Advanced Topics

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:

  ```c
  void *ptr;
  cudaHostAlloc( &ptr, size_in_bytes, cudaHostAllocDefault );
  cudaFreeHost( ptr );
  ```

- 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 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 applications
- CUDA streams provide a queue-like functionality for GPU operations that are 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
Is Overlap Possible on Your Device?

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

```c
cudaDeviceProp prop;
int whichDevice, dev_id;
cudaGetDevice( &dev_id );
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: Prolog

- **cudaStreamCreate()** creates a new stream:
  
  ```c
  // initialize the stream and create the stream
  cudaStream_t stream;
  cudaStreamCreate( &stream );
  ```

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

- CudaMemcpAsync() copies data to/from CPU to GPU. The copy continues after the call returns – completion happens later.
  ```c
  cudaMemcpyAsync( dev_a, host_a, N*sizeof(int), cudaMemcpyHostToDevice, stream );
  ```
Using One CUDA Stream: Epilog

- Sample kernel launch
  
  \[
  \text{kernel}<<<N/256,256,0,\text{stream}>>>(\text{dev}_a,\text{dev}_b,\text{dev}_c);
  \]

- copy back data from device to locked memory
  
  \[
  \text{cudaMemcpyAsync} (\text{host}_c, \text{dev}_c, N*\text{sizeof(int)},
  \text{cudaMemcpyDeviceToHost}, \text{stream});
  \]

- Stream synchronization waits for the stream finish all operations
  
  \[
  \text{cudaStreamSynchronize} (\text{stream});
  \]

- Free the memory allocated and destroy the stream after synchronization:
  
  \[
  \text{cudaFreeHost} (\text{host}_a);
  \text{cudaFree} (\text{dev}_a);
  \text{cudaStreamDestroy} (\text{stream});
  \]
Using More than One Stream

- Kernel execution and memory copies can be performed concurrently as long as:
  - they are in multiple streams and,
  - hardware supports it
- Some GPU cards support concurrent memory copies if they are in opposite directions
- The concurrency when multiple streams are present helps with:
  - Improving performance
  - Using PCIe express bus more efficiently
  - Will also help with NVLink
Sample Execution Time Line for 2 Streams

Stream 0
- Memcpy A to GPU
- Memcpy B to GPU
- kernel0
- Memcpy C from GPU
- Memcpy D to GPU
- Memcpy E to GPU
- kernel1
- Memcpy F from GPU

Stream 1
- Memcpy X to GPU
- Memcpy Y to GPU
- kernel8
- Memcpy Z from GPU
- Memcpy S to GPU
- Memcpy T to GPU
- kernel9
- Memcpy U from GPU

Compute and transfer overlap
GPU Work Scheduling

- GPU hardware has separate queues (engines) to perform memory copies and to execute kernels
  - Commands from CUDA streams are mapped to hardware queues
- 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 made streams useful in practice
- The default stream (NULL) always blocks commands 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
Programming CUDA Streams with Synchronization
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
- CUDA Graphs simplifies asynchronous CUDA programming
Representing Multiple Streams as Task Graph

Stream/queue view:

Direct Acyclic Graph (DAG):

```
cudaMemcpyAsync(->)
cudaMemcpyAsync(->)
Launch<<<>>>()
cudaMemcpyAsync(<=)
```

```
MemcpyAsync(->)
Launch<<<,,,>>>>
MemcpyAsync(<=)
```
Dependences Between Tasks are Based on Data

MemcpyAsync(A1, ..., -> )

Launch<<<,,,>>> (A1, A2, A3)

MemcpyAsync(..., A3, <-)

MemcpyAsync(A2, ..., -> )
In Practice, DAGs May Get Quite Complicated
Representing Complex Dependences with Streams

Divide into streams

Insert missing dependences
Execution Traces for Parallel Streams

Fortunate ordering of tasks

M1 → M2 → K1 → E1 → M3

M4 → M4

Only one transfer at a time

Must wait for E1

Unfortunate ordering of tasks

M1 → M1 → M2 → K1 → E1 → M3

M4 → E2

Must wait for E1

E2 → K2 → M5
CUDA Calls to Create the Stream and Dependence

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

Diagram:

- **M1**
- **M2**
- **M3**
- **M4**
- **M5**
- **K1**
- **K2**
- **E1**
- **E2**
- **M3** is connected to **K1**, and **E1**.
- **M4** is connected to **K2** and **E2**.
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 `CUDADEVICE_MAX_CONNECTIONS` to affect that setting:
  - Default is 8
  - Max supported by Kepler is 32
CUDA Streams’ 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
CUDA API for Multiple GPU Devices
Enumerating, Querying, and Using Multiple GPUs

- Management with multiple GPUs is done with global state
  - To obtain the total count: `cudaGetDeviceCount(int *num_gpus)`
    - GPUs are number from 0 to num_gpus-1
  - Call `cudaGetDeviceProperties()` to query device properties
    - For example: which GPU has the largest memory
  - Use `cudaSetDevice(gpu_id)` to select the GPU for the subsequent CUDA calls

- More information may be obtained via command “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’s use of GPUs is affected by 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 them 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 higher NVLink speeds
- Synchronization between multiple GPUs may be achieved through streams with synchronizing events
Synchronization across Multiple Devices

• For each device (or a selected subset)
  – Switch to the device with \texttt{cudaSetDevice()}
  – Create device’s streams and events: \texttt{cudaCreateStream()}, \texttt{cudaCreateStreamWithPriority()}, \texttt{cudaCreateEvent()}, \texttt{cudaCreateEventWithFlags()}
  – Allocate device memory: allocation is blocking so do it 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 not just the current device and its stream
CUDA Development for Existing Applications

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