Streams and Events

Streams, Events, Multiple GPUs

GPU Programming

http://cuda.nik.uni-obuda.hu

Szénási Sándor
szenasi.sandor@nik.uni-obuda.hu

GPU
EDUCATION
CENTER

GPU Education Center of Óbuda University

Streams and Events

Streams

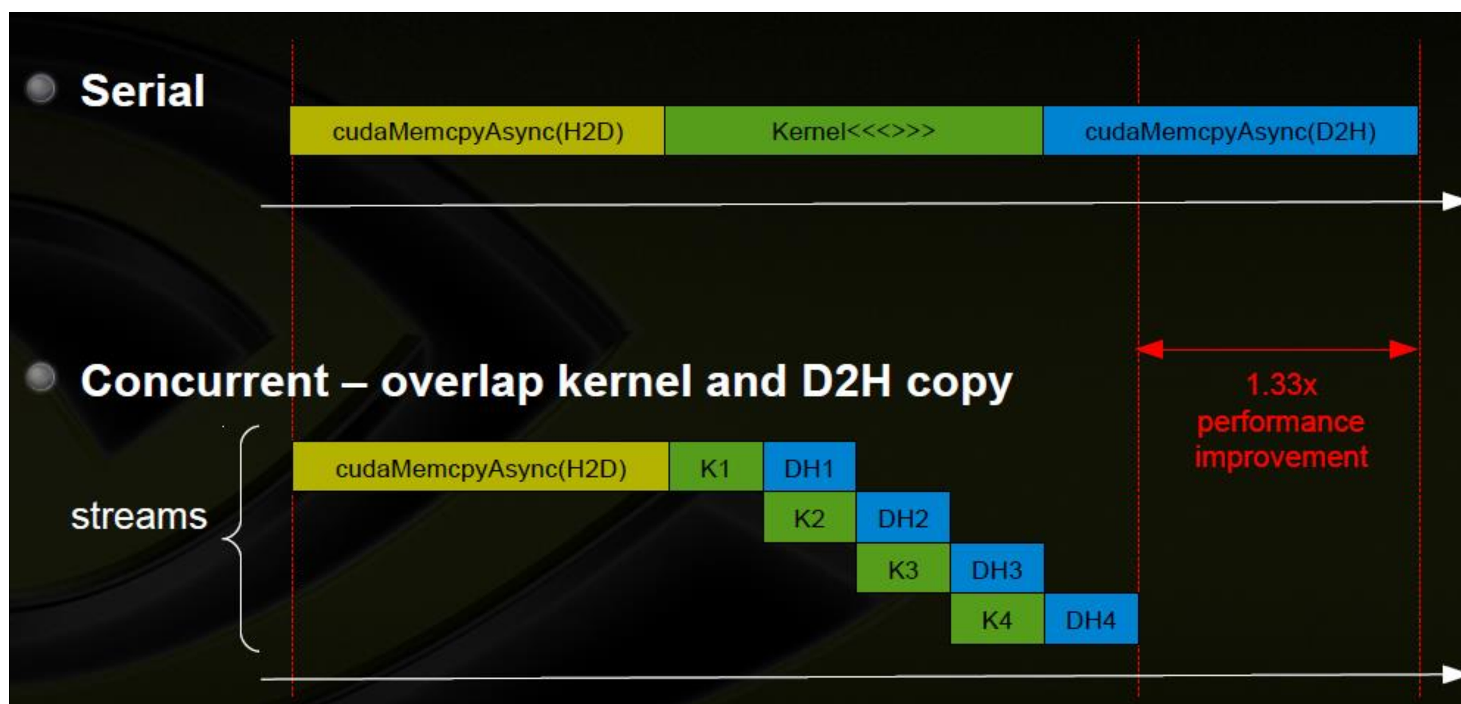
Events

Using multiple GPUs

Unified memory addressing

Streams

- Applications manage concurrency through streams
- A stream is a sequence of commands (possibly issued by different host threads) that execute in order. Different streams, on the other hand, may execute their commands out of order with respect to one another or concurrently; this behaviour is not guaranteed and should therefore not be relied upon for correctness
- Streams support concurrent execution
  - Operations in different streams may run concurrently
  - Operations in different streams may be interleaved



Creating/destroying streams

Stream object

- Stream is represented by a cudaStream_t type

Create a stream

- Function name: cudaStreamCreate
- Parameters
  - pStream – pointer to a new stream identifier
- Creates a new asynchronous stream
- Example

```
cudaStream_t stream;
cudaStreamCreate(&stream);
```

Destroy a stream

- Function name: cudaStreamDestroy
- Parameters
  - pStream - pointer to the stream to destroy
- Destroys and cleans up the asynchronous stream specified by stream
- Example

```
cudaStreamDestroy(stream);
```

Using multiple streams

Common pattern

- Create and destroy multiple streams

```
cudaStream_t stream[N];
for (int i = 0; i < N; ++i)
    cudaStreamCreate(&stream[i]);
for (int i = 0; i < N; ++i)
    cudaStreamDestroy(stream[i]);
```

Using streams

Built-in functions

- Some CUDA functions have an additional stream parameter
  - cudaError_t cudaMemcpyAsync(
    void *dst,
    const void *src,
    size_t count,
    enum cudaMemcpyKind kind,
    cudaStream_t stream = 0)
  - Kernel launch:
    Func<<< grid_size, block_size, shared_mem, **stream** >>>
- Concurrent execution may need some other requirements
  - Async memory copy to different directions
  - Page locked memory
  - Enough device resources

Default stream

- In case of missing stream parameter the CUDA runtime uses the default stream (identified by 0)
  - Used when no stream is specified
  - Completely synchronous host to device calls
  - Exception: GPU kernels are asynchronous with host by default

Using streams example

Using concurrent operations

- Example

```
cudaStream_t stream1, stream2;
cudaStreamCreate (&stream1) ;
cudaStreamCreate (&stream2) ;
...
cudaMalloc (&dev1, size ) ;
cudaMallocHost (&host1, size ) ;
cudaMalloc (&dev2, size ) ;
cudaMallocHost (&host2, size ) ;
...
cudaMemcpyAsync (dev1, host1, size, H2D, stream1 ) ;
kernel2 <<< grid, block, 0, stream2 >>> ( …, dev2, … ) ;
kernel3 <<< grid, block, 0, stream1 >>> ( …, dev1, … ) ;
cudaMemcpyAsync ( host2, dev2, size, D2H, stream2 ) ;
...
```

- All stream1 and stream2 operations will run concurrently
- Data used by concurrent operations should be independent

Stream synchronization

Synchronize everything

- Function name: cudaDeviceSynchronize
- Blocks until the device has completed all preceding requested tasks
- returns an error if one of the preceding tasks has failed
- If the cudaDeviceBlockingSync flag was set for this device, the host thread will block until the device has finished its work

Synchronize to one stream

- Function name: cudaStreamSynchronize
- Parameters
  - stream - stream to synchronize
- Blocks until stream has completed all operations
- If the cudaDeviceBlockingSync flag was set for this device, the host thread will block until the stream is finished with all of its tasks

Query stream status

- Function name: cudaStreamQuery
- Parameters
  - stream - stream to query
- Returns cudaSuccess if all operations in stream have completed, or cudaErrorNotReady if not.

Operations implicitly followed a synchronization

Implicit synchronization

- Page-locked memory allocation
  - cudaMallocHost
  - cudaHostAlloc
- Device memory allocation
  - cudaMalloc
- Non-async version of memory operations
  - cudaMemcpy
  - cudaMemset
- Change to L1/shared memory configuration
  - cudaDeviceSetCacheConfig

Stream scheduling

Hardware queues [30]

- Fermi hardware has 3 queues
  - 1 Compute Engine queue
  - 2 Copy engine queues
    - Host to device copy engine
    - Device to host copy engine

Dispatching operations

- CUDA operations are dispatched to devices in the sequence they were issued
  - Placed in the relevant queue
  - Stream dependencies between engine queues are maintained but lost within an engine queue
- CUDA operation is dispatched from the engine queue if
  - Preceding calls in the same stream have completed,
  - Preceding calls in the same queue have been dispatched, and
  - Resources are available
- CUDA kernels may be executed concurrently if they are in different streams
  - Thread blocks for a given kernel are scheduled if all thread blocks for preceding kernels have been scheduled and there still are SM resources available

Concurrency support

Compute Capability 1.0

- Support only for GPU/CPU concurrency

Compute Capability 1.1

- Supports asynchronous memory copies
  - Check asyncEngineCount device property

Compute Capability 2.0 or later

- Supports concurrent GPU kernels
  - Check concurrentKernels device property
- Supports bidirectional memory copies based on the second copy engine
  - Check asyncEngineCount device property

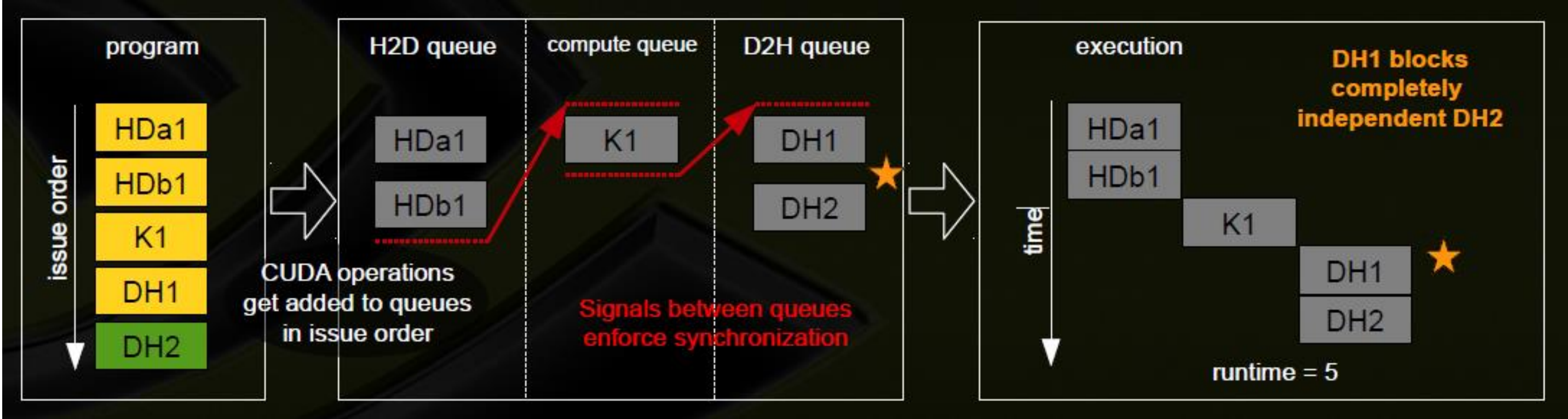
Technical note [32]

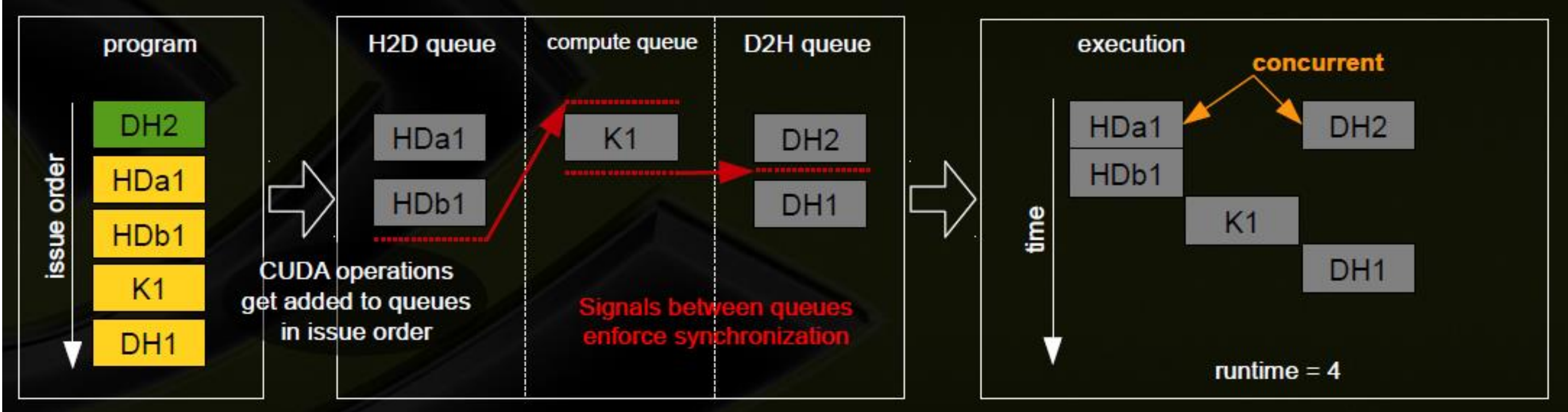
- On Windows OS using WDDM driver model work is submitted to the GPU in command buffers. Only work in the same command buffer can be executed concurrently
- **cudaStreamQuery(stream)** flushes the CUDA user mode work queue
- Each flush results in a WDDM KMD command buffer

Blocked Queue example

Two streams with the following operations

- Stream1: HDa1, HDb1, K1, DH1
- Stream2: DH2



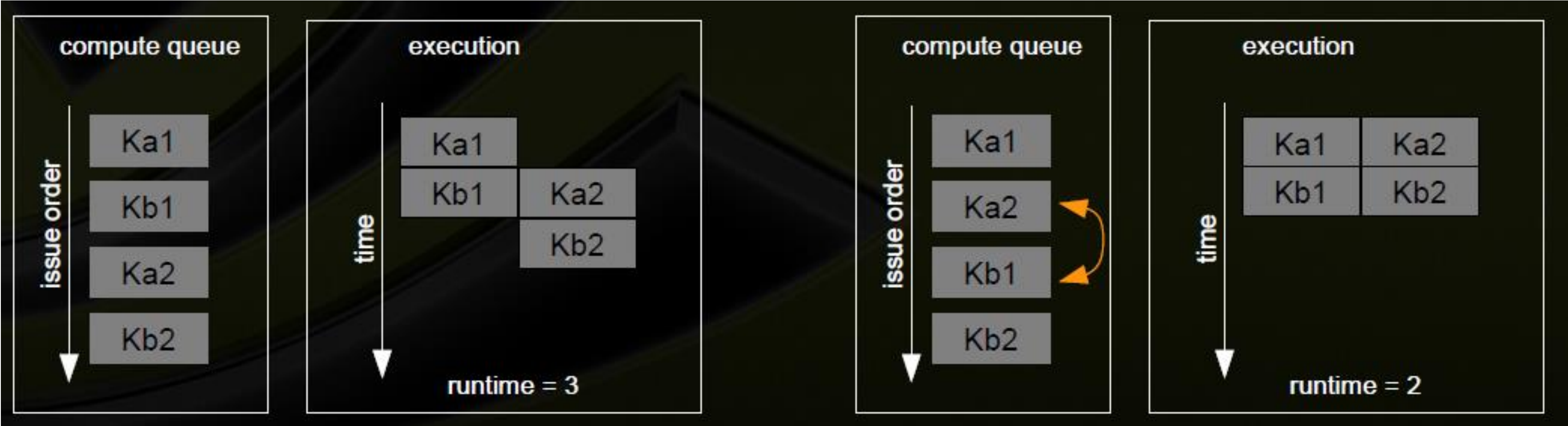


[30]

Blocked Kernel example

Two streams with the following operations

- Stream1: Ka1, Kb1
- Stream2: Ka2, Kb2



[30]

Streams and Events

- Streams
- **Events**
- Using multiple GPUs
- Unified memory addressing

Create and destroy a new event

Create an event

- Function name: cudaEventCreate
- Parameters
  - event - pointer to an event object
- Create a new event object the passed pointer will reference to this
- The result of the function is the common CUDA error code
- An example

```
cudaEvent_t test_event;
cudaEventCreate(&test_event);
```

- Variant: cudaEventCreateWithFlags

Destroy an event

- Function name: cudaEventDestroy
- Parameters
  - event - event to destroy
- Destoroys a CUDA event object
- Example:

```
cudaEvent_t test_event;
cudaEventCreate(&test_event);
cudaEventDestroy(test_event);
```

Record an event

Record event in a stream

- Function name: cudaEventRecord
- Parameters
  - event - event to record
  - stream - stream in which to record
- Records an event after all preceding operations in stream have been completed
- In case of zero stream it is recorded after all preceding operations in the entire CUDA context have been completed

Example

- Create and record an event

```
cudaEvent_t test_event;
cudaEventCreate(&test_event);
cudaEventRecord(test_event, stream);
```

Synchronize an event

Synchronize event

- Function name: cudaEventSynchronize
- Parameters
  - event - event to synchronize
- Wait until the completion of all device work preceding the most recent call to cudaEventRecord
- Waiting for an event that was created with the cudaEventBlockingSync flag will cause the calling CPU thread to block
- If the cudaEventBlockingSync flag has not been set, then the CPU thread will busy-wait until the event has been completed by the device

Example

```
cudaEvent_t test_event;
cudaEventCreate(&test_event);
cudaEventRecord(test_event, stream);
...
cudaEventSynchronize(test_event);
```

Check an event

Query event

- Function name: cudaEventQuery
- Parameters
  - event - event to query
- Query the status of all device work preceding the most recent call to cudaEventRecord()
  - If this work has successfully been completed by the device, or if cudaEventRecord() has not been called on event, then cudaSuccess is returned
  - If this work has not yet been completed by the device then cudaErrorNotReady is returned

Example

```
cudaEvent_t test_event;
cudaEventCreate(&test_event);
cudaEventRecord(test_event, stream);
…
if (cudaEventQuery(event) == cudaSuccess) {
      … event has been finished …
} else {
      … event has not been finished …
}
```

Synchronize streams with events

- Function name: cudaStreamWaitEvent
- Parameters:
  - stream - stream to wait
  - event - event to wait on
  - flags - optional parameters (must be 0)
- Makes all future work submitted to stream wait until event reports completion before beginning execution. This synchronization will be performed efficiently on the device
- The event may be from a different context than stream, in which case this function will perform cross-device synchronization
- The stream will wait only for the completion of the most recent host call to cudaEventRecord on event
- If stream is NULL, any future work submitted in any stream will wait for event to complete before beginning execution. This effectively creates a barrier for all future work submitted to the device on this thread

Synchronize events example

Synchronize streams with events

- Synchronize two streams with events

```
cudaEvent_t event;
cudaEventCreate (&event);

cudaMemcpyAsync (d_in, in, size, H2D, stream1);
cudaEventRecord (event, stream1);
cudaMemcpyAsync (out, d_out, size, D2H, stream2);

cudaStreamWaitEvent ( stream2, event );

kernel <<< , , , stream2 >>> ( d_in, d_out );

 asynchronousCPUmethod ( … )
```

Calculate elapsed time between two events

Calculate elapsed time

- Function name: cudaEventElapsedTime
- Parameters
  - ms - pointer to float
  - start - start event
  - end - end event
- Computes the elapsed time between two finished events
- Both of the events must be in finished state
- If timing is not necessary for performance use:
  cudaEventCreateWithFlags(&event, cudaEventDisableTiming)

```
cudaEvent_t start_event, end_event;
cudaEventCreate(&start_event);
cudaEventCreate(&end_event);
cudaEventRecord(start_event, 0);
kernel<<<..., ...>>>(...);
cudaEventRecord(end_event, 0);
cudaEventSynchronize(start_event);
cudaEventSynchronize(end_event);

float elapsed_ms;
cudaEventElapsedTime(&elapsed_ms, start_event, end_event);
```

- Streams
- Events
- **Using multiple GPUs**
- Unified memory addressing

CUDA 4.0 features

Share GPUs accross multiple threads

- Easier porting of multi-threaded applications. CPU threads can share one GPU (OpenMP etc.)
- Launch concurrent threads from different host threads (eliminates context switching overhead)
- New, simple context management APIs. Old context migration APIs still supported

One thread can access all GPUs

- Each host thread can access all GPUs
  (CUDA had a „1 thread – 1 GPU" limitation before)
- Single-threaded application can use multi-GPU features
- Easy to coordinate more than GPUs

Set current device

Current device

- All CUDA operations are issued to the “current” GPU (except asynchronous P2P memory copies)
- Any device memory subsequently allocated from this host thread using cudaMalloc, cudaMallocPitch or cudaMallocArray will be physically resident on device
- Any host memory allocated from this host thread using cudaMallocHost or cudaHostAlloc or cudaHostRegister will have its lifetime associated with device
- Any streams or events created from this host thread will be associated with device
- Any kernels launched from this host thread using the <<< >>> operator or cudaLaunch will be executed on device

Select current device

- Function name: cudaSetDevice
- Parameters
  - device - device number
- This call may be made from any host thread, to any device, and at any time
- This function will do no synchronization with the previous or new device, and should be considered a very low overhead call

Current device - streams, events

Allocations

- Streams and events are per device
  - Streams are created in the current device
  - Events are created in the current device
- NULL stream (or 0 stream)
  - Each device has its own default stream
  - Default streams of different devices are independents
- Using streams and events
  - Streams can contain only events of the same device
- Using current device
  - Calls to streams are available only when the appropriate device is current

Synchronization between devices

- eventB belongs to streamB and device 1
- At cudaEventSynchronize the current GPU is device 0

```
cudaStream_t streamA, streamB;
cudaEvent_t eventA, eventB;

cudaSetDevice(0);
cudaStreamCreate(&streamA);
cudaEventCreate(&eventA);

cudaSetDevice(1);
cudaStreamCreate(&streamB);
cudaEventCreate(&eventB);

kernel<<<..., ..., streamB>>>(...);
cudaEventRecord(eventB, streamB);

cudaSetDevice(0);
cudaEventSynchronize(eventB);
kernel<<<..., ..., streamA>>>(...);
```

Using multiple CPU threads

In case of multiple CPU threads of the same process

- GPU handling is the same as single-thread environment
- Every thread can select the current device
- Every thread can communicate to any GPUs
- The process has its own address space, all of the threads can reach this region

In case of multiple processes

- Processes have their own memory address spaces
- It's like the processes are on different nodes
- Therefore some CPU side messaging needed (MPI)

In case of different nodes

- The CPUs have to solve the communication
- From the GPUs perspective it is the same as the single-node environment

Vector multiplication with multiple GPUs - kernel

Simple kernel to multiply all items in the array by 2

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>

#define N 100
#define blockN 10
#define MaxDeviceCount 4

__global__ static void VectorMul(float *A, int NperD) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < NperD) {
        A[i] = A[i] *2;
    }
}
```

Vector multiplication with multiple GPUs – memory allocation

Get information about devices and allocate memory in all devices

```
int main(int argc, char* argv[])
    int deviceCount;
    cudaGetDeviceCount(&deviceCount);
    printf("Available devices:\n");
    cudaDeviceProp properties[MaxDeviceCount];
    for(int di = 0; di < deviceCount; di++) {
      cudaGetDeviceProperties(&properties[di], di);
      printf("'%d' - %s\n", di, properties[di].name);
    }

    float A[N], oldA[N];
    for(int i = 0; i < N; i++) {
      A[i] = I; oldA[i] = A[i];
    }

    int NperD = N / deviceCount;
    float* devA[MaxDeviceCount];
    for(int di = 0; di < deviceCount; di++) {
      cudaSetDevice(di);
      cudaMalloc((void**) &devA[di], sizeof(float) * NperD);
    }
```

Kernel invocation

- Select one of the devices
- Copy the appropriate part of the input array (asynchronously)
- Start a kernel in the selected device
- Copy back the results to the host memory (asynchronously)
- Do the iteration before for all devices
- After this synchronize all devices

```
for(int di = 0; di < deviceCount; di++) {
    cudaSetDevice(di);
    cudaMemcpy(devA[di],&A[di*NperD],sizeof(float)*NperD,cudaMemcpyHostToDevice);

    dim3 grid((NperD - 1) / blockN + 1);
    dim3 block(blockN);
    VectorMul<<<grid, block>>>(devA[di], NperD);

    cudaMemcpy(&A[di * NperD], devA[di], sizeof(float) * NperD,
    cudaMemcpyDeviceToHost);
}
cudaThreadSynchronize();
```

Free context

- Free all memory objects in devices
- Print out the resuls

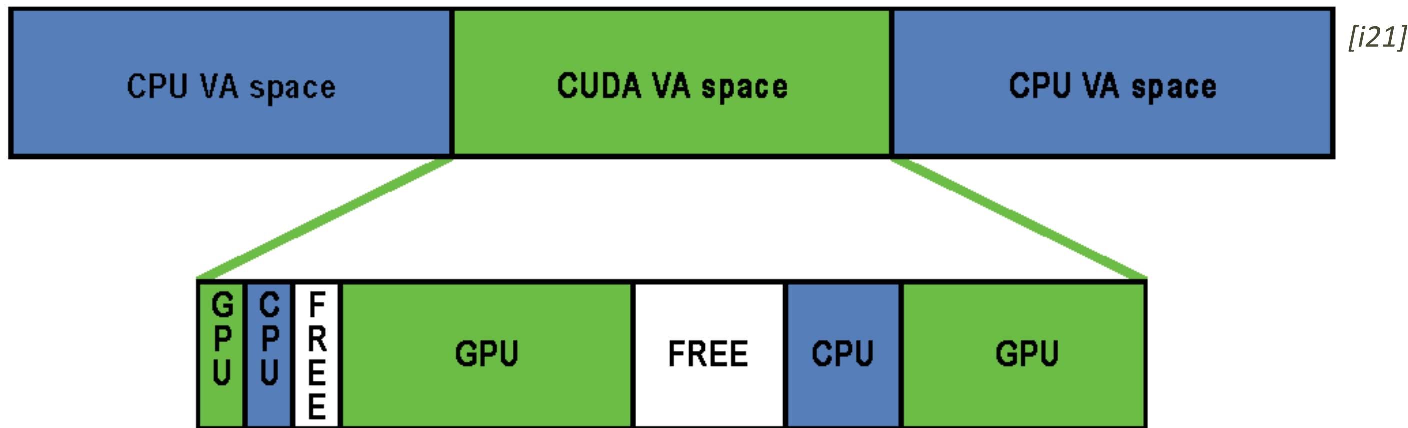
```
    for(int di = 0; di < deviceCount; di++) {
        cudaFree(devA[di]);
    }

    for(int i = 0; i < N; i++) {
        printf("A[%d] = \t%f\t%f\n", i, oldA[i], A[i]);
    }
}
```

- Streams
- Events
- Using multiple GPUs
- **Unified memory addressing**

CUDA Unified Virtual Address Management

Unified Virtual Addressing

- Unified virtual addressing (UVA) is a memory address management system enabled by default in CUDA 4.0 and later releases on Fermi and Kepler GPUs running 64-bit processes.
- The design of UVA memory management provides a basis for the operation of RDMA for GPUDirect



In the CUDA VA space, addresses can be:

- GPU – page backed by GPU memory. Not accessible from the host
- CPU – page backed by CPU memory. Accessible from the host and the GPU
- Free – reserved for future CUDA allocations

Unified Virtual Address Space

- UVA means that a single memory address is used for the host and all the devices
- CPU and GPU use the same unified virtual address space
  - the driver can determine from an address where data resides (CPU, GPU, one of the GPUs)
  - allocations still reside on the same device (in case of multi-GPU environments)
- A pointer can reference an address in
  - global memory on the GPU
  - system memory on the host
  - global memory on another GPU

Availability

- Available from
  - CUDA 4.0 or later
  - Compute Capability 2.0 or later
  - 64bit operation system
- Applications may query if the unified address space is used for a particular device by checking that the unifiedAddressing device property

Pointer attributes

Get pointer attributes

- Function name: cudaPointerGetAttributes
- Parameters
  - attr - a cudaPointerAttributes object
  - ptr - pointer
- Which memory a pointer points to – host memory or any of the device memories – can be determined from the value of the pointer using cudaPointerGetAttributes

```
void* A;
cudaPointerAttributes attr;
cudaPointerGetAttributes(&attr, A);
```

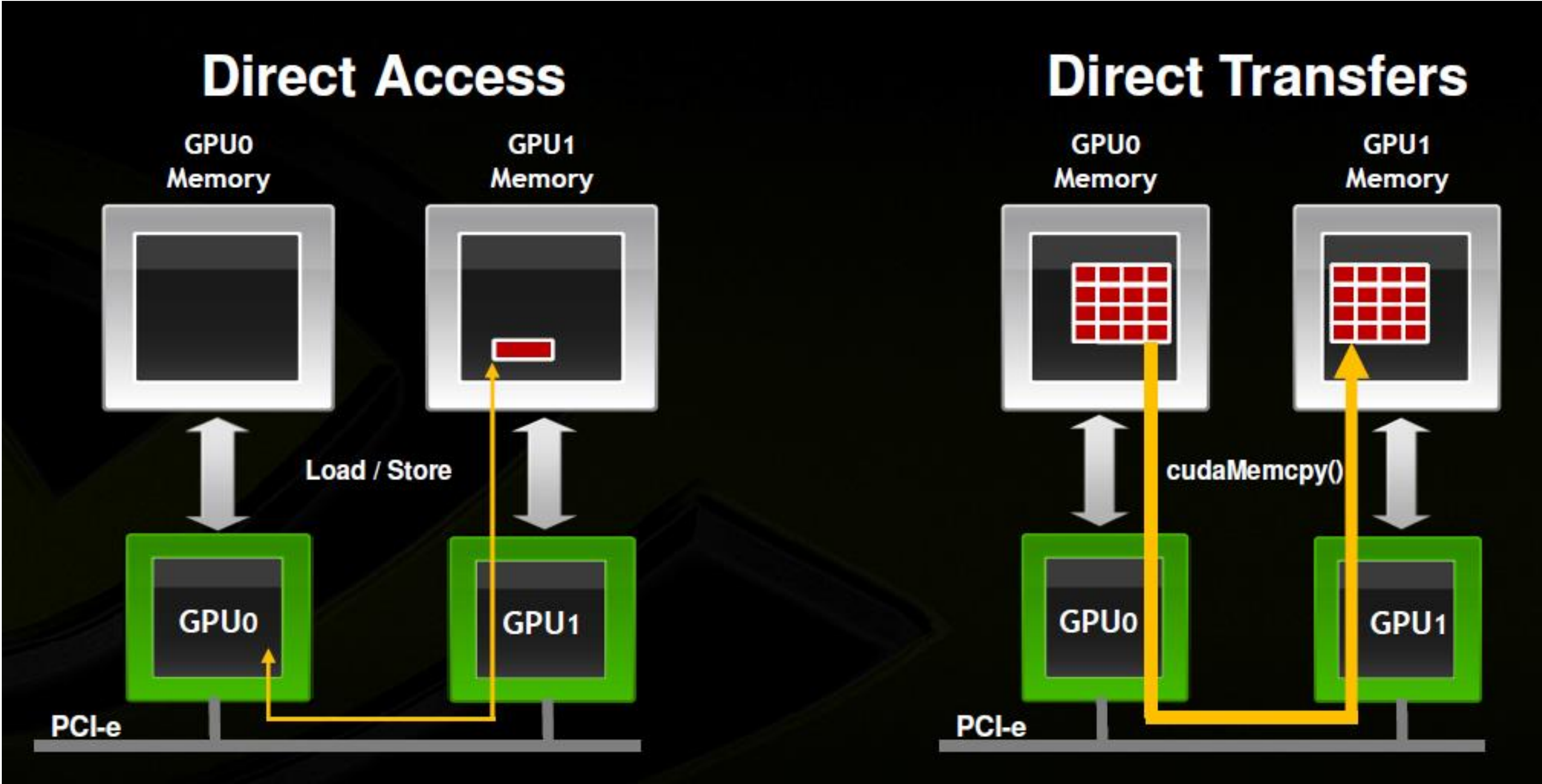
cudaPointerAttributes structure

- memoryType identifies the physical location of the memory associated with pointer ptr. It can be cudaMemoryTypeHost for host memory or cudaMemoryTypeDevice for device memory
- device is the device against which ptr was allocated
- devicePointer is the device pointer alias through which the memory referred to by ptr may be accessed on the current device
- hostPointer is the host pointer alias through which the memory referred to by ptr may be accessed on the host

Peer to peer communication between devices

Access memory of other devices

- UVA memory copy
- P2P memory copy
- P2P memory access



[31]

Using Unified Addressing and P2P transfer

Memory access

- All host memory allocated using cudaHostAlloc is directly accessible from all devices that support unified addressing
- The pointer value is the same in the host and in the device side, so it is not necessary to call any additional functions
- Enables libraries to simplify their interfaces
- Note that this will transparently fall back to a normal copy through the host if P2P is not available

Memory copy

- All the pointers are unique, so it is not necessary to specify information about pointers to cudaMemCpy or any other copy functions. The cudaMemCpy functions needs a parameter about transfer direction, it would be cudaMemcpyDefault. The runtime will know the location of the pointer from its value

cudaMemcpyHostToHost
cudaMemcpyHostToDevice
cudaMemcpyDeviceToHost
cudaMemcpyDeviceToDevice

→ cudaMemcpyDefault

Using Unified Addressing and P2P transfer

Memory access

- All host memory allocated using cudaHostAlloc is directly accessible from all devices that support unified addressing
- The pointer value is the same in the host and in the device side, so it is not necessary to call any additional functions
- Enables libraries to simplify their interfaces
- Note that this will transparently fall back to a normal copy through the host if P2P is not available

Memory copy

- All the pointers are unique, so it is not necessary to specify information about pointers to cudaMemCpy or any other copy functions. The cudaMemCpy functions needs a parameter about transfer direction, it would be cudaMemcpyDefault. The runtime will know the location of the pointer from its value

cudaMemcpyHostToHost
cudaMemcpyHostToDevice
cudaMemcpyDeviceToHost
cudaMemcpyDeviceToDevice

→ cudaMemcpyDefault

Enable peer access

Check peer access

- Function name: cudaDeviceCanAccessPeer
- Parameters
  - canAccessPeer - pointer to an int
  - device - device id from the allocation are accessed
  - peerDevice - device on which the allocations to be directyle accessed
- Returns in *canAccessPeer a value of 1 if device device is capable of directly accessing memory from peerDevice and 0 otherwise

Enable peer access

- Function name: cudaDeviceEnablePeerAccess
- Parameters
  - peerDevice - device id
  - flags - optional flags
- Enables registering memory on peerDevice for direct access from the current device
- If both the current device and peerDevice support unified addressing then all allocations from peerDevice will immediately be accessible by the current device upon success

Disable peer access

- Function name: cudaDeviceDisablePeerAccess
- Parameters
  - peerDevice - device id
- Disables registering memory on peerDevice for direct access from the current device

Peer-to-peer memory transfer between GPUs

Check for P2P access between GPUs [10]:

```
cudaDeviceCanAccessPeer(&can_access_peer_0_1, gpuid_0, gpuid_1);
cudaDeviceCanAccessPeer(&can_access_peer_1_0, gpuid_1, gpuid_0);
```

Enable peer access between GPUs:

```
cudaSetDevice(gpuid_0);
cudaDeviceEnablePeerAccess(gpuid_1, 0);
cudaSetDevice(gpuid_1);
cudaDeviceEnablePeerAccess(gpuid_0, 0);
```

We can use UVA memory copy:

```
cudaMemcpy(gpu0_buf, gpu1_buf, buf_size, cudaMemcpyDefault)
```

Stop peer access:

```
cudaSetDevice(gpuid_0);
cudaDeviceDisablePeerAccess(gpuid_1);
cudaSetDevice(gpuid_1);
cudaDeviceDisablePeerAccess(gpuid_0);
```

Peer-to-peer memory access kernel

Access memory region of other GPU

- Needs the same initialization steps
- Well known kernel copy an array from destination to target:

```
__global__ void CopyKernel(float *src, float *dst)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    dst[idx] = src[idx];
}
```

- We can start a kernel with different parameters:

```
CopyKernel<<<blocknum, threadnum>>>(gpu0_buf, gpu0_buf);
CopyKernel<<<blocknum, threadnum>>>(gpu1_buf, gpu1_buf);
CopyKernel<<<blocknum, threadnum>>>(gpu1_buf, gpu0_buf);
CopyKernel<<<blocknum, threadnum>>>(gpu0_buf, gpu1_buf);
```

- Due to UVA the kernel knows whether its argument is from another GPU memory/host memory/local memory

CUDA Unified Virtual Address summary

Overview

- Faster memory transfers between devices
- Device to device memory transfers with less host overhead
- Kernels in a device can access memory of other devices (read and write)
- Memory addressing on different devices (other GPUs, host memory)
- Requirements
  - 64bit OS and application (Windows TCC)
  - CUDA 4.0
  - Fermi GPU
  - Latest drivers
  - GPUs need to be on same IOH

More information about UVA

- CUDA Programming Guide 4.0
  - 3.2.6.4 Peer-to-Peer Memory Access
  - 3.2.6.5 Peer-to-Peer Memory Copy
  - 3.2.7 Unified Virtual Address Space