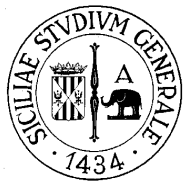
General-purpose programming on GPU

Exploiting multi-GPU systems

Eugenio Rustico
rustico@dmi.unict.it

D.M.I. - Università di Catania

Updated: May 20, 2011



Overview


SLI
GeForce
GeForce
GeForce
CORSAIR
XMS2

With less than 1.000, it is possible to setup a 1 TERAFLOPS workstation...

With less than 1.000, it is possible to setup a 1 TERAFLOPS workstation...

...is it really possible to achieve such (theoretical) peak power?

We've already seen that it is not easy to model a problem in a way that exploits efficiently hardware parallelism - even for problem intrinsicly parallel.

We've already seen that it is not easy to model a problem in a way that exploits efficiently hardware parallelism - even for problem intrinsicly parallel.

To exploit a multi-GPU system we have to think of the problem at **two levels of parallelism**:

We've already seen that it is not easy to model a problem in a way that exploits efficiently hardware parallelism - even for problem intrinsicly parallel.

To exploit a multi-GPU system we have to think of the problem at **two levels of parallelism**:

- A grid of parallel sequences of operations (threads)

We've already seen that it is not easy to model a problem in a way that exploits efficiently hardware parallelism - even for problem intrinsicly parallel.

To exploit a multi-GPU system we have to think of the problem at **two levels of parallelism**:

- A grid of parallel sequences of operations (threads)
- A set of such grids

We've already seen that it is not easy to model a problem in a way that exploits efficiently hardware parallelism - even for problem intrinsicly parallel.

To exploit a multi-GPU system we have to think of the problem at **two levels of parallelism**:

- A grid of parallel sequences of operations (threads)
- A set of such grids

It is absolutely not trivial unless different threads are *completely* independent each other

Within the first call to a CUDA runtime routine a *context* is implicitly created.

Within the first call to a CUDA runtime routine a *context* is implicitly created.

A CUDA context is a sort of *session* of CUDA operations associated to a thread. One context is associated to one GPU (n-to-1), and can operate only with the associated GPU.

Within the first call to a CUDA runtime routine a *context* is implicitly created.

A CUDA context is a sort of *session* of CUDA operations associated to a thread. One context is associated to one GPU (n-to-1), and can operate only with the associated GPU.

One GPU may have several contexts associated; a CPU thread may have only one active context, unless the low-level API is used to switch between contexts.

Within the first call to a CUDA runtime routine a *context* is implicitly created.

A CUDA context is a sort of *session* of CUDA operations associated to a thread. One context is associated to one GPU (n-to-1), and can operate only with the associated GPU.

One GPU may have several contexts associated; a CPU thread may have only one active context, unless the low-level API is used to switch between contexts.

We can see a context as a hidden set of settings and structures that are used by the CUDA runtime, when we call a CUDA function like a `cudaMemCpy()`, to answer the question: *which device are we working on? With which settings?*

To be able to use multiple GPUs simultaneously, we need one context per GPU.

To be able to use multiple GPUs simultaneously, we need one context per GPU.

- With **low-level API**, we can explictly switch between context (`cuCtx...()` methods)

To be able to use multiple GPUs simultaneously, we need one context per GPU.

- With **low-level API**, we can explictly switch between context (`cuCtx...()` methods)
- With **high-level API**, we need one separate thread per GPU (multithreaded applications)

To be able to use multiple GPUs simultaneously, we need one context per GPU.

- With **low-level API**, we can explictly switch between context (`cuCtx...()` methods)
- With **high-level API**, we need one separate thread per GPU (multithreaded applications)

High-level APIs are way easier to use, but designing a robust multi-threaded application may be challenging. In general, we will need a thread-safe synchronization mechanism (e.g. signals, semaphores, barriers, etc.).

To be able to use multiple GPUs simultaneously, we need one context per GPU.

- With **low-level API**, we can explictly switch between context (`cuCtx...()` methods)
- With **high-level API**, we need one separate thread per GPU (multithreaded applications)

High-level APIs are way easier to use, but designing a robust multi-threaded application may be challenging. In general, we will need a thread-safe synchronization mechanism (e.g. signals, semaphores, barriers, etc.).

It is a recommended practice to call `cudaThreadExit()` at the and of every host thread to explicitly clean up the context.

At the problem level: how do we split the computations set across the available GPUs?

There are two general approaches:

At the problem level: how do we split the computations set across the available GPUs?

There are two general approaches:

1. Divide in the data domain (split input, assign to GPUs, manage overlapping)

At the problem level: how do we split the computations set across the available GPUs?

There are two general approaches:

1. Divide in the data domain (split input, assign to GPUs, manage overlapping)
2. Divide in the computation domain (pass all input data across all GPUs in pipeline)

At the problem level: how do we split the computations set across the available GPUs?

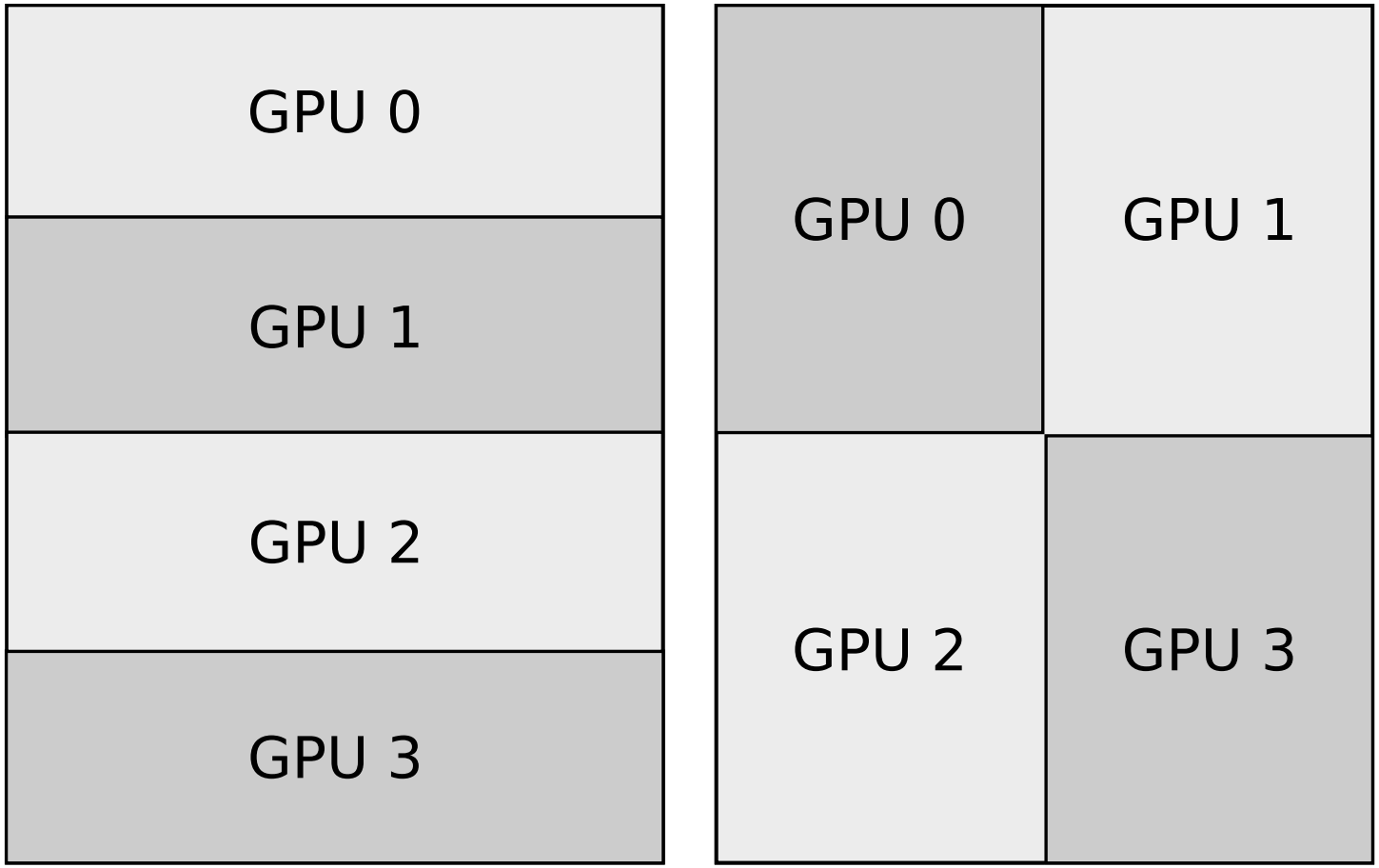
There are two general approaches:

1. Divide in the data domain (split input, assign to GPUs, manage overlapping)
2. Divide in the computation domain (pass all input data across all GPUs in pipeline)

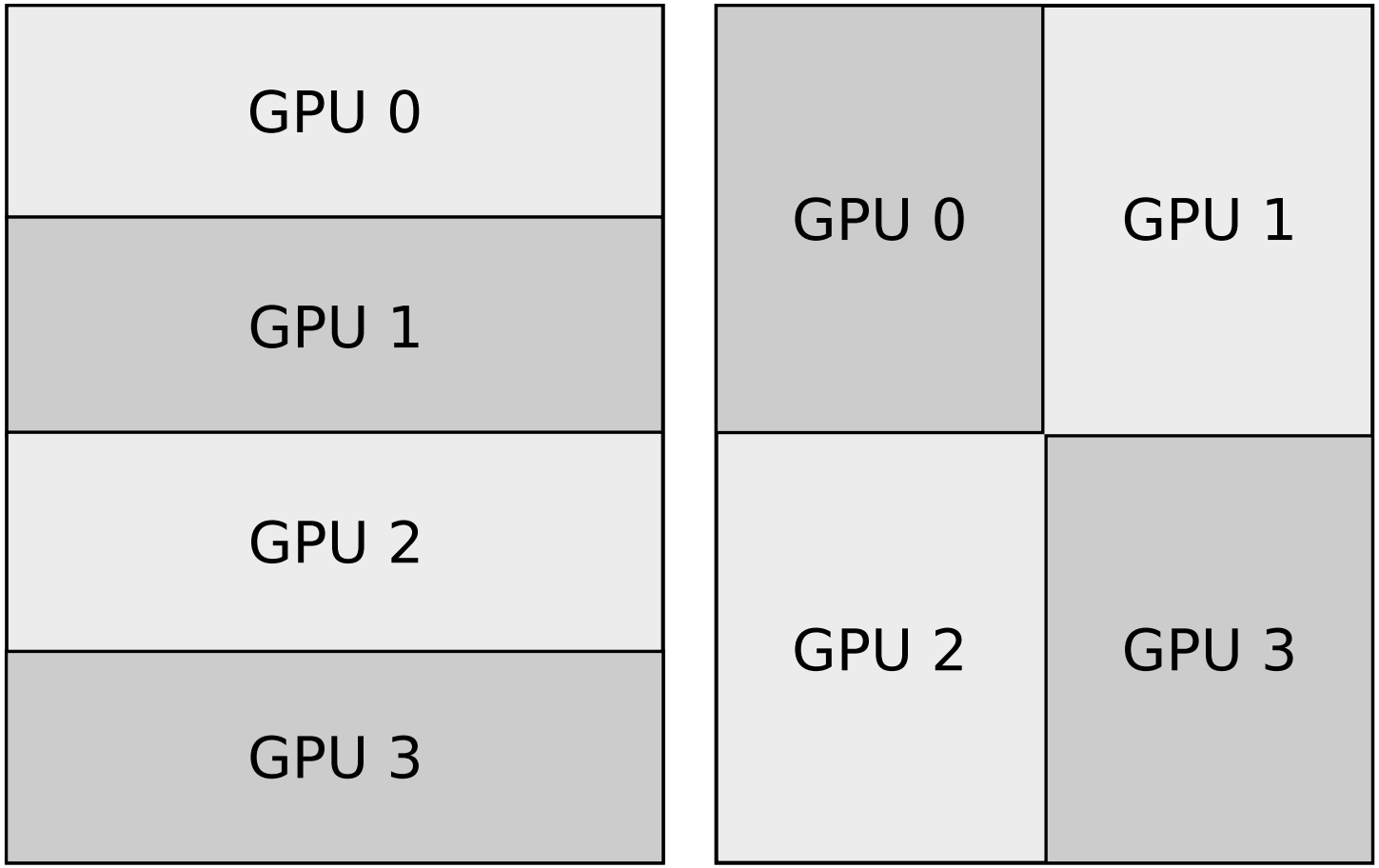
Approach n.1 is in most cases the simplest and most efficient one.

How should we divide an image?

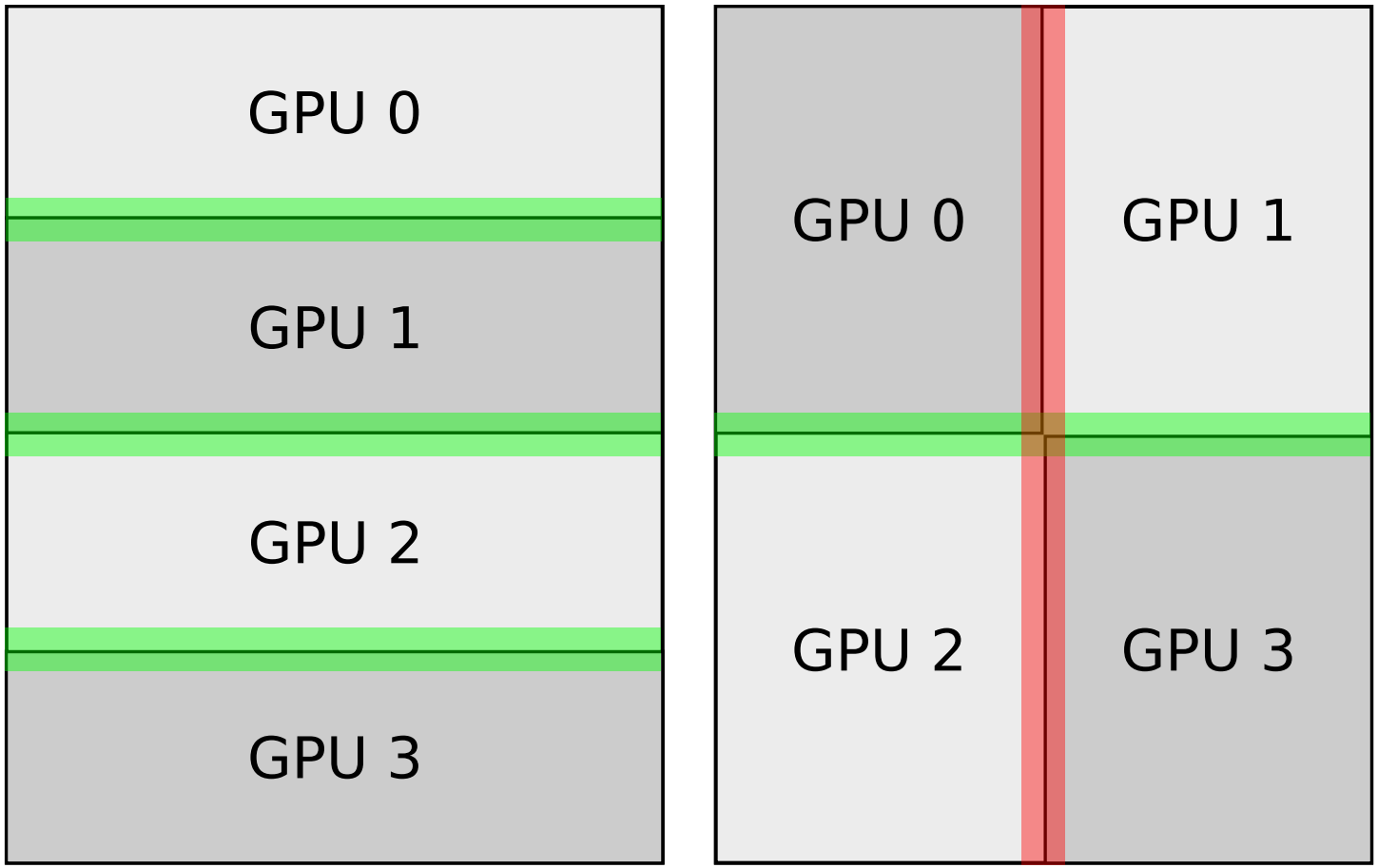
How should we divide an image?



How should we divide an image?



Why first is preferable? When it is not?



Second has a better perimeter/area ratio (i.e. divergences and special cases), however: sparse border accesses (is memory per row?), two neighbors for every GPU (imagine 2x3!)

One GPU may be more powerful than another, or may have to deal also with visualization (e.g. an attached monitor), or may be in a PCI slot slower than other, or may simply have more operations or less coalesced memory requirements.

One GPU may be more powerful than another, or may have to deal also with visualization (e.g. an attached monitor), or may be in a PCI slot slower than other, or may simply have more operations or less coalesced memory requirements.

Dynamic *load balancing* may become fundamental to obtain a real speedup. Can you imagine a generic load balancing technique suitable for different applications?

One GPU may be more powerful than another, or may have to deal also with visualization (e.g. an attached monitor), or may be in a PCI slot slower than other, or may simply have more operations or less coalesced memory requirements.

Dynamic *load balancing* may become fundamental to obtain a real speedup. Can you imagine a generic load balancing technique suitable for different applications?

Answer: the only general technique is timing *a posteriori*. And it is very complex: requires some basic signal processing to get rid of oscillations and avoid local minima.

There are two main limitations in multi-GPU: **interdependence** and **transfers**.

There are two main limitations in multi-GPU: **interdependence** and **transfers**.

Most applications present a form of data **interdependence**. Example: if we need to apply in sequence two convolution effects to an image, kernel nr. 2 requires the updated borders of kernel nr. 1 from other GPUs. Imagine a continuous timeline (streaming)...

There are two main limitations in multi-GPU: **interdependence** and **transfers**.

Most applications present a form of data **interdependence**. Example: if we need to apply in sequence two convolution effects to an image, kernel nr. 2 requires the updated borders of kernel nr. 1 from other GPUs. Imagine a continuous timeline (streaming)...

Reality is even worse than this: **all transfers must pass through the host**, no direct GPU-GPU transfers up to CUDA 3.2.

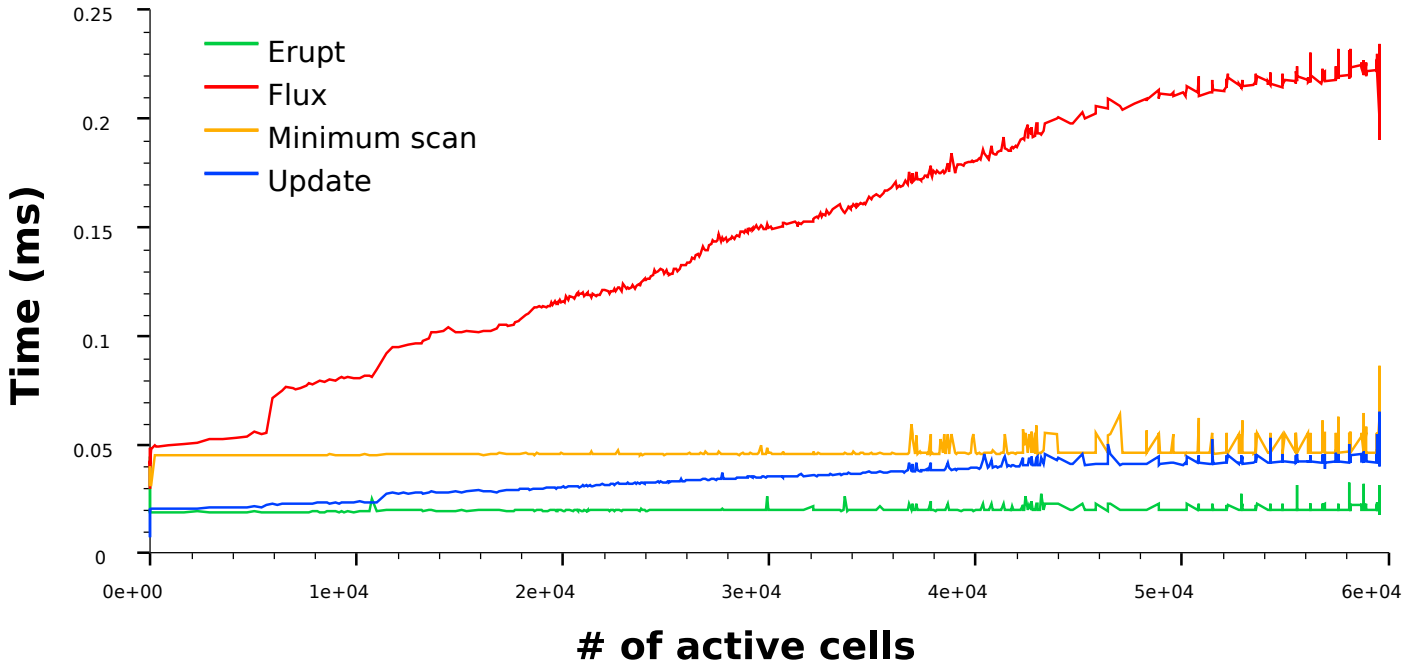
There are two main limitations in multi-GPU: **interdependence** and **transfers**.

Most applications present a form of data **interdependence**. Example: if we need to apply in sequence two convolution effects to an image, kernel nr. 2 requires the updated borders of kernel nr. 1 from other GPUs. Imagine a continuous timeline (streaming)...

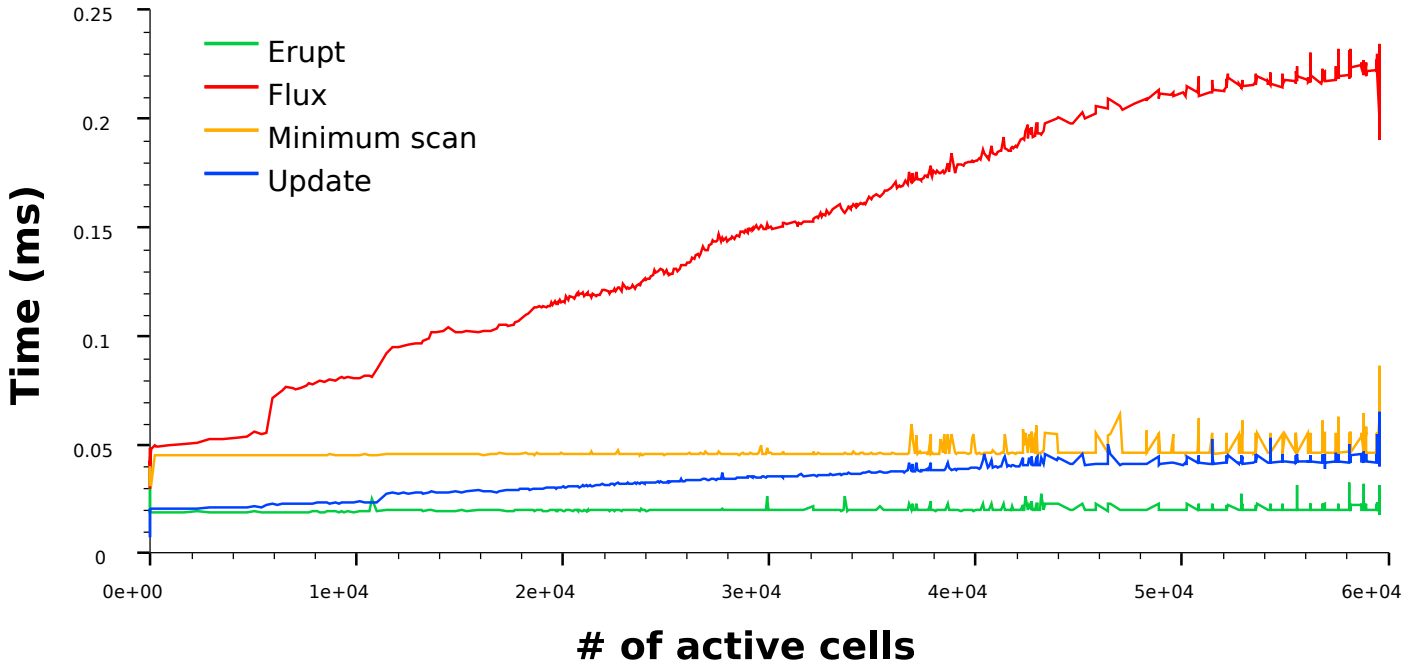
Reality is even worse than this: **all transfers must pass through the host**, no direct GPU-GPU transfers up to CUDA 3.2.

The higher the number of GPUs, the more memory transfers will request exclusive use of PCI bus. The more memory transfers, the less benefits of using multiple GPUs. Need **asynchronous operations** to cover latencies (next lecture).

An example of expected vs. real speedup:



An example of expected vs. real speedup:



Kernel launch latency and saturation threshold cause

```
time(data/2) > time(data)/2
```

No matter how I split the input data: with K GPUs, in most cases, speedup will be $< K$.

Examples:

- MAGFLOW (image-like)
- SPH (list-like)

At the moment we write, CUDA 4 is still release candidate. Differencies with CUDA 3.2 mostly regard multi-GPU:

At the moment we write, CUDA 4 is still release candidate. Differencies with CUDA 3.2 mostly regard multi-GPU:

- **Unified Virtual Addressing**: on x86_64 systems, a unique address space for CPU and GPUs.

At the moment we write, CUDA 4 is still release candidate. Differencies with CUDA 3.2 mostly regard multi-GPU:

- **Unified Virtual Addressing**: on x86_64 systems, a unique address space for CPU and GPUs.
- **Implicit context switch**: just repeat `cudaSetDevice()` in high level API to switch device

At the moment we write, CUDA 4 is still release candidate. Differencies with CUDA 3.2 mostly regard multi-GPU:

- **Unified Virtual Addressing**: on x86_64 systems, a unique address space for CPU and GPUs.
- **Implicit context switch**: just repeat `cudaSetDevice()` in high level API to switch device
- **Device to device direct memory transfers**, `cudaMemCpy()` bursts and direct access

At the moment we write, CUDA 4 is still release candidate. Differencies with CUDA 3.2 mostly regard multi-GPU:

- **Unified Virtual Addressing**: on x86_64 systems, a unique address space for CPU and GPUs.
- **Implicit context switch**: just repeat `cudaSetDevice()` in high level API to switch device
- **Device to device direct memory transfers**, `cudaMemCpy()` bursts and direct access
- New libraries (`NPP`) and middlewares (`Thrust`)

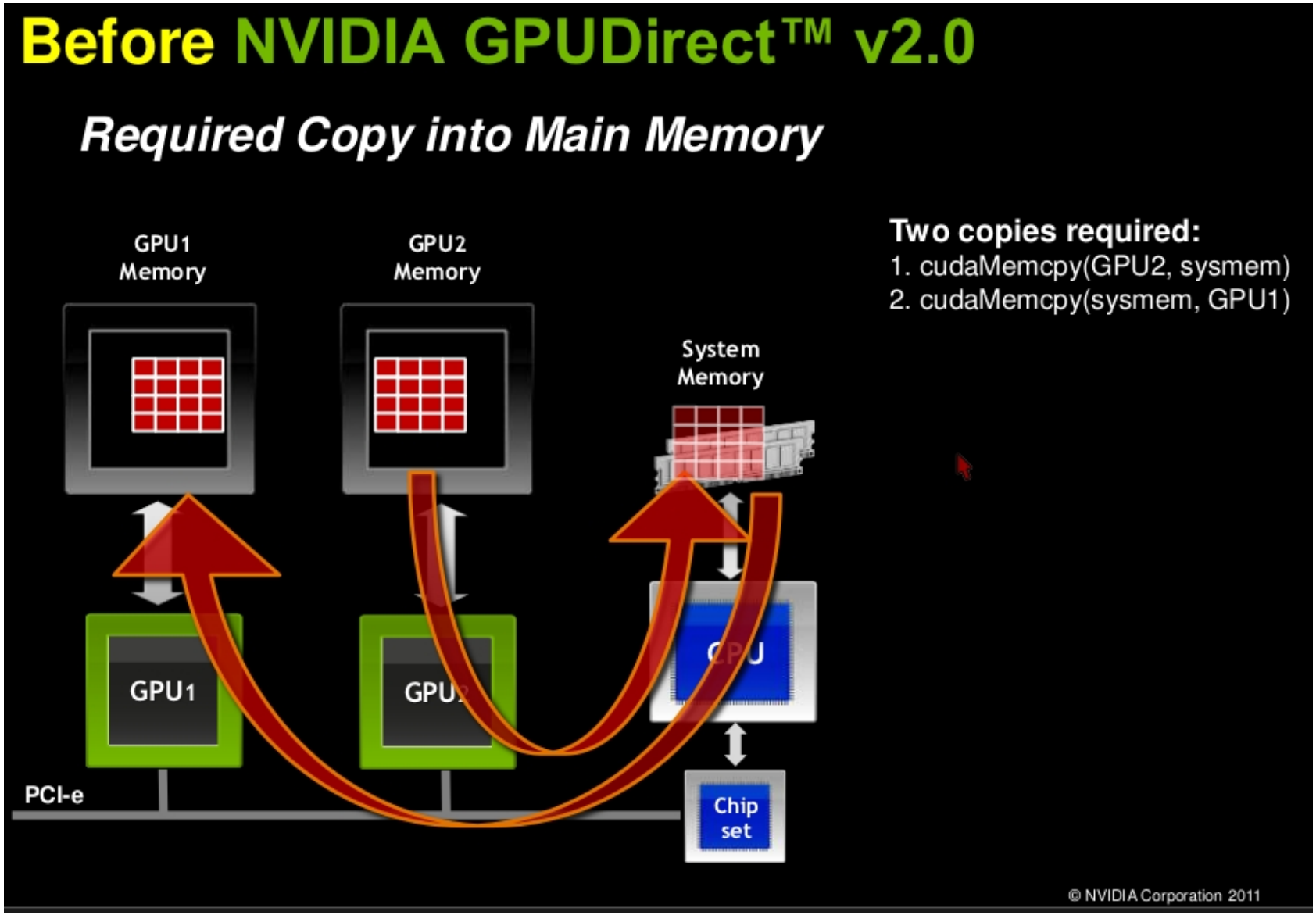
At the moment we write, CUDA 4 is still release candidate. Differencies with CUDA 3.2 mostly regard multi-GPU:

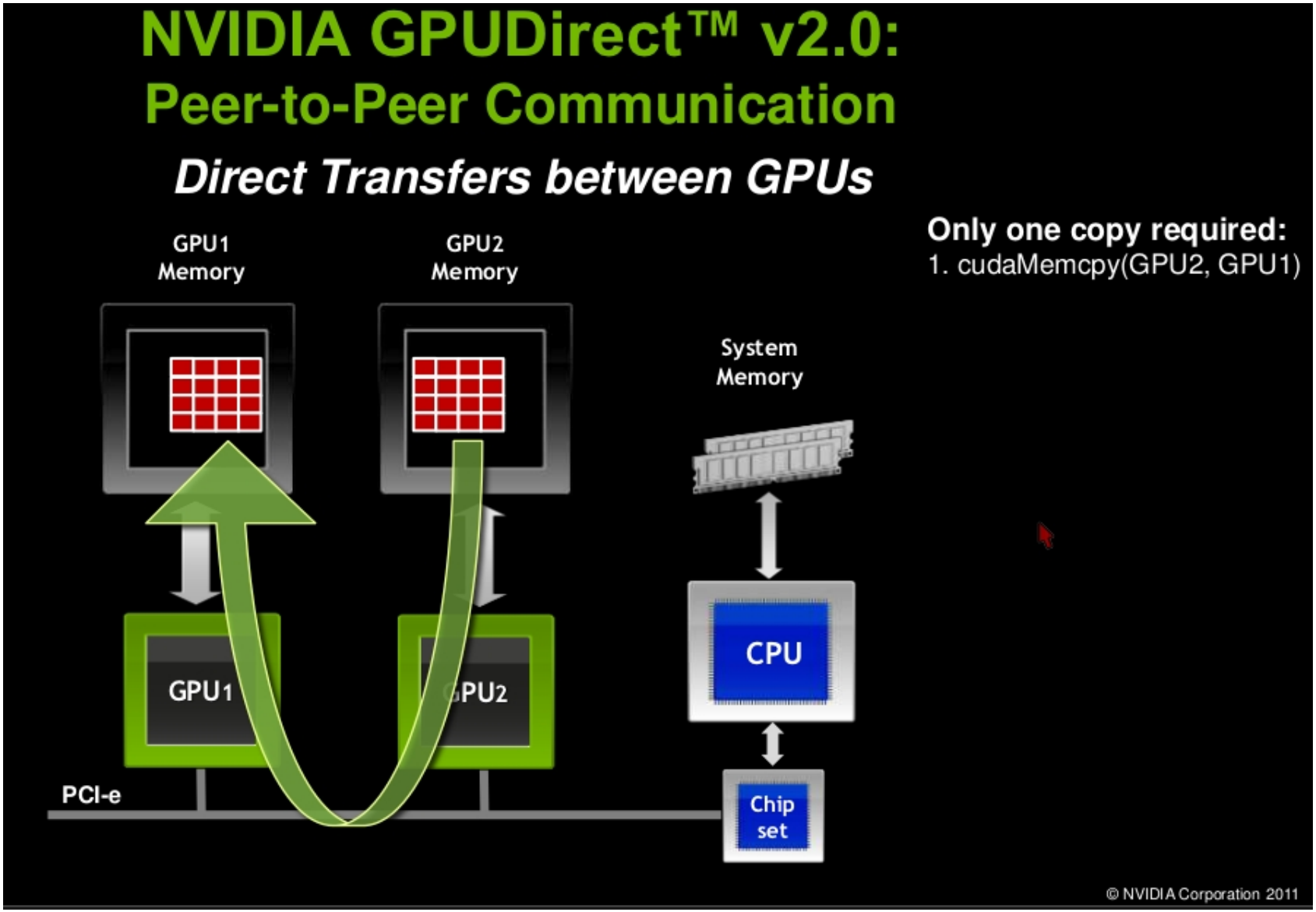
- **Unified Virtual Addressing**: on x86_64 systems, a unique address space for CPU and GPUs.
- **Implicit context switch**: just repeat `cudaSetDevice()` in high level API to switch device
- **Device to device direct memory transfers**, `cudaMemCpy()` bursts and direct access
- New libraries (`NPP`) and middlewares (`Thrust`)
- Dynamic memory allocation and virtual functions in device code

At the moment we write, CUDA 4 is still release candidate. Differencies with CUDA 3.2 mostly regard multi-GPU:

- **Unified Virtual Addressing**: on x86_64 systems, a unique address space for CPU and GPUs.
- **Implicit context switch**: just repeat `cudaSetDevice()` in high level API to switch device
- **Device to device direct memory transfers**, `cudaMemCpy()` bursts and direct access
- New libraries (`NPP`) and middlewares (`Thrust`)
- Dynamic memory allocation and virtual functions in device code

New method to retrieve memory type by address and simpler `cudaMemcpy()` (`cudaMemcpyDefault()`: automatically determine transfer direction)


Before NVIDIA GPUDirect™ v2.0
Required Copy into Main Memory
GPU1 Memory
GPU2 Memory
System Memory
Two copies required:
1. cudaMemcpy(GPU2, sysmem)
2. cudaMemcpy(sysmem, GPU1)
GPU1
GPU2
CPU
PCI-e
Chip set
© NVIDIA Corporation 2011


NVIDIA GPUDirect™ v2.0:
Peer-to-Peer Communication
Direct Transfers between GPUs
GPU1 Memory
GPU2 Memory
System Memory
Only one copy required:
1. cudaMemcpy(GPU2, GPU1)
GPU1
GPU2
CPU
PCI-e
Chip set
© NVIDIA Corporation 2011