UMONS
Université de Mons

Optimizing Performance of Batch of applications on Cloud Servers exploiting Multiple GPUs

Sébastien Frémal, Michel Bagein, Pierre Manneback

firstname.name@umons.ac.be

2012 International Conference on Complex Systems

Palais des Roses Hotel – Agadir, Morocco

November 5-6, 2012

POLYTECH MONS

OLIMP

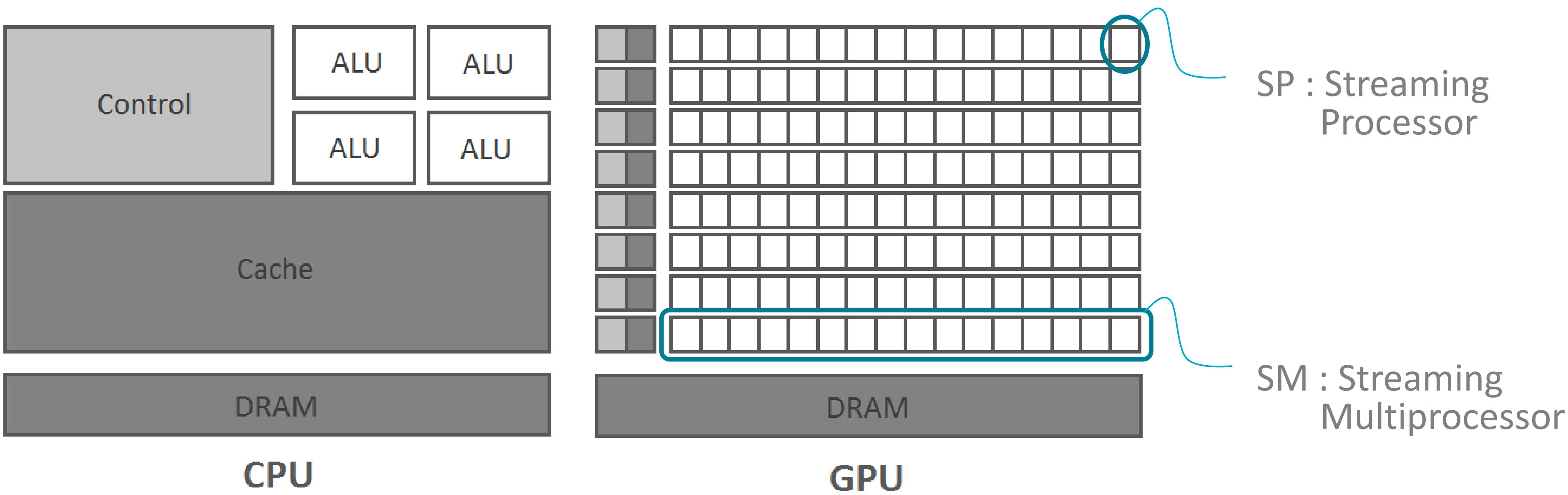
Wallonie

FEDER
UNION EUROPEENNE
Wallonie
LE FONDS EUROPEEN DE DEVELOPPEMENT REGIONAL ET LA WALLONIE INVESTISSENT DANS VOTRE AVENIR.

Part 1 : GPU General Presentation

CPU & GPU: architectural differences

CPU : from 1 to 16 cores (tens of threads)
GPU : from 32 to 800 cores (millions of threads)

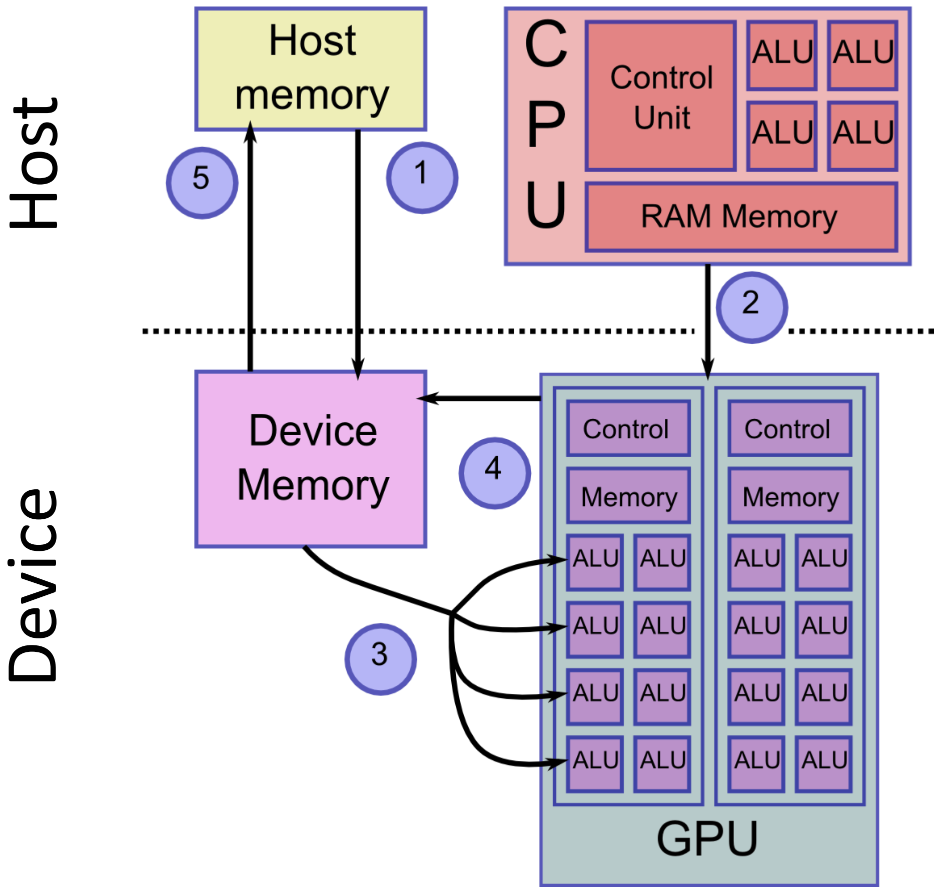


CPU : **asynchronous** code execution on cores
GPU : **synchronous** code execution on SPs of an SM.

Device memories

Memory	Info.	Size	Latency (clock cycles)
Register	On chip – Thread Scope	8192 x 4 bytes	1
Local	Off chip - Thread scope	Undetermined	400 to 600
Shared	On chip – Block scope	16 kB	4
Constant	Off chip – Read only	8 kB	Min. 1
Texture	Off chip - Read only	1 kB / core	4
Global	Off chip – Main memory	Can reach 4 GB	400 to 600

Steps to execute a kernel on a GPU



1. Device memory allocation and data transfer from the host memory to the device memory.
2. Kernel launching.
3. Data reading and processing.
4. Data writing.
5. Results transfer from the device memory to the host memory

Some constraints of GPUs

- Data transfers from a memory to another : needs to get enough treatments to compensate lantecy due to transfer times

- Different applications are not distributed on all available GPUs but are all assigned at the same default GPU

-The number of coexisting CUDA contexts (GPU processus) is limited (+- 30 on a NVIDIA GTX 580)

- CUDA contexts initialization takes some time (400 ms for a NVIDIA GTX 580)

Programming languages : CUDA & OpenCL

- C for CUDA:
  - CUDA = parallel computing architecture developed by Nvidia
  - Proprietary (NVIDIA)
  - cudaMalloc, cudaMemcpy …
- OpenCL:
  - OpenCL = open, royalty-free standard for cross-platform, parallel programming
  - Open (NVIDIA, ATI)
  - clCreateBuffer, clCreateKernel, clSetKernelArg …

Part 2 : GPUs Management

Efficient use of a set of graphic processors

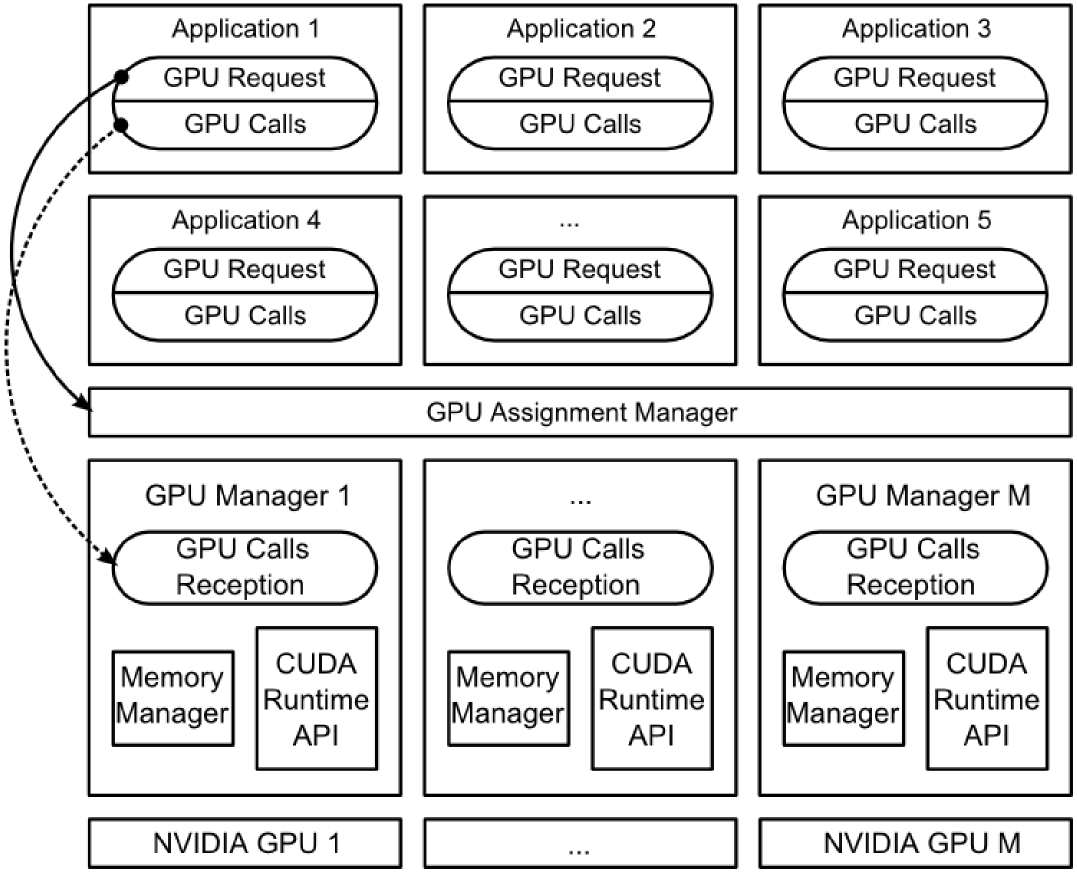
Default behavior of CUDA applications :

- Use the default GPU (GPU 0)
- The system freezes when there are too much CUDA contexts
- Initialization of a CUDA context takes some time

Goals of our work :

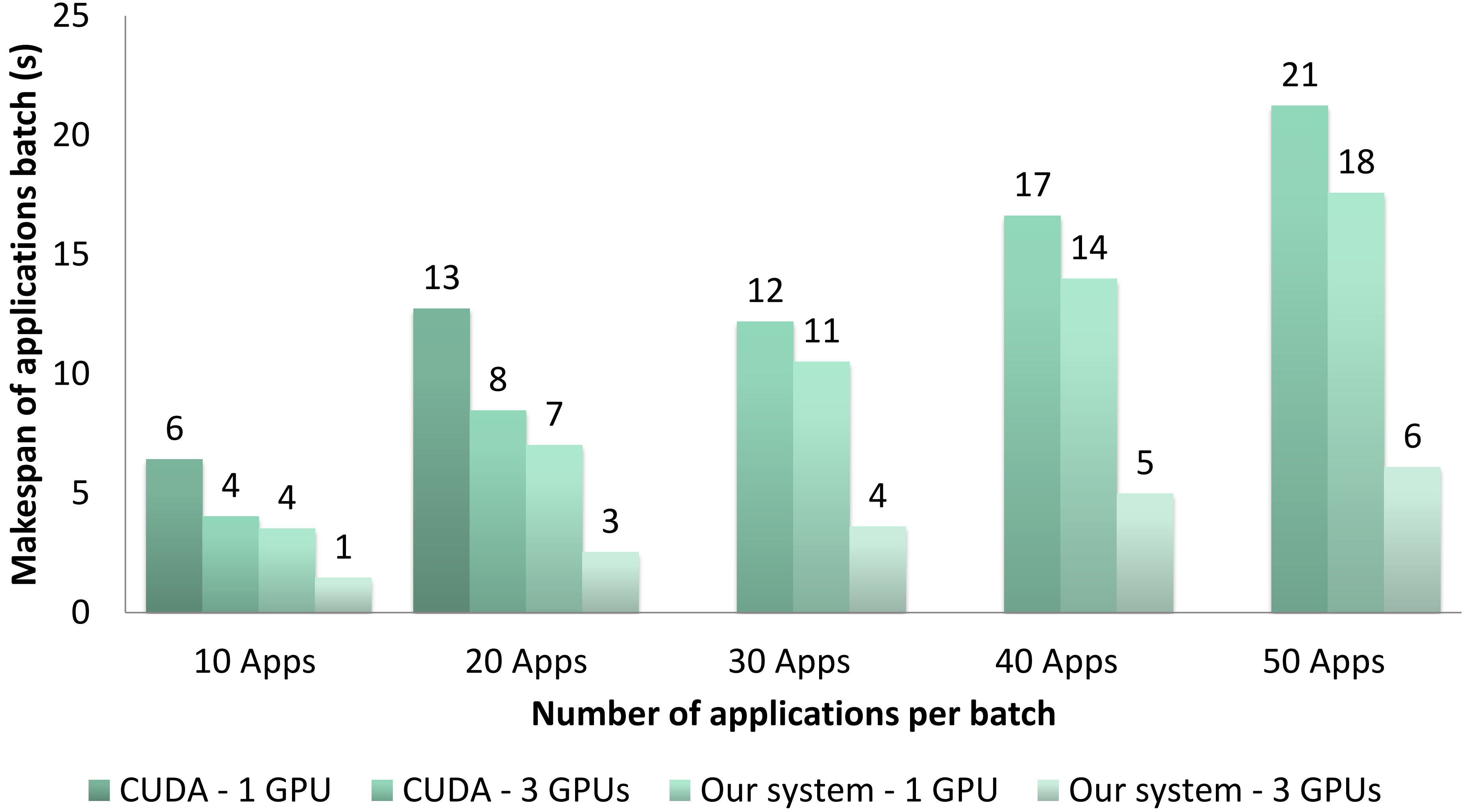
- CUDA context mutualization : sharing CUDA contexts between multiple applications avoids freezing the system when there is a lot of applications and avoids the initialization time of CUDA contexts
- Using all available GPUs : distributing the applications so they use in a transparent way all available GPUs to improve performances

A first system distributing applications on GPUs of a computer

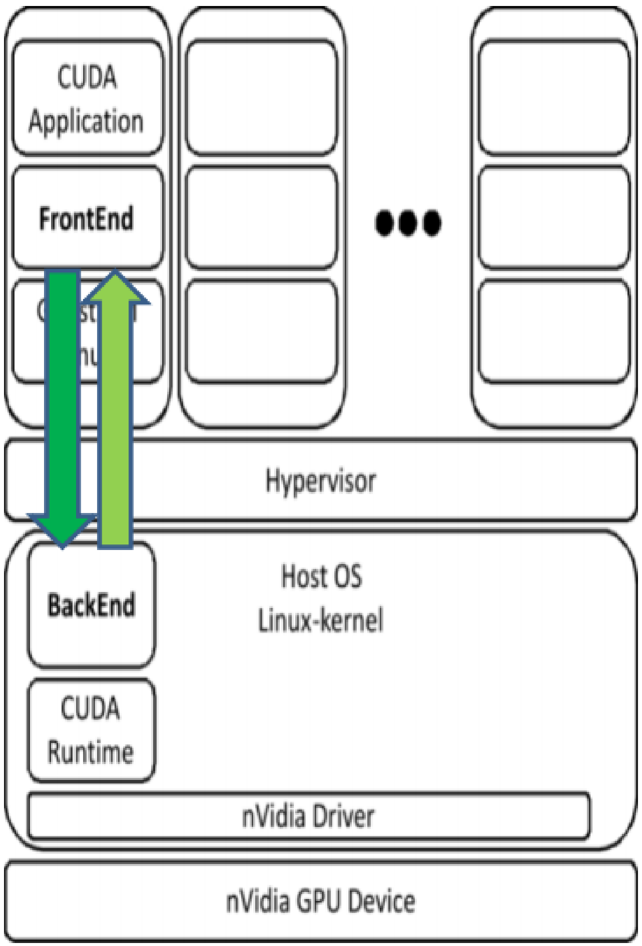


- **Assignment Manager** : distributing applications on GPUs
- **GPU Calls** : interface sending requests for GPU functions to GPU Manager thanks to message queues
- **GPU Manager** : receiving requests from applications and executing them on GPUs. Each one have one thread managine one CUDA context ➔ contexts mutualization
- **Memory Manager** : manages GPU memory and a memory zone shared with applications

Results with the first system



GVirtus : a system allowing an instanced virtual machine to access GPGPUs

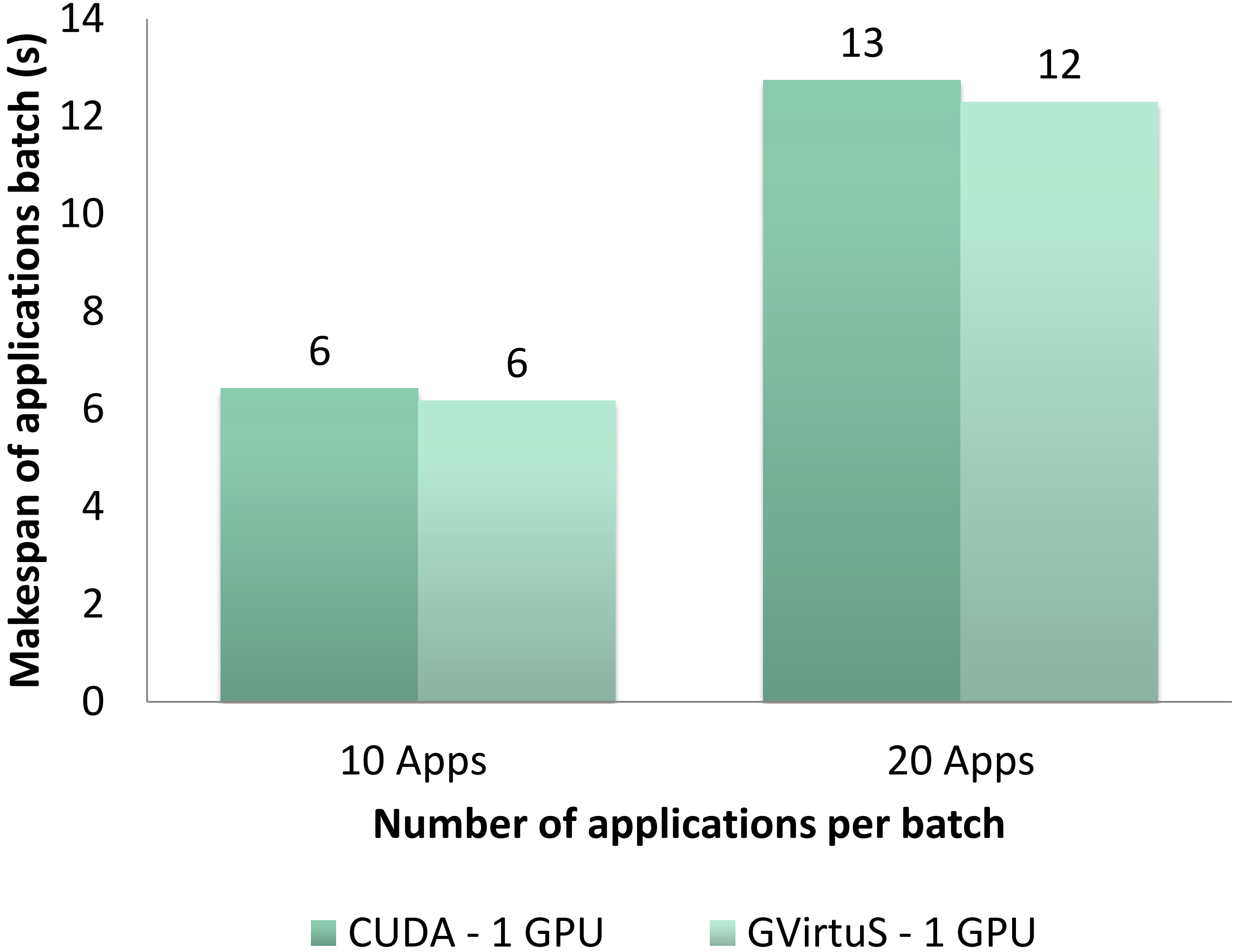


A GPGPU transparent virtualization component for high performance computing clouds, G. Giunta et Al.

Similar to our system but gets some differences :

- Communications are available through TCP and allow the server to communicate with an application in a virtual machine.
- The frontend library uses the CUDA Runtime API function's interfaces. Applications don't need a lot of modifications to use GVirtus.
- GVirtus uses one GPU (the default GPU).
- There is no contexts mutualization. For each application, the server spawns a processus. This processus initializes and manages a CUDA context. It receives requests of the application it's bound to and executes them on the GPU.

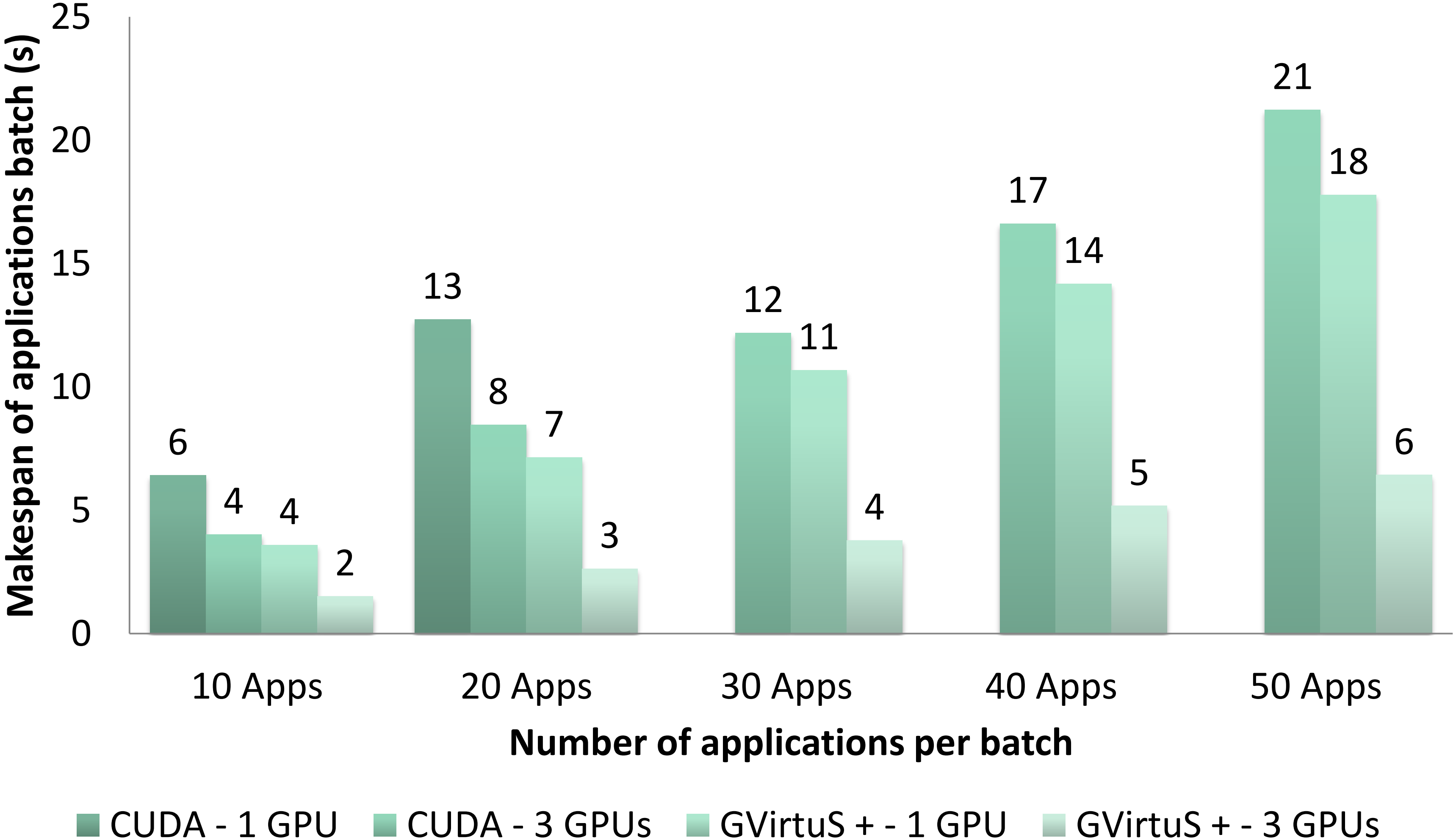
Results with GVirtus



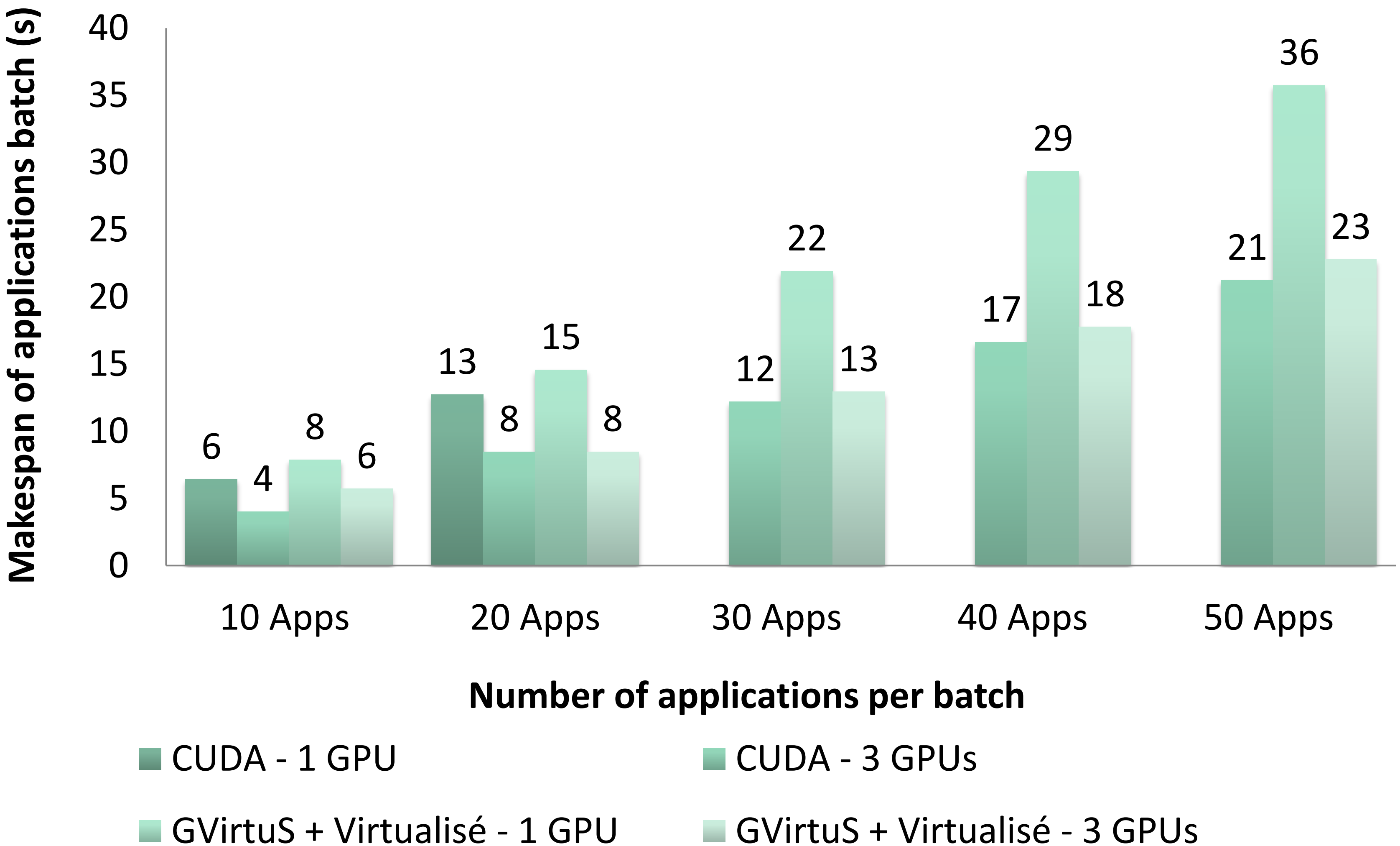
GVirtus + : improvements of GVirtus

- We brought two modifications we experienced with our system to GVirtus to improve its performances :
  - CUDA contexts mutualization
  - Using all available GPUs

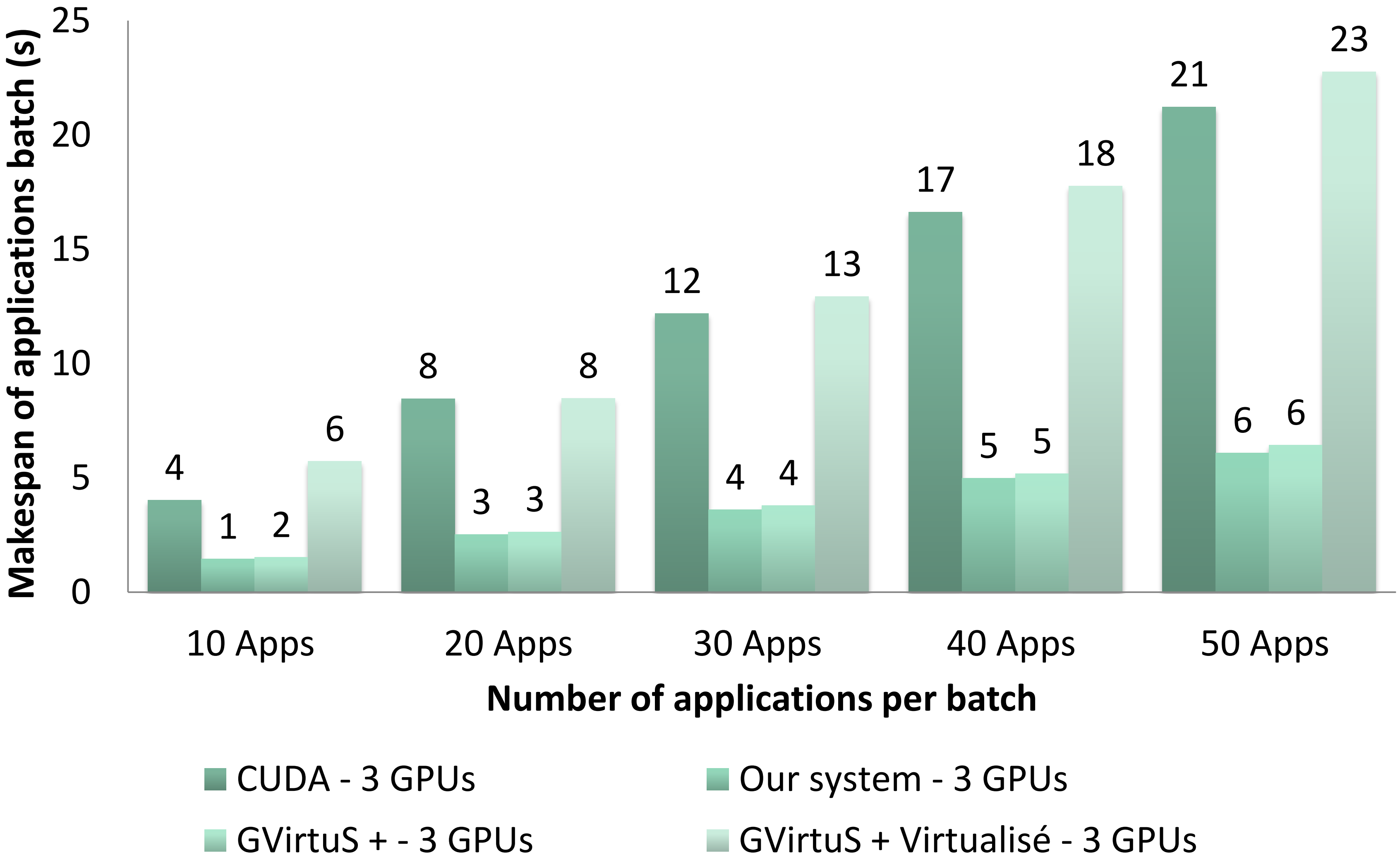
Results with GVirtus +



Results with GVirtus + : virtualized applications



Comparison of systems on 3 GPUs



Conclusions & Perspectives

- Applications benefit from the initialized contexts and the mutualization allows executing more applications.
- Applications are properly distributed on available GPUs.
- Communications will be improved to make global execution faster. This is especially important when applications are virtualized.
- A memory manager will be implemented to remove memory limitation.
- The use of CUDA Stream will be experimented to overlap transfer with kernel execution