2\textsuperscript{nd} analysis: use of remote GPUs}

The much shorter execution time of XEON-based servers render more energy-efficient systems.
rCUDA and low-power processors

- 2nd analysis: use of remote GPUs

<table>
<thead>
<tr>
<th>Combination</th>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>KAYLA</td>
<td>KAYLA+FERMI</td>
</tr>
<tr>
<td>2</td>
<td>KAYLA</td>
<td>XEON+FERMI</td>
</tr>
<tr>
<td>3</td>
<td>ATOM</td>
<td>XEON+FERMI</td>
</tr>
<tr>
<td>4</td>
<td>XEON</td>
<td>XEON+FERMI</td>
</tr>
</tbody>
</table>
rCUDA and low-power processors

- 2nd analysis: use of remote GPUs

Similar conclusions to CUDASW++ apply to LAMMPS, due to the network bottleneck.
Outline

- Current GPU computing facilities
- How GPU virtualization adds value to your cluster
- The rCUDA framework: basics and performance
- Integrating rCUDA with SLURM
- rCUDA and low-power processors
- rCUDA work in progress
Scheduling in SLURM taking into account the actual GPU utilization
  - Analysis of the benefits of using the SLURM+rCUDA suite in real production clusters with InfiniBand fabric

Test rCUDA with more applications from list of “Popular GPU-accelerated Applications” from NVIDIA

Support for CUDA 5.5

Integration of GPUDirect RDMA into rCUDA communications

Further analyze hybrid ARM-XEON scenarios
  - Use of better network protocols for 1Gbps Ethernet
  - Use of InfiniBand fabrics
http://www.rcuda.net
More than 350 requests

Thanks to Mellanox for its support to this work

rCUDA team

Antonio Peña(1)       Adrian Castelló
Carlos Reaño           Sergio Iserte
Federico Silla          Vicente Roca
José Duato               Rafael Mayo

Enrique S. Quintana-Ortí

(1) Now at Argonne National Lab. (USA)