rCUDA: a ready-to-use remote GPU virtualization framework

Rafael Mayo
Universitat Jaume I de Castelló (Spain)

Joint research effort
Outline

- GPU computing
- rCUDA framework
- Other GPU virtualization frameworks
- rCUDA SLURM integration
- SDK and real applications tests
Outline

- GPU computing
- rCUDA framework
- Other GPU virtualization frameworks
- rCUDA SLURM integration
- SDK and real applications tests
GPU computing

- GPU computing: defines all the technological issues for using the GPU computational power for executing general purpose code.

- GPU computing has experienced remarkable growth in the last years.

Top500 November 2012

- 87.6% N/A
- 9.6% Nvidia Fermi
- Intel Xeon Phi
- ATI Radeon
- Nvidia Kepler
- IBM Cell
GPU computing

- GPU computing: defines all the technological issues for using the GPU computational power for executing general purpose code
- GPU computing has experienced remarkable growth in the last years

Top500
November 2012
- The basic building block is a node with 1 or more GPUs
GPU computing

- From the programming point of view:
  - A set of nodes, each one with:
    - one or more CPUs (with several cores per CPU)
    - one or more GPUs (1-4)
    - disjoint memory spaces for each GPU and CPU
  - An interconnection network
GPU computing

- Development tools have been introduced in order to ease the programming of the GPUs
- Two main approaches in GPU computing development environments:
  - CUDA → NVIDIA proprietary
  - OpenCL → open standard
Basically CUDA and OpenCL have the same working scheme:

- **Compilation**: Separate CPU code from GPU code (GPU kernel)

- **Execution**:
  - **Data transfers**: CPU and GPU memory spaces
    1. **Before** GPU kernel execution: data from CPU memory space to GPU memory space
    2. **Computation**: Kernel execution
    3. **After** GPU kernel execution: results from GPU memory space to CPU memory space
• Time spent on data transfers may not be negligible

![Influence of data transfers for SGEMM](chart)

- **Pinned Memory**
- **Non-Pinned Memory**
For the right kind 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 in 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
Outline

- GPU computing
- rCUDA framework
- Other GPU virtualization frameworks
- rCUDA SLURM integration
- SDK and real applications tests
rCUDA

A framework enabling that a CUDA-based application running in one node can access GPUs in other nodes

It is useful when you have:

- Moderate level of data parallelism
- Applications for multi GPU computing
Virtualized Remote GPUs
rCUDA framework

CUDA application

Application

CUDA libraries

CUDA device
rCUDA framework

Client side  \(\leftarrow\)  CUDA application  \(\rightarrow\)  Server side

Application

CUDA driver + runtime

CUDA libraries

CUDA device

CUDA device
rCUDA framework

Client side

- Application
- rCUDA library
- Network device

CUDA application

- rCUDA daemon
- CUDA libraries
- Network device

Server side

- CUDA device
rCUDA framework

Client side

Application

rCUDA library

Network device

CUDA application

rCUDA daemon

Network device

Server side

CUDA driver + runtime

CUDA device
rCUDA framework

Client side

Application

rCUDA library

Network device

CUDA application

rCUDA daemon

Network device

Server side

CUDA driver + runtime

CUDA 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
Optimized communications

- Use GPUDirect to avoid memory copies
- Pipeline to improve performance
- Preallocated pinned memory buffers
- Optimal pipeline block size
rCUDA framework

- Efficient pipeline implementation
  - Synchronous transfers $\rightarrow$ Pageable memory
rCUDA framework

- Efficient pipeline implementation
  - Synchronous transfers → Pageable memory
CUDAnet framework

- Efficient pipeline implementation
  - Synchronous transfers → Pageable memory
rCUDA framework

- Efficient pipeline implementation
  - Synchronous transfers → Pageable memory

![Diagram showing network and memory connections between client and server sides with pinned memory and send/receive function.](image)
- Efficient pipeline implementation
  - Synchronous transfers $\rightarrow$ Pageable memory

![Diagram of rCUDA framework](image-url)
rCUDA framework

- Efficient pipeline implementation
  - Synchronous transfers → Pageable memory
rCUDA framework

- Efficient pipeline implementation
  - Synchronous transfers $\rightarrow$ Pageable memory

![Diagram of rCUDA framework](image)

Client side

Server side

- Main Memory
- Pinned
- SEND / RECV
- Network
- Main Memory
- Pinned
- GPU Memory
rCUDA framework

- Efficient pipeline implementation
  - Synchronous transfers → Pageable memory

[Diagram of network communication between client and server sides with SEND/RECV and pinned memory]
rCUDA framework

- Efficient pipeline implementation
  - Synchronous transfers → Pageable memory
- Efficient pipeline implementation
  - Asynchronous transfers → Pinned memory
- Efficient pipeline implementation
  - Asynchronous transfers $\rightarrow$ Pinned memory
rCUDA framework

- Efficient pipeline implementation
  - Asynchronous transfers → Pinned memory
The rCUDA framework includes the following features:

- **Efficient pipeline implementation**
- **Asynchronous transfers → Pinned memory**

The diagram illustrates the flow of data between the client and server sides, involving main memory and GPU memory. The process includes

- A request from the client
- Network communication
- A request to the server
- RDMA READ operation
- Main Memory transfer
- GPU Memory access
rCUDA framework

- Efficient pipeline implementation
  - Asynchronous transfers $\rightarrow$ Pinned memory
rCUDA framework

- Efficient pipeline implementation
  - Asynchronous transfers $\rightarrow$ Pinned memory
- Efficient pipeline implementation
  - Asynchronous transfers $\rightarrow$ Pinned memory
rCUDA framework

- Efficient pipeline implementation
  - Asynchronous transfers → Pinned memory
- Efficient pipeline implementation
  - Asynchronous transfers $\rightarrow$ Pinned memory
Pipeline block size for Infiniband FDR

- NVIDIA Tesla K20
- Mellanox ConnectX-3 + SX6025 Mellanox switch
- Host to device copy with pinned memory
- Host to device copy with pageable memory
- Copy a small dataset for latency study (64 bytes)
Outline

- GPU computing
- rCUDA framework
- Other GPU virtualization frameworks
- rCUDA SLURM integration
- SDK and real applications tests
Several efforts have been done regarding GPU virtualization in the recent 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)
GPU virtualization

- Comparison of virtualization solutions.
  - Latency (transfer 64 bytes)
    - Intel Xeon E5-2620 (6 cores) 2,0GHz
    - GPU NVIDIA Tesla K20
    - Mellanox ConnectX-3 single-port InfiniBand Adapter (FDR)
    - CentOS 6.3 + Mellanox OFED 1.5.3

<table>
<thead>
<tr>
<th></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>
• Bandwidth host to device using pageable memory
• Bandwidth device to host using pageable memory
GPU virtualization

- Bandwidth host to device using pinned memory
GPU virtualization

- Bandwidth device to host using pinned memory
- GPU computing
- rCUDA framework
- Other GPU virtualization frameworks
- rCUDA SLURM integration
- SDK and real applications tests
SLURM integration

- SLURM job scheduler.
- Does not understand about virtualized GPUs.
- Add a new GRES (general resource) in order to manage the 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 for the job.
SLURM integration

Node daemon

slurmd

slurmd

slurmd

Controller daemon

slurmctld

commands

srun

squeue

smap

sinfo

scontrol

scancel
SLURM integration

slurm.cof

ClusterName=rcu
...
GresTypes=gpu,rgpu
...
NodeName=rcu16 NodeAddr=rcu16 CPUs=8 Sockets=1
  CoresPerSocket=4 ThreadsPerCore=2
RealMemory=7990 Gres=rgpu:4,gpu:4
slurm integration

scontrol show node

NodeName=rcu16 Arch=x86_64 CoresPerSocket=4
   CPUAlloc=0 CPUErr=0 CPUTot=8 Features=(null)
   Gres=rgpu:4,gpu:4
NodeAddr=rcu16 NodeHostName=rcu16
OS=Linux RealMemory=7682 Sockets=1
State=IDLE ThreadsPerCore=2 TmpDisk=0 Weight=1
BootTime=2013-04-24T18:45:35
SlurmdStartTime=2013-04-30T10:02:04
### 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>
SLURM integration

Submit a job

$ srun -N 1 --gres=rgpu:4:512M script.sh...

- Environment variables are initialized by SLURM and used by 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
SLURM integration

Resources per job:
- 1: 2 nodes 3 GPUs
- 2: 3 nodes 1 GPU
- 3: 1 node 0 GPUs
- 4: 2 nodes 1 GPUs
- 5: 2 node 1 GPUs

Graph showing job allocation over time:
- NODE 0: GPU 0, GPU 1
- NODE 1: GPU 0
- NODE 2: GPU 0
- NODE 3: GPU 0

Job Queue:
- 1
- 2
- 3
- 4
- 5
- 5

Note: srnn: error: Unable to allocate resources
SLURM integration

Resources per job:
- 2 nodes 3 GPUs
- 3 nodes 1 GPU
- 1 node 0 GPUs
- 2 nodes 1 GPUs

GPUs are decoupled from nodes:

<table>
<thead>
<tr>
<th>GPU 0</th>
<th>GPU 1</th>
<th>GPU 2</th>
<th>NODE 0</th>
<th>NODE 1</th>
<th>NODE 2</th>
<th>NODE 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Job Queue:
- t0: 1
- t1: 2
- t2: 2
- t3: 4
- t4: 4
- t5: 5
- t6: 5
- t7: 5
- GPU computing
- rCUDA framework
- Other GPU virtualization frameworks
- rCUDA SLURM integration
- SDK and real applications tests
Test system

- Intel Xeon E5-2620 (6 cores) 2.0GHz
- GPU NVIDIA Tesla K20
- Mellanox ConnectX-3 single-port InfiniBand Adapter (FDR)
- Mellanox switch SX6025 (FDR)
- Cisco switch SLM2014 (1Gbps Ethernet)
- CentOS 6.3 + Mellanox OFED 1.5.3
Test CUDA SDK examples
• CUDASW++

Bioinformatics software for Smith-Waterman protein database searches
- CUDABLAST, an 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
- MonteCarlo MultiGPU (from NVIDIA SDK)
- GEMM MultiGPU
  - using **libflame**: high performance dense linear algebra library
    http://www.cs.utexas.edu/~flame/web/
rCUDA work in progress

- Scheduling in SLURM taking into account the actual GPU utilization

- Test in ARM platforms
  - CARMA board as client and rCUDA daemon on Intel Platform equipped with NVIDIA Tesla Fermi
  - Functional test with NVIDIA SDK and LAMMPS
  - Limited performance due to 1Gbps ethernet network

- Test more applications from “Popular GPU-accelerated Applications”

- Support to CUDA 5.5 (release candidate)
http://www.rcuda.net
More than 300 requests

Thanks to Mellanox for its support to this work

rCUDA people

Antonio Peña(1)
Carlos Reaño
Federico Silla
José Duato

Adrian Castelló
Rafael Mayo
Enrique S. Quintana-Ortí

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