GMAC 2

Easy and efficient programming for CUDA-based systems

Javier Cabezas	**BSC/Multicoreware**
Isaac Gelado	**BSC/Multicoreware**
Lluís Vilanova	BSC
Nacho Navarro	UPC/BSC
Wen-Mei Hwu	UIUC/Multicoreware

NVIDIA GTC
May 17th, 2012

Barcelona Supercomputing Center (BSC)

- BSC develops a wide range of High Performance Computing applications
  - Alya (Computational Mechanics), RTM (Seismic imaging), Meteorological modeling, Air quality, …
  - Explore the trends in computer architecture (GPUs, FPGAs, …)
- Programming models research group
  - OmpSs: task-based programming model that exploits parallelism at different levels (multi-core, multi-GPU, multi-node, …)
- Accelerators research group
  - System support for accelerators
  - Collaboration with UIUC and Multicoreware Inc.

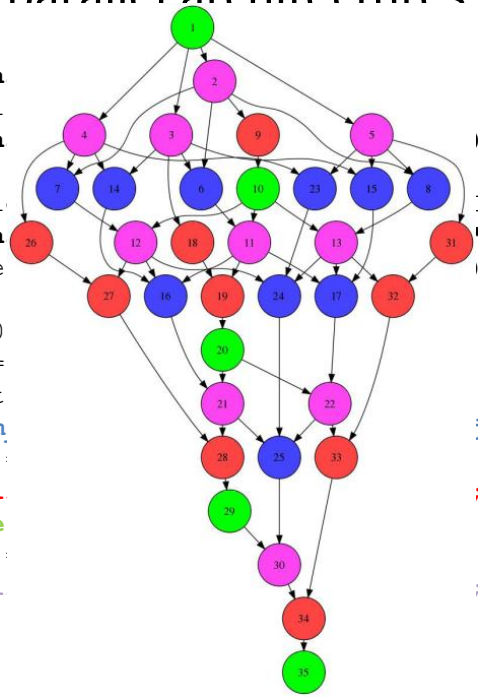
OmpSs

- Goal: programmability of heterogeneous parallel architectures
  - **Task-based** programming model
  - Simple **pragma** annotations
  - Automatic task **dependency tracking**
- Needs support for a growing number of devices, APIs and platforms

```
#pragma omp ta
void spotrf_ti
#pragma omp ta

void sgemm_til                                  C, u_long NB);
#pragma omp ta                                  *NB] C)
void syrk_tile

for (int j = 0
   for (int k=
      for (int
         sgemm                                  j], &A[j][i]);
   for (int i
      ssyrk_ti                                  ;
   spotrf_tile
   for (int i
      strsm_ti                                  ;
}
```



GMAC

- GMAC provides an easy **system-level** programming model for accelerator-based systems
  - Shared-memory model
    - Single pointer per allocation
    - No explicit copies
  - Portability across systems
    - Hardware capabilities
    - System configuration
  - Portability across platforms
    - CUDA, OpenCL
    - Windows, Linux, MacOS X

```
void vec_add(size_t N)
{
    float *a, *b, *c;
    gmacMalloc(&a, N * sizeof(float));
    gmacMalloc(&b, N * sizeof(float));
    gmacMalloc(&c, N * sizeof(float));

    // Runs on the CPU
    init_array(a, N);
    init_array(b, N);

    // Runs on the GPU
    add_kernel<<<N/512, N>>>(c, a, b, N);
}
```

GMAC

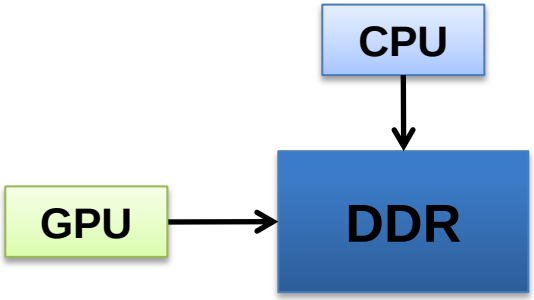
- Since last GTC, many *exciting* things have happened
  - CUDA 4
  - Fermi/Kepler
  - UVAS
  - Peer-to-peer memory transfers
  - Host threads can access any device memory (`cudaMemcpy`)
  - …
- … but most of them were already implemented in GMAC
- So we have been going forward

Outline

- **Programmability issues in CUDA systems**
- GMAC 2
- Use cases
- Conclusions

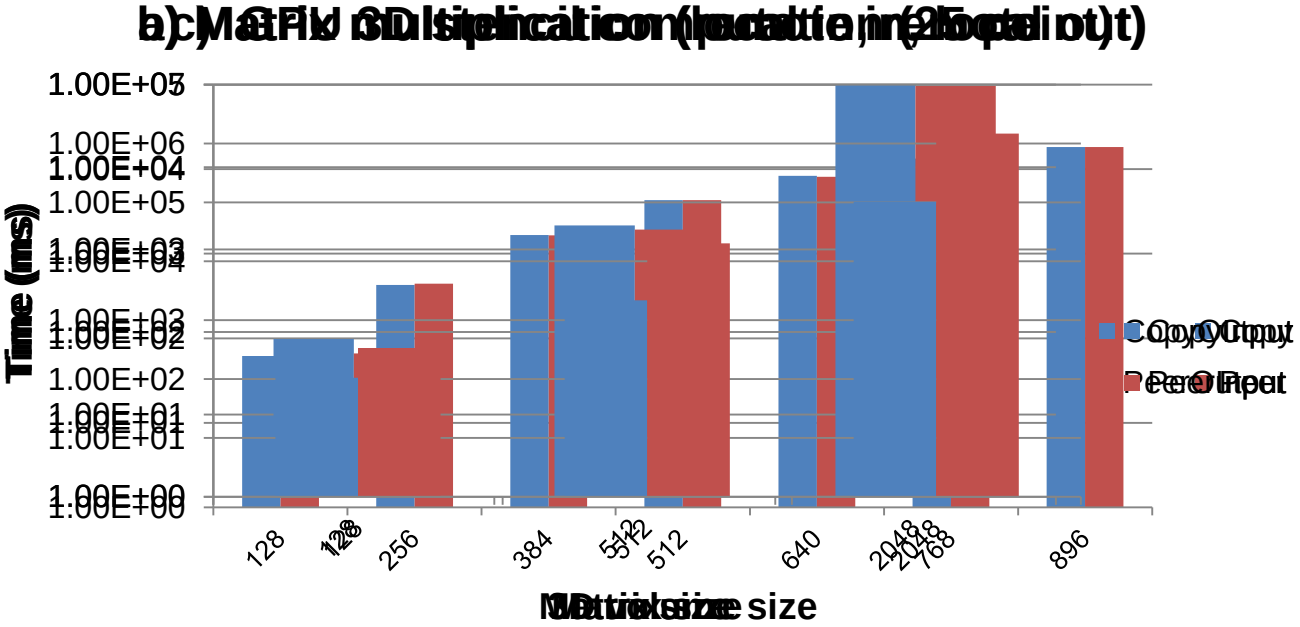
A universe of capabilities

- New memory addressing capabilities
  - Private memories
    - Copies between memories through PCIe
  - Host-mapped memory *(Compute Capability 1.1)*
    - e.g. **Output buffers** in systems with discrete GPUs
    - **Avoid unnecessary copies** in shared memory systems (e.g. ION)
  - Peer-memory access *(Compute Capability 2.0)*
    - **Avoid inter-GPU copies**
    - Only for GPUs in the same PCIe bus



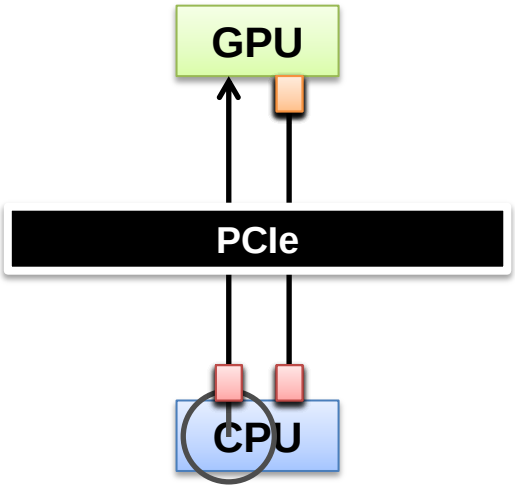
A universe of capabilities

- New memory addressing capabilities (performance)



A universe of capabilities

- Execution concurrency
  - Concurrency between GPU execution and memory transfers *(Compute Capability 1.1)*
  - Concurrent HostToDevice/DeviceToHost memory transfers *(Compute Capability 2.0)*
  - Concurrent kernel execution *(Compute Capability 2.0)*
    - Limited to back-to-back execution
    - Limited to a single CUDA context
  - Complete concurrent kernel execution *(Compute Capability 3.5)*



Access to hardware capabilities

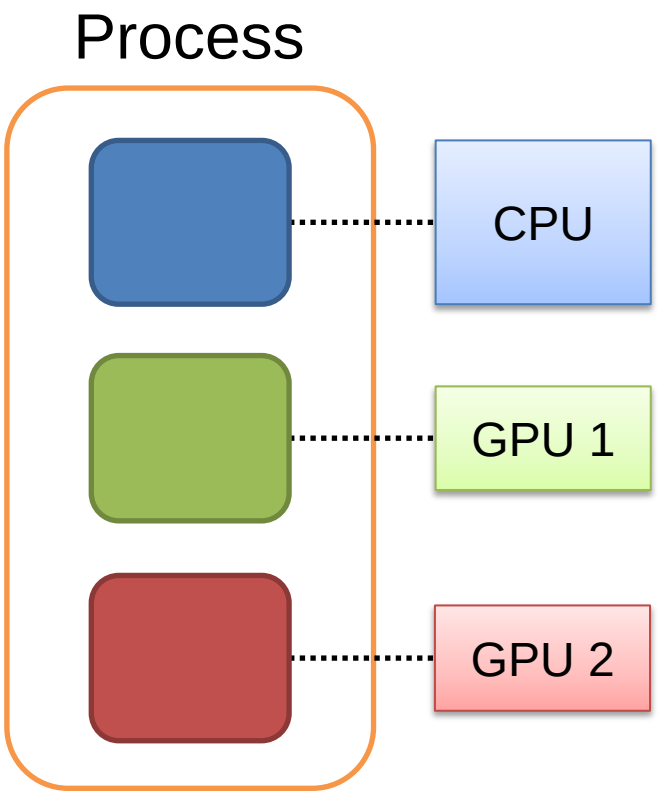
- Describing memory accessibility
  - Private memory: `cudaMalloc/malloc`
  - GPU → Host shared memory: `cudaHostRegister/cudaHostAlloc`
    - Host memory pinned and mapped on the GPUs' page tables
  - GPU↔GPU shared memory : `cudaDeviceEnablePeerAccess`
    - GPU page tables synchronized
    - Context-grain sharing (and only works for contexts on different GPUs)
- Describing dependencies among operations
  - CUDA stream: in-order queue to execute implicitly dependent operations
  - CUDA event: marker in a stream that allows to use finer-grain synchronization

Outline

- Programmability issues in CUDA systems
- **GMAC 2**
- Use cases
- Conclusions

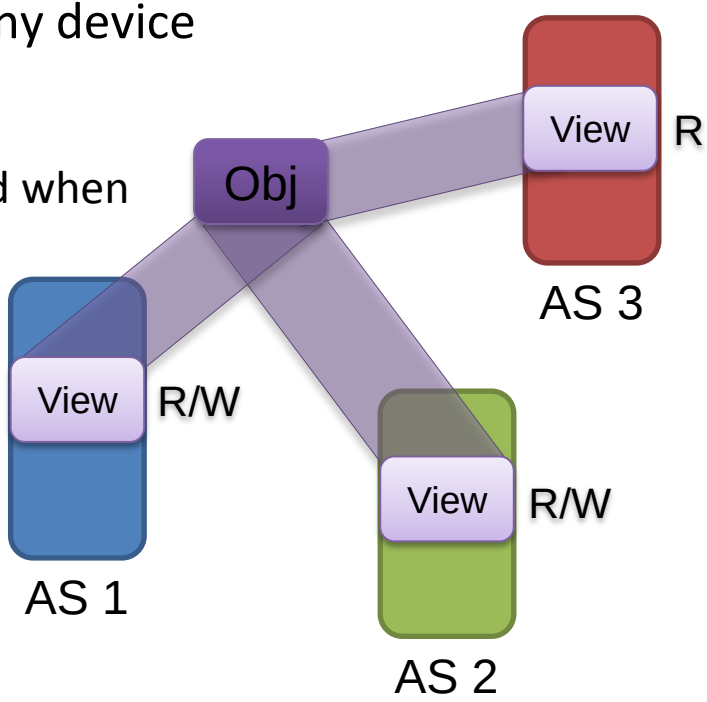
Memory model

- A process provides a **single *logical* address space**
  - Defines the memory objects that can be accessed in a program
- Implemented on top of **multiple *virtual* address spaces**
  - An address space defines which memory objects can be accessed from a device
- In contrast to the unfied *virtual* address space offered by CUDA

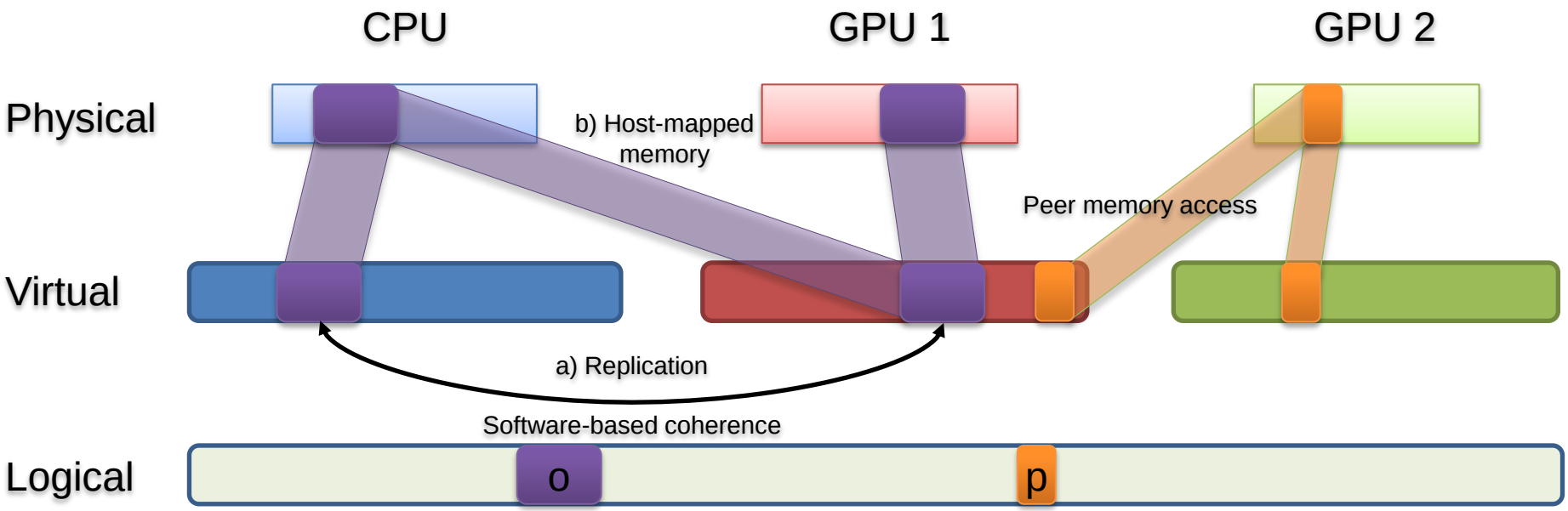


Memory model

- Memory objects can be made accessible to any device
  - Single allocation + remote access
  - Replication + software-based coherence is used when no hardware access is available
- Each **virtual address space (AS)** can have different ***views*** of an object
  - A view is created by mapping an object on a virtual address space
  - Properties of the view define the behavior of the coherence protocol (DSM)



Memory model

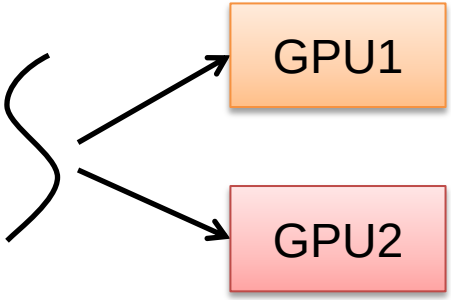


Memory model

- Automatic optimizations
  - Pinned memory management for memory transfers
  - Eager memory transfers
  - Features available in the hardware are used to optimize some scenarios
- Implementation details: restrictions
  - CUDA UVAS makes replicated objects have different virtual addresses
    - Stored pointers cannot be used on the device ☹
  - Mappings work at page granularity

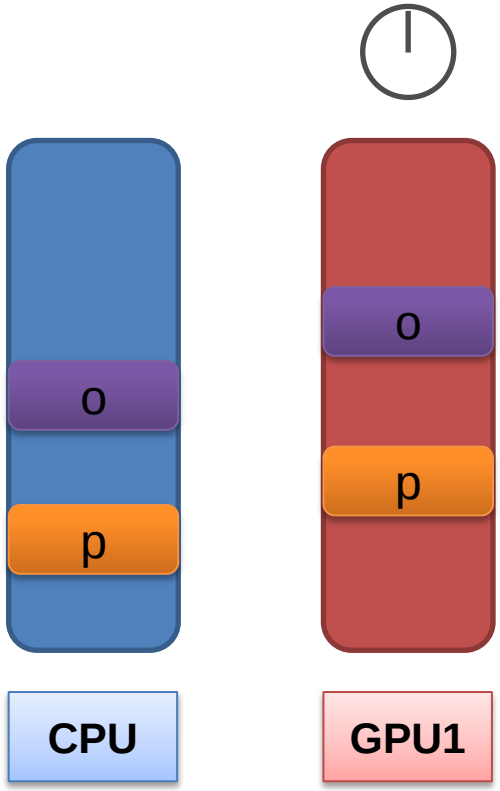
Execution model

- A process contains several execution contexts
  - The execution context (OS thread) is the basic unit of concurrency
- Each execution context is assigned a default GPU
  - Inherited on creation
  - Like CUDA 4 “current” device (`cudaSetDevice`)
- CUDA code is executed on…
  - Implicitly: using the *default* GPU
  - Explicitly: a specific GPU (using the stream parameter of the call)



Execution model

- Data accessible on a GPU is implicitly acquired/released at kernel call boundaries
  - Provides implicit synchronization on data dependencies
  - Dependencies between kernel calls are defined by the programmer using the standard inter-thread synchronization mechanisms
  - ***Explicit consistency is also available for fine tuning***



Explicit consistency

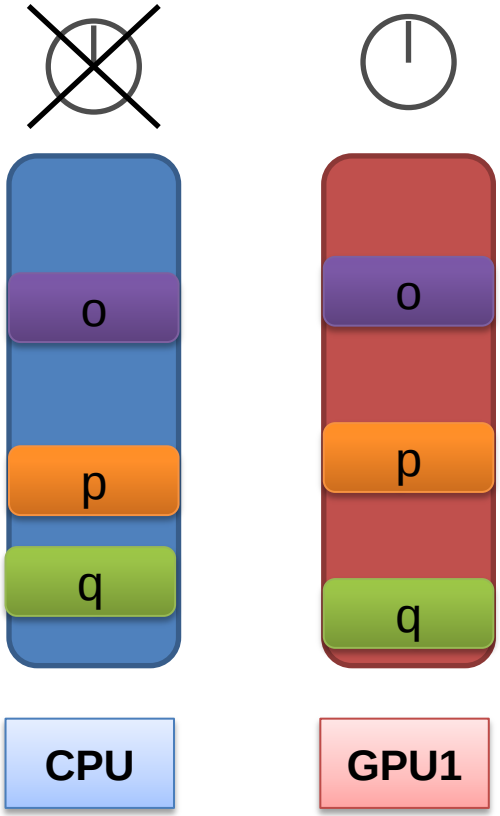
```
-> release ownership of o
-> release ownership of p

kernel<<<grid, block>>>(o, p);

for (size_t i = 0; i < SIZE_C; ++i) {
   q[i]++;
}

gmacThreadSyncronize();

-> acquire ownership of o
-> acquire ownership of p
```



Execution model

- Hides specific CUDA abstractions needed for execution concurrency (streams, events…)
- Allows concurrent code execution and memory transfers and back-to-back kernel execution
  - When triggered from different execution contexts (host threads)

GMAC 2 API

- C/C++
- More **flexible** and **composable** API than GMAC
  - Allows to specify memory access rights
  - Allows explicit consistency management
- **POSIX-like** semantics
- GMAC primitives are still provided
  - Easily implemented on top of the new API
  - Single interface for CUDA/OpenCL programs

Memory management API

```
void * gmac::map(void *ptr, size_t count, int prot,
                 int flags, device_id id)
void * gmac::map(const std::string &path, off_t offset, size_t count,
                 int prot, int flags, device_id id)
```

- `ptr == NULL` creates a new **logical** object, and creates a **view** of the object for the device `id`
- `ptr != NULL` creates a **view** of the specified object for the device `id`
- `prot` declares the kind of access to the object: RO,W,RW
- `flags` allows to specify some special properties for the view
  - e.g. `gmac::MAP_FIXED`: use the address `ptr` for the new *view*

Execution model API

`error gmac::set_gpu(device_id id)`

- Sets `id` as the default GPU to be used in implicit operations on the current execution context

`device_id gmac::get_gpu()`

- Gets the `id` of the default GPU assigned to the current execution context

Execution model API

```
error func<<<grid, block, shared, device_id>>>(...)
```

- Executes the kernel `func` on the GPU with identifier `device_id`. If no device is specified, the default one is used
- Before executing, transfers the ownership of all the GPU accessible objects to the GPU…
- … and acquires the ownership for the CPU again after kernel execution

- Alternative syntax for OpenCL based on C++11 variadic templates

```
kerne.launch::launch(config global,
                     config block,
                     config offset, device_id)(...)
```

Memory coherence/consistency API

```
error gmac::acquire(void *ptr, size_t count, int prot, device_id id)
```

- Ensures that:
  1) Device `id` sees an updated copy of the given data object
  2) Device `id` has the ownership of the object
- `prot` specifies weaker or equal access level than in `map`

```
error gmac::release(void *ptr, size_t count)
```

- Releases the ownership of the data object

Backward compatibility

- Previous calls are still available
  - `gmacMalloc`, `gmacFree`, `gmacGlobalMalloc`
- Implemented on top of the new calls
  - You can mix both (if you know what are you doing)

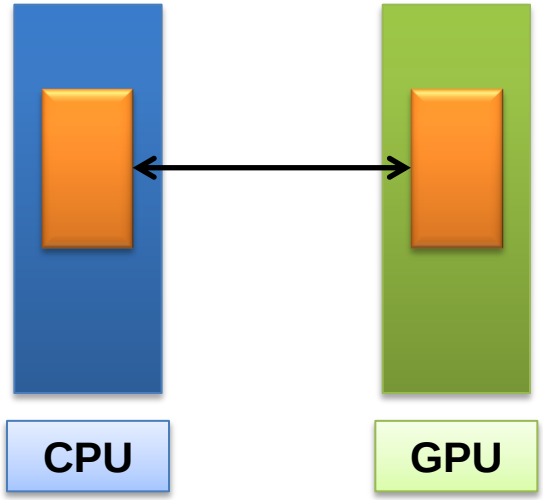
Outline

- Programmability issues in CUDA systems
- GMAC 2
- **Use cases**
- Conclusions

Simple shared allocation

```
gmacError_t gmacMalloc(void **a, size_t count)
{
  // Map on host memory
  *a = gmac::map(NULL, count,
                 gmac::PROT_READWRITE,
                 0,
                 gmac::cpu_id);

  // Map on the thread's current GPU
  gmac::map(a, count,
            gmac::PROT_READWRITE,
            gmac::MAP_FIXED,
            gmac::get_gpu());
  return gmacSuccess;
}
```

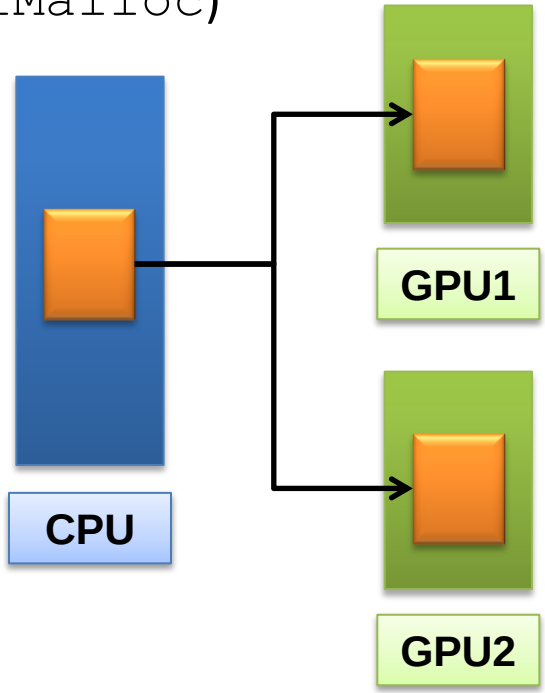


Multi-GPU data replication

- Share input data across GPUs (aka `gmacGlobalMalloc`)

```
float * a = gmac::map(NULL, count,
                      gmac::PROT_READWRITE,
                      0,
                      gmac::cpu_id);
// Map on GPU1
gmac::map(a, count,
          gmac::PROT_READ,
          gmac::MAP_FIXED, gpu1_id);

// Map on GPU2
gmac::map(a, count,
          gmac::PROT_READ,
          gmac::MAP_FIXED, gpu2_id);
```



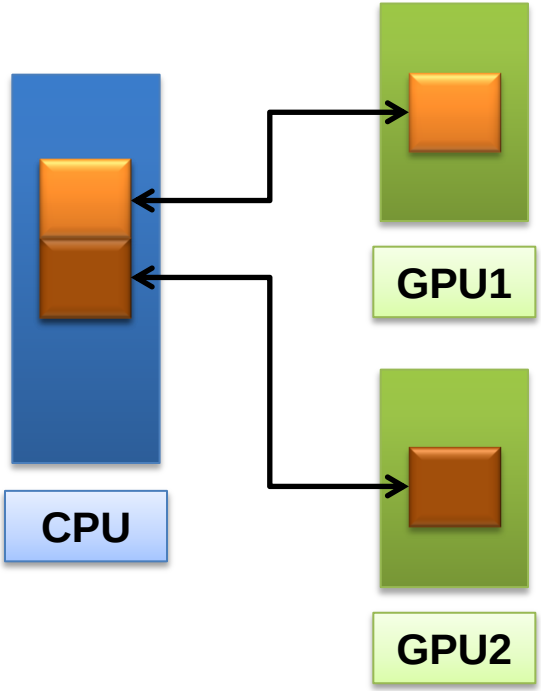
Data partitioning

- Partition data structures among GPUs

```
// Map on host memory
float * a = gmac::map(NULL, count,
                      gmac::PROT_READWRITE,
                      0,
                      gmac::cpu_id);

// Map on GPU1 memory
gmac::map(a, count/2,
          gmac::PROT_READWRITE,
          gmac::MAP_FIXED, gpu1_id);

// Map on GPU2 memory
gmac::map(a + count/2, count/2,
          gmac::PROT_READWRITE,
          gmac::MAP_FIXED, gpu2_id);
```



Peer-to-peer memory acccess

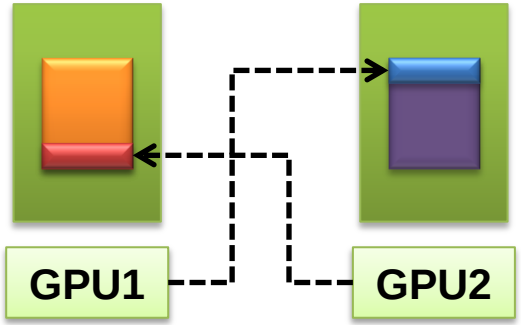
- Accessing boundaries across GPU memories (peer memory access)

```
a = gmac::map(NULL, domain_size,
              gmac::PROT_READWRITE,
              gmac::MAP_FIXED, gpu1_id);
```

```
b = gmac::map(NULL, domain_size,
              gmac::PROT_READWRITE,
              gmac::MAP_FIXED, gpu2_id);
```

```
gmac::map(b, boundary_size,
          gmac::PROT_READ,
          gmac::MAP_FIXED, gpu1_id);
```

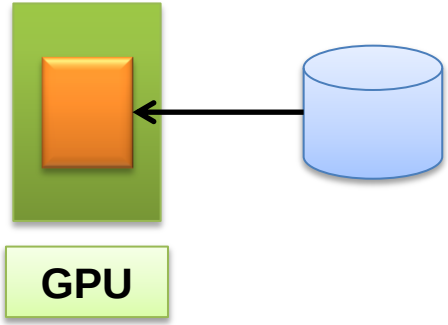
```
gmac::map(a + domain_size - boundary_size,
          boundary_size,
          gmac::PROT_READ,
          gmac::MAP_FIXED, gpu2_id);
```



File I/O

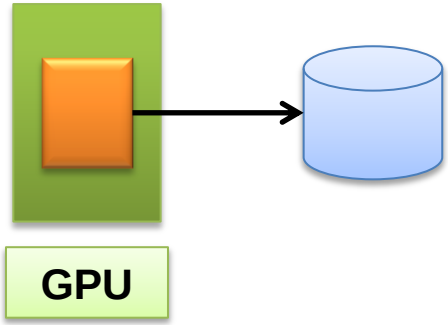
- Input from file

```
a = gmac::map("input.data", 0, count,
              gmac::PROT_READ,
              0, gpu1_id);
```



- Optimize writing output to file (host-mapped memory)

```
a = gmac::map("output.data", 0, count,
              gmac::PROT_WRITE,
              0, gpu1_id);
```



Outline

- Programmability issues in CUDA systems
- GMAC 2
- Use cases
- **Conclusions**

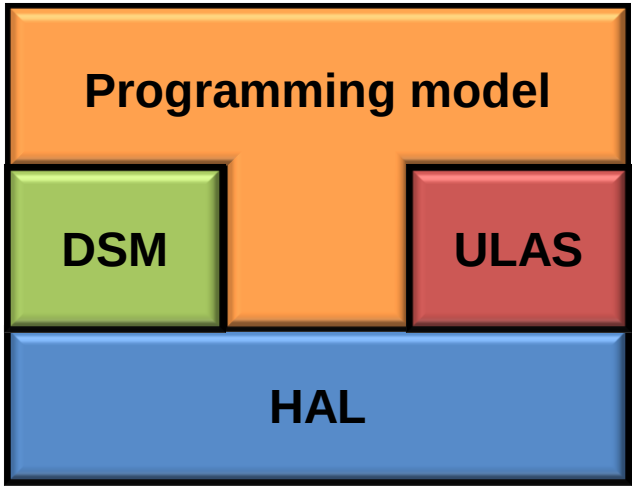
Conclusions

GMAC 2…

- Eases the programmability on CUDA-based systems
- Offers a flexible API that allows to declare complex data schemes
- Transparently exploits the capabilities of the underlying hardware
- Solves the shortcomings of the previous GMAC version
  - Backward compatibility
- We need support in the CUDA driver to fully implement replicated objects (while keeping the same memory address)

Design

- HAL (Hardware Abstraction Layer)
  - Generic hardware abstractions
- DSM (Distributed Shared Memory)
  - Coherence among address spaces using release consistency
- ULAS (Unified Logical Address Space)
  - Provides a single flat logical space for all memories
- System-level programming model



Thanks for your attention

- Questions?

Backup slides

Programmability issues in CUDA systems

Exposing hardware capabilities

- GPUs are still I/O devices in the OS
  - No enforcement of system-level policies
- No standard memory management routines
  - No shared memory across processes
  - No GPU memory swapping
  - Copies to/from I/O devices (GPU Direct is an ad-hoc solution for Infiniband)
- No standard code execution routines
  - No scheduling

Programming model

- Advanced code management and execution

```
error_t pm::load_module(void *ptr, device_id id)

error_t pm::load_module(const std::string &path,
                        device_id id)

kernel_t pm::get_kernel(const std::string &path
                        device_id id)

error_t pm::execute(kernel_t k, config c, arg_list a,
                    device_id id)
```

GMAC 2: Hardware Abstraction Layer

- Physical description layer: description of the components in the system
  - Processing units (e.g., GPUs, CPUs)
  - Physical Memories (e.g., CPU memory, GPU memory)
  - Physical address spaces
    - Aggregation of memories that are directly accessible from a processing unit
      - CUDA: GPU Direct 2, Host-mapped memory
      - OpenCL: clEnqueueMap

Hardware Abstraction Layer

- System abstractions
  - Virtual address space
    - `struct mm_struct`
  - Object: physical memory
    - Collection of `struct page`
  - View: mapping of an object in a vAS
    - `struct vm_area_struct`
  - Virtual processing unit: execution context to be executed on a processing unit
    - `struct task_struct`

Hardware Abstraction Layer

- Platform-independent
  - CUDA, OpenCL
- Fat pointers
  - Transparent inter-device copies

```
hal::copy(hal::ptr dst, hal::ptr src, size_t count)
```

  - I/O devices
- Event-based API
  - Dependency tracking and synchronization
  - Timing and tracing

Distributed Shared Memory

- Links memory areas:

```
dsm::link(hal::ptr dst, hal::ptr src, size_t count, int flags);
```

- Acquire/release concistency

```
dsm::acquire(hal::ptr p, size_t count, int flags);
dsm::release(hal::ptr p, size_t count);
```

- Depending on the conditions, optimizations are enabled
  - Eager transfer

Unified Virtual Address Space

- Since the introduction of CUDA 4.0… but!
  - Not in 32-bit machines
  - Data structures must reside in pinned memory