Advanced GPU programming techniques

István Reguly, Oxford e-Research Centre
Many-Core Seminar, 20th November
Overview

Some of the lesser-known techniques and features

- Streams
- Synchronisation, Events, Callbacks
- Scheduling, Concurrency & false dependencies
- Hyper-Q & CUDA Proxy
- Peer-to-peer access (single proc, cudalpc, GPU Direct)
- CUDA 6.0
“Usual way”

cudaMemcpy(…)
kernel_1<<<...>>>(…)
cudaDeviceSynchronize()
timing()
kernel_2<<<...>>>(…)
cudaDeviceSynchronize()
timing()
...

Kernel launch overhead
~10-50us
“Improved way”

cudaMemcpyAsync(…)
kernel_1<<<…>>>(…)

kernel_2<<<…>>>(…)

Kernel launch overhead
~10-50us
Why streams?

- Concurrent CUDA operations
- Kernels
- Memcpy
- CPU execution
Why streams?

- Concurrent CUDA operations
- Kernels
- Memcpy
- CPU execution

Can have:
- up to 16 kernels
- 2 async copies
So what is a stream?

- A sequence of operations that execute strictly in-order
- But obviously there are limited resources -> concurrency
- The theoretical model:
  - Operations in a stream are serialised
  - Operations in different streams can run concurrently
So what is a stream?

• A sequence of operations that execute strictly in-order
• But obviously there are limited resources -> concurrency
• The theoretical model:
  • Operations in a stream are serialised
  • Operations in different streams can run concurrently
Default (NULL) stream

- This is the stream used when you don’t specify one
- Still asynchronous w.r.t the host for kernel launches and anything called *Async(...)
- But synchronous to all other streams -> no concurrency with them

Except for streams created with:
cudaStreamCreateWithOptions(&stream, cudaStreamNonBlocking)
Requirements for overlapped operations

- All operations that you want to potentially execute concurrently
- Have to be in different streams (non-NULL or nonBlocking)
- Memory copies have to be MemcpyAsync, using page-locked memory (cudaMallocHost)
- Available resources (SMs, registers, shared memory):
  - On a K20c you can have 13 (SM) * 16 (blocks/SM) running in any sort of combination - round robin scheduling
cudaStream_t streams[8];
for (int i = 0; i < 8; i++) cudaStreamCreate(&stream[i]);

cudaMallocHost(host1); cudaMalloc(device1);

cudaMemcpyAsync(host1, device1, D2H, stream[0]);
cudaMemcpyAsync(device2, host2, H2D, stream[1]);
for (int i = 2; i < 8; i++) {
    kernel_do_sg<<<4,128,0,stream[i]>>>(...);
}
Synchronisation

- Device-wide sync: `cudaDeviceSynchronize()`
  - CPU will block until everything finishes

- Synchronising a specific stream: `cudaStreamSynchronize(stream)`
  - CPU blocks until everything in the given stream finishes

- Events
  - Insert Events into streams, synchronise or time with them
  - `cudaEventRecord(event, stream)`
  - `cudaEventSynchronize(event)` - for the CPU
  - `cudaStreamWaitEvent(stream, event)` - for a stream
  - `cudaEventQuery(event)`
  - `cudaStreamAddCallback(stream, callback, u_data, flags)`
Synchronisation

CPU

- launch(s3)
- launch(s1)
- eventRecord(e1,s1)
- eventSynchronize(e1)
- launch(s2)
- launch(s3)
- sW4E(s1,e2)
- launch(s1)

GPU

- stream 1
- streamWait(s1)
- Kernel 1
- stream 2
- Kernel 2
- stream 3
- Kernel 3
- do_something_on_CPU()
*Footnote*

- There are some operations which cause a device-wide synchronisation:
  - cudaMallocHost, cudaMemcpy
  - cudaMalloc
  - Memory operations without Async (cudaMemset, cudaMemcpy)
  - cudaDeviceSetCacheConfig
- Fermi has 1 compute queue
  - Serialisation & phase dependencies
  - Can lead to deadlock
- Not an issue with Kepler (16 queues with Hyper-Q)
Priority streams

- `cudaStreamCreateWithPriority`

- Kernel launches in higher priority streams will execute before any pending launches in lower priority do.
Proxy server

- An incarnation of Hyper-Q
- Fancy name/technique for multiple processes sharing the same CUDA context
- Targeted at legacy applications, you can have 32 processes connect to the same GPU, if they each do small size GPU computations, you can get overlap

http://cudamusing.blogspot.co.uk/2013/07/enabling-cuda-multi-process-service-mps.html
Multi-GPU

• You can control multiple GPUs from the same process

• One or more threads - `cudaSetDevice(id)`

• Can cross-schedule `cudaEventSynchronize()` and `cudaStreamWaitForEvent()`

• Peer-to-peer access - `cudaDeviceEnablePeerAccess`

• `cudaMemcpyPeerAsync(...)`

• Can pass pointers between devices - paged copies

• UVA - can get lazy with source, destination pointers

  • `cudaMemcpyAsync(target, source, size, stream, cudaMemcpyDefault)`
cudalpc

• Sharing GPU pointers between MPI processes
  • cudaIpcGetMemHandle(handle, ptr)
  • send handle over MPI
    • cudaIpcOpenMemHandle(handle, ptr, flags)
• Same works for Events
  • cudaIpcGetEventHandle, cudaIpcOpenEventHandle
• Will fail if Peer-to-Peer is not possible!
• Just use GPU Direct... (export MV2_USE_CUDA=1)
CUDA 6.0

• Unified Memory

• Pointers accessible from both the CPU and the GPU

• `cudaMallocManaged`

• Will automatically copy data up and down

• Synchronisation!

+ Improved Dynamic Parallelism
Tip #1 - single GPU

- Reductions - D2H copy of a few bytes
- If you don’t make a control decision using it, do a callback
- If you do, consider making it less often
- Timings - do not use cudaDeviceSynchronize
- Do a callback instead using events
Tip #2 - multi GPU

- Always try avoiding:
  - Small memcopies (good advice for MPI too!)
  - cudaMalloc, cudaMemcpyHost calls
- Overlap (or at least Async) what you can:
  - Halo copies
  - Halo unpack

<table>
<thead>
<tr>
<th>Gather 1</th>
<th>D2H</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gather 2</td>
<td>Kernel interior</td>
</tr>
</tbody>
</table>
Tip #2 - multi GPU

- Always try avoiding:
  - Small memcopies (good advice for MPI too!)
  - cudaMemcpy, cudaMemcpyHost calls
- Overlap (or at least Async) what you can:
  - Halo copies
  - Halo unpack
Thank you!

istvan.reguly@oerc.ox.ac.uk