Asynchronous CUDA Techniques

for

Overlapping Computing and Data Transfer

Piotr Luszczek

Virtual Memory: Page-locked / Pinned Pages

- C library's `malloc()` is used to allocate memory in the host
  - But `malloc()` only allocates pageable host memory pages
- Call to `cudaHostAlloc()` allocates pages from page-locked pool:

```
cudaHostAlloc( (void**)&a, size * sizeof(*a),
               cudaHostAllocDefault );
cudaFreeHost ( a );
```

- Page-locked memory pages stay fixed in the physical memory
  - OS does not evict them onto disk as is the case for regular virtual memory pages
  - They are locked to their physical location

Paged Memory Properties

- Copying content of pageable memory to GPU
  - CPU copies data from pageable memory to a page-locked memory
  - GPU uses direct memory access (DMA) to move the data to/from the host's page-locked memory (copy operation is done twice when using C's malloc allocator)
- When using page-locked memory (from `cudaHostAlloc`), the first copy is skipped
- Page-locked memory allows faster copies but uses physical memory pages and cannot be swapped to disk
  - Total number of pinned pages is limited
    - OS may run out of pinned memory much quicker than regular memory
  - MPI uses pinned pages for communicating
    - Network card uses DMA to push data onto the interconnect

Introduction to CUDA Streams

- Streams introduce task-based parallelism to CUDA codes
  - Kernel launches and CPU-GPU communication are the tasks
- Streams play an important role in adding overlap to CUDA-accelerated the applications
- CUDA stream represents a queue-like functionality for GPU operations that will be executed in the order of insertions (FIFO):
  - The order in which the operations are added to a stream specifies the order in which they will be executed
- GPU hardware helps in scheduling operations enqueued in streams
  - Fermi required clever ordering to achieve overlap
  - Kepler introduced HyperQ
    - 4 hardware queues to execute stream operations and ease the programming effort

Using One CUDA Streams

- First, check if the device supports the 'device overlap' property
- Call `cudaGetDeviceProperties()` to check for device overlap:

```
cudaDeviceProp prop;
int whichDevice;
cudaGetDevice( &devID );
cudaGetDeviceProperties( &prop, dev_id );
if (0 == prop.deviceOverlap) {
  fprintf(stderr, "No handle overlap support");
}
```

- GPU with overlap capability (Fermi and above) may execute a kernel while performing a memory copy between to/from host

Using One CUDA Stream – contd.

- cudaStreamCreate() creates a new stream:

```
// initialize the stream and create the stream
cudaStream_t  stream;
cudaStreamCreate( &stream );
```

- Allocate the memory on the host and GPU

```
//pagelocked memory at GPU
cudaMalloc( (void**)&dev_a, N*sizeof(int) ) );
// allocate page-locked memory
cudaHostAlloc( (void**)&host_a, N*sizeof(int),
               cudaHostAllocDefault );
```

- CudaMemcpyAsync() copies data to/from CPU to GPU. The copy continues after the call returns – completion happens later.

```
cudaMemcpyAsync( dev_a, host_a, N*sizeof(int),
                 cudaMemcpyHostToDevice, stream );
```

Using One CUDA Stream – prolog

- Sample kernel launch

```
kernel<<<N/256,256,0,stream>>>(dev_a,dev_b,dev_c);
```

- copy back data from device to locked memory

```
cudaMemcpyAsync( host_c, dev_c, N*sizeof(int),
                 cudaMemcpyDeviceToHost, stream);
```

- Stream synchronization waits for the stream finish all operations

```
cudaStreamSynchronize(stream);
```

- Free the memory allocated and destroy the stream after synchronization:

```
cudaFreeHost(host_a);
cudaFree(dev_a);
cudaStreamDestroy(stream);
```

Using More than one Streams

- Kernel execution and memory copies can be performed concurrently as long as:
  - they are in multiple streams and,
  - hardware supports it
- Some GPU architectures support concurrent memory copies if they are in opposite directions
- The concurrency when multiple streams are helps with:
  - Improving performance
  - Using PCIexpress bus more efficiently
  - Will also help with NVLink

Execution Time Line for 2 Streams

Stream 0	Stream 1
Memcpy A to GPU	
Memcpy B to GPU	
kernel	Memcpy A to GPU
	Memcpy B to GPU
Memcpy C from GPU	kernel
	Memcpy C from GPU
Memcpy A to GPU	
Memcpy B to GPU	
kernel	Memcpy A to GPU
	Memcpy B to GPU
Memcpy C from GPU	kernel
	Memcpy C from GPU

Time

GPU Work Scheduling

- GPU hardware has no notion of streams
- GPU hardware has separate queues (engines) to perform memory copies and to execute kernels
- The commands in the queue are like dynamically scheduled tasks
  - It is possible to create dependent tasks in different streams
- Hyper-Q introduced in Kepler makes streams much useful in practice
- The default (NULL) stream is always blocking, even for other streams
  - Operations in the default stream prevent other streams from proceeding with kernel launches
  - But the kernel launches on any stream are asynchronous
    - Use cudaDeviceSynchronize() to wait for launched kernels to finish on the current device

Enumerating, Querying, and Using Multiple GPUs

- Management with multiple GPUs is done with global state
  - To obtain the total count call: `cudaGetDeviceCount(int *num_gpus)`
    - GPUs are number from 0 to num_gpus-1
  - Call `cudaGetDeviceProperties()` to select the best device if you need the best one
    - for example with the largest memory
  - Use `cudaSetDevice(gpu_id)` to select the GPU for the subsequent CUDA calls
- More information may be obtained via command line “nvidia-smi -L”
  - The command deviceQuery may not be installed
  - nvidia-smi allows selection of what to display with -d switch
    - ACCOUNTING, CLOCK, MEMORY, ECC, TEMPERATURE, POWER, COMPUTE, PIDS, PERFORMANCE, SUPPORTED_CLOCKS, PAGE_RETIREMENT, UTILIZATION
  - System administrator may restrict the use of nvidia-smi
- CUDA access to GPUs may be restricted with an environment variable
  - `CUDA_VISIBLE_DEVICES=3,5,7`

Memory Management with Multiple GPUs

- CUDA memory allocation works on the current device
  - Switch to a different device to switch target device for allocations
- GPUs may be able to copy data between themselves without CPU
  - This may be disabled programmatically or by the system administrator
  - Use cudaDeviceCanAccessPeer() to if peer access is enabled
  - Use cudaDeviceEnablePeerAccess() to enable it
  - Use cudaDeviceDisablePeerAccess() to disable it
  - Use cudaMemcpyPeerAsync() to copy between GPUs
    - Requires a stream parameter
    - May leverage NVLink speed
- Synchronization between multiple GPUs may be achieved through streams with synchronizing events

Programming CUDA Streams with Synchronization

Piotr Luszczek

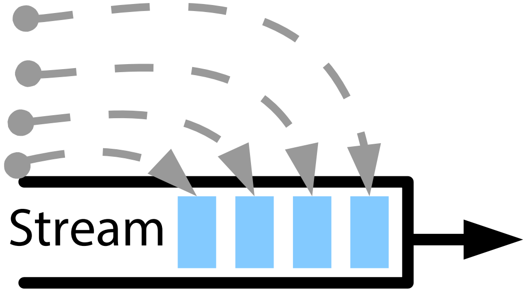
Main Goal of Streams and Asynchronous Interface

- Work around inefficiency of CPU-GPU data transfers
  - For PCIexpress, this means working around the low bandwidth
  - For NVLink, this means utilizing multiple links at the same time
  - For DMA, it means using transfer and compute hardware simultaneously
    - CUDA does not detect dependences between tasks automatically
    - The user must specify dependences manually between different streams
- Proper code organization is required to avoid many pitfalls and bugs arising in complex asynchronous transfers
- Keep in mind that switching between tasks in streams is much more expensive than switching threads, blocks, and grids on the GPU

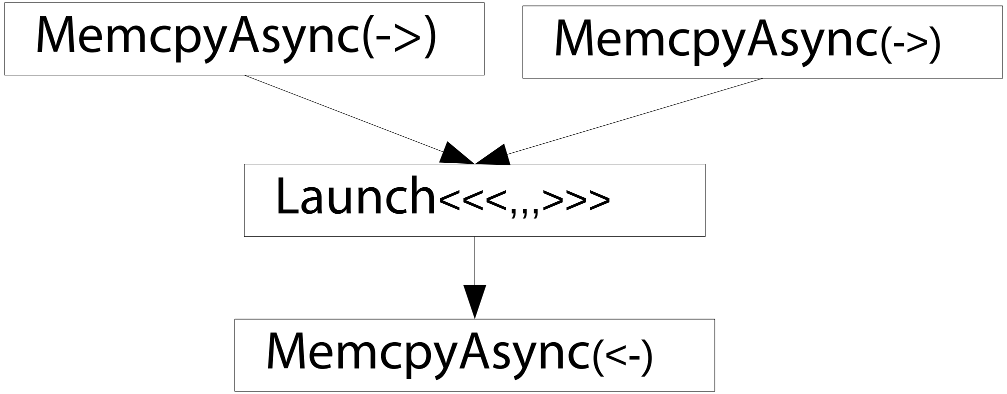
Representing Multiple Streams as Task Graph

Stream/queue view:

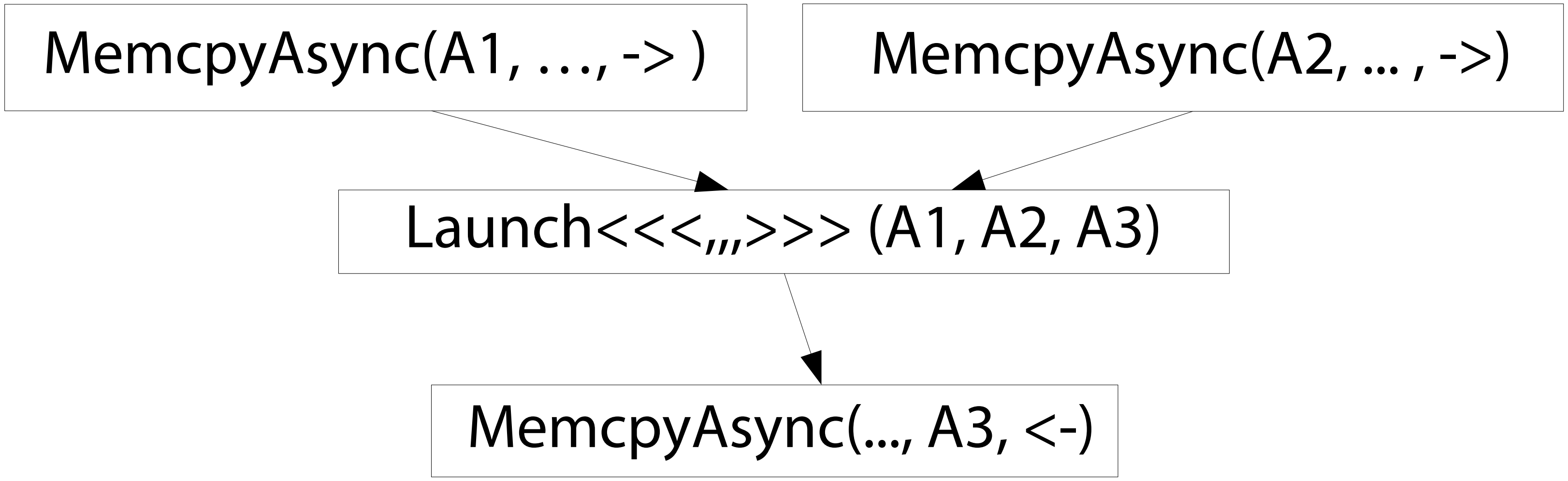
cudaMemcpyAsync(->)
cudaMemcpyAsync(->)
Launch<<< >>>()
cudaMemcpyAsync(<-)



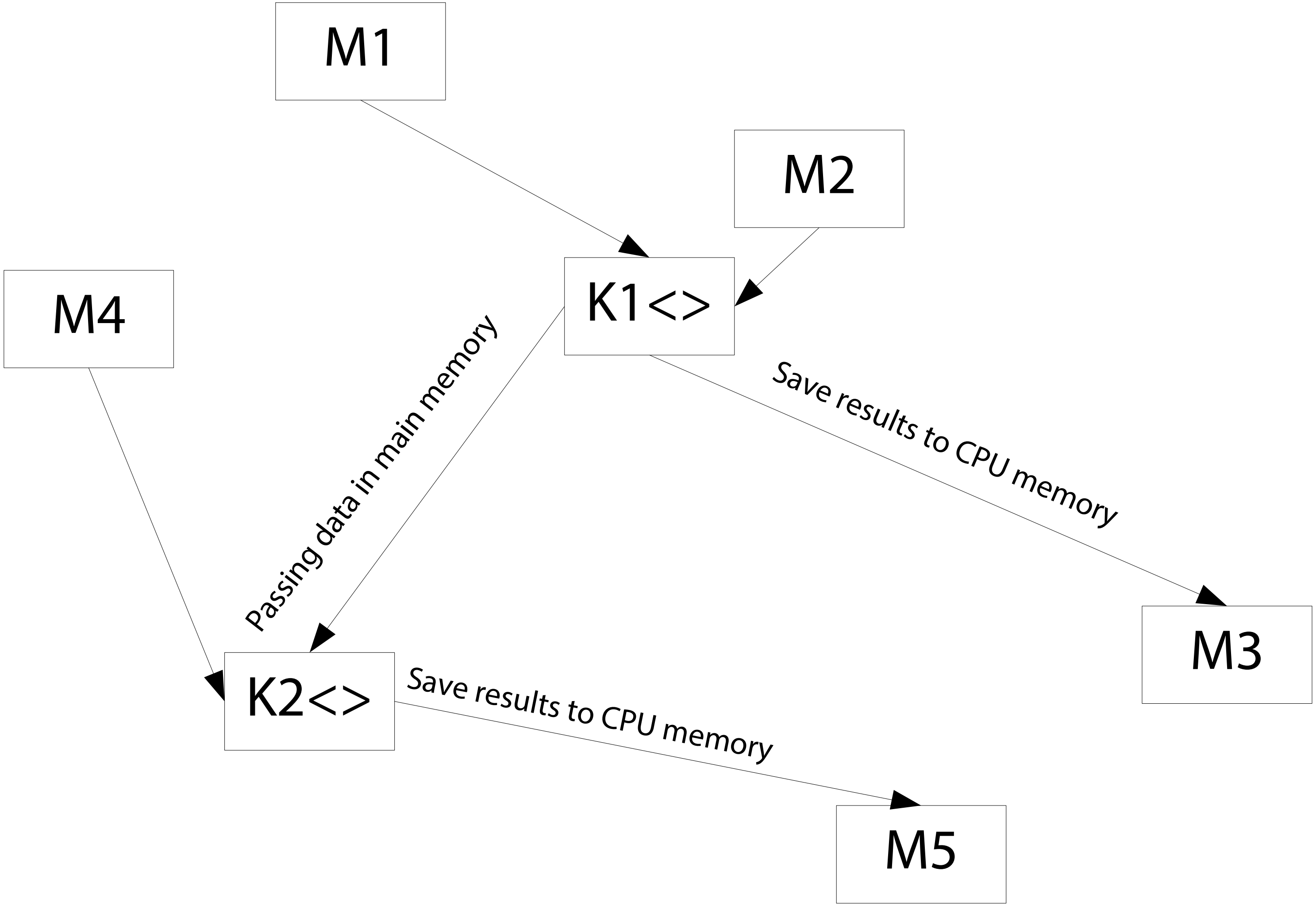
Direct Acyclic Graph (DAG):



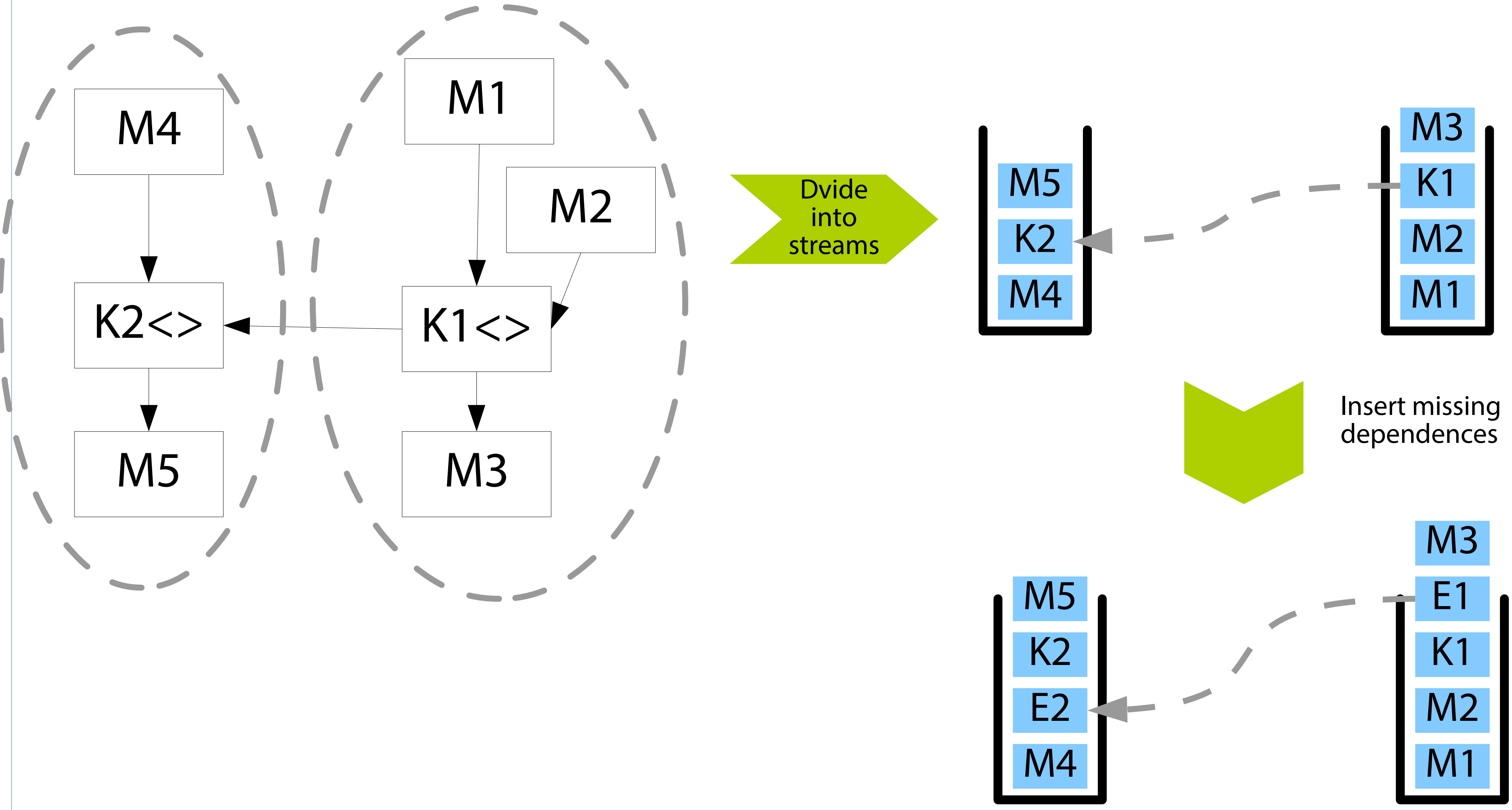
Dependences Between Tasks are Based on Data



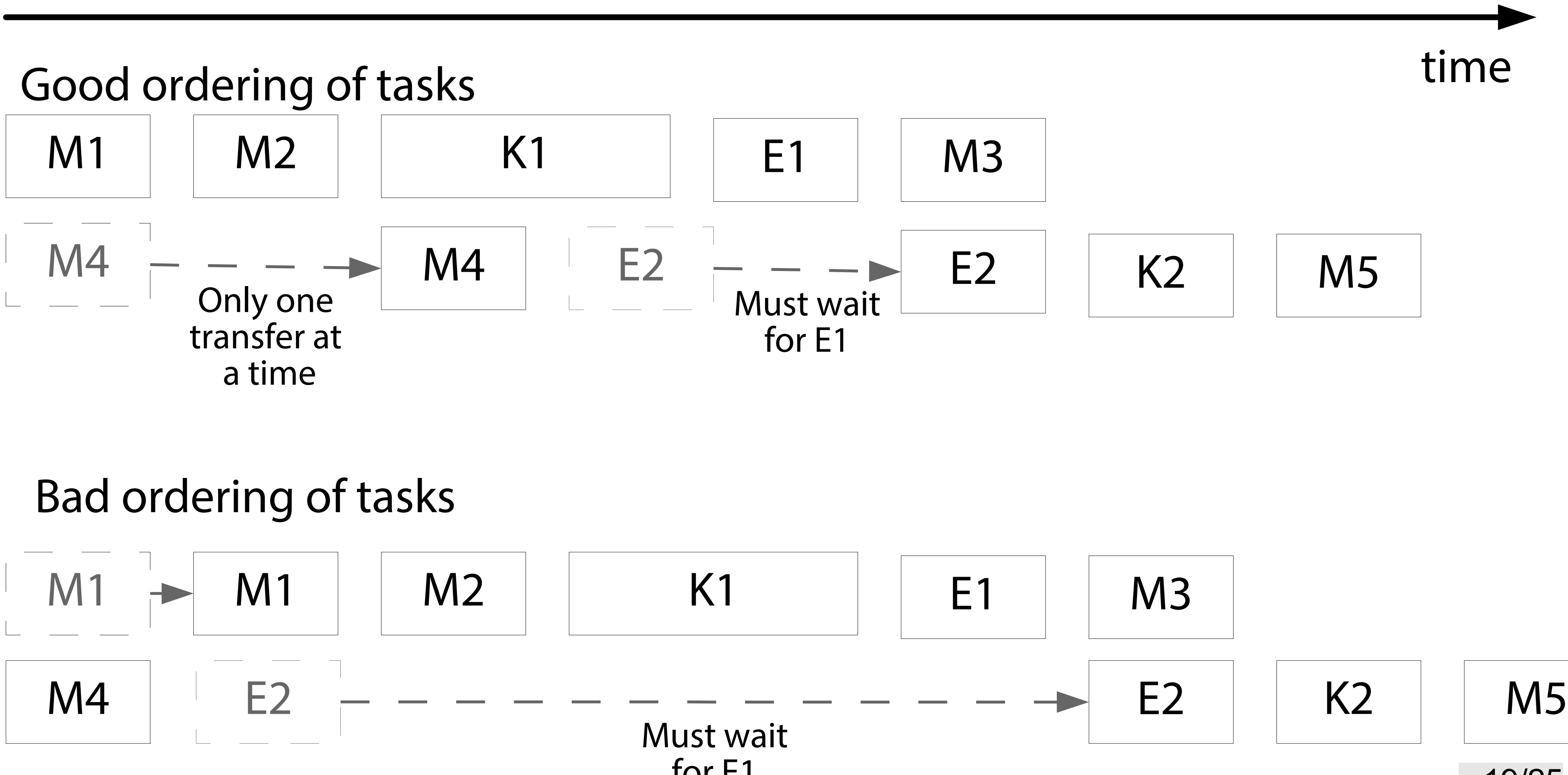
In Practice, DAGs May Get Quite Compicated



Representing Complex Dependences with Streams

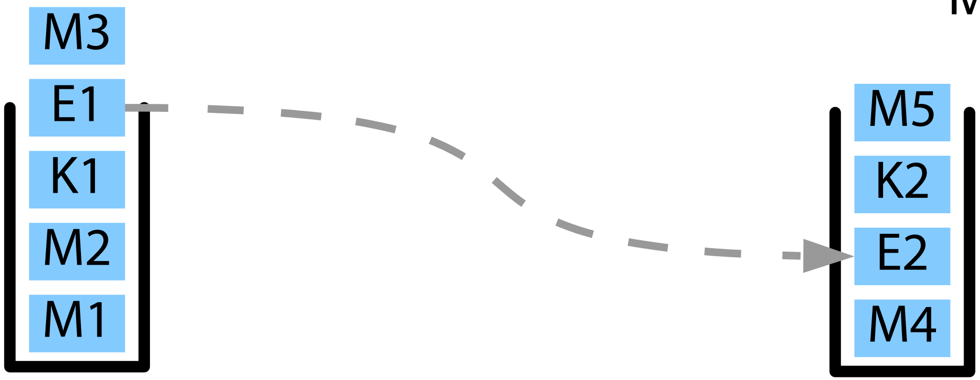


Execution Traces for Parallel Streams



CUDA Calls to Create the Stream and Dependence

- CreateStream(S1)
  CreateStream(S2)
- MemcpyAsync(A1, S1)
- MemcpyAsync(A2, S1)
- MemcpyAsync(A4, S2)
- Kernel1<<< S1 >>>
- RecordEvent(E1, S1)
- StreamWaitEvent(S2, E1)
- Kernel2<<< S2 >>>
- MemcpyAsync(A3, S1)
- MemcpyAsync(A5, S2)



Details on cudaStreamWaitEvent()

- There is no need for target event in cudaStreamWaitEvent()
- Syntax:
  - cudaStreamWaitEvent(stream that will block & wait, event to wait on)
- The event should be lightweight, so bypass time-stamping:
  - cudaEventCreateWithFlags(&event, cudaEventDisableTiming)
  - It is still possible to use cudaEventSynchronize() on events without time stamps
  - Conclusion:
    - Recording time inside GPU is expensive
    - Do not use time stamps unless you necessarily need them
    - Use other tools for performance analysis

Controlling Stream Behavior

- Starting with Kepler, there are many (32) hardware queues that can accept tasks from streams
- In some circumstances, there is no need for such a large number of concurrent connections to the GPU
- Use CUDA_DEVICE_MAX_CONNECTIONS to affect that setting
  - Default is 8
  - Max supported by Kepler is 32

Stream Callbacks

- Events may be used to synchronize tasks across streams
- Sometimes GPU-CPU synchronization is needed
  - Did my kernel finish?
  - Did my transfer finish?
- With cudaStreamAddCallback(), an event is enqueued that will call a user function at the time the event is reached in the stream
  - There are restrictions on the callback function
    - No CUDA calls can be made in the callback
    - The stream will not continue until the callback returns

Synchronization across Multiple Devices

- For each device (or a subset you need)
  - Switch to the device with cudaSetDevice()
  - Create streams and events on the device
    - cudaCreateStream(), cudaCreateStreamWithPriority()
    - cudaCreateEvent(), cudaCreateEventWithFlags()
  - Allocate memory on the device as needed
    - There is no non-blocking allocation yet so do this step separately
  - Insert transfer and compute tasks into streams
    - Avoid synchronous calls and the default stream
  - Insert events into streams to maintain dependence of tasks
  - Deallocate resources when done
- Keep in mind that...
  - Kernel launches and event recording can only occur to the stream associated with the current device
  - Memory transfers, event queries, and event synchronization may be issued to any stream regardless of the current device and the device associated with that stream

CUDA Development for Existing Applications

- Repeat the following steps until finished
  - Assessment
    - Find data-parallel structures in your code
  - Parallelization
    - Use existing libraries
      - cuSPARSE, cuBLAS, cuFFT, cuRAND, cuDNN, ...
    - Use tools for parallelization and vectorization
      - OpenACC, OpenMP
    - Develop missing CUDA code
  - Optimize
    - Maximize bandwidth and use of compute units
    - Minimize overhead and hide latency
    - Use existing profiling tools: nvprof, nvvp
  - Deploy
    - Consider machines with different resources than your development machine
      - Different count of different GPUs