Peer-to-Peer & Unified Virtual Addressing
CUDA Webinar

Tim C. Schroeder,
HPC Developer Technology Engineer
Outline

- Overview: P2P & UVA
- UVA Memory Copy
- UVA Zero-Copy
- P2P Memory Copy
- P2P Direct Addressing
- Summary, Further Reading and Questions
Unified Virtual Addressing
Easier to Program with Single Address Space

No UVA: Multiple Memory Spaces

UV A: Single Address Space
Peer-to-Peer (P2P) Communication

**Direct Access**
- GPU0 Memory
- GPU1 Memory
- PCI-e
- Load / Store

**Direct Transfers**
- GPU0 Memory
- GPU1 Memory
- PCI-e
- cudaMemcpy()

Eliminates system memory allocation & copy overhead
More convenient multi-GPU programming
What we’re focusing on today

- Lots of different use cases for P2P & UVA, we’re going to go through the following:
  - UVA Memory Copy - cudaMemcpy(…, cudaMemcpyDefault)
  - P2P Memory Copy (GPU to GPU)
  - P2P Memory Access from a CUDA kernel
  - …as demonstrated by the simpleP2P sample
Unified Virtual Addressing (UVA)

- **One address space for all CPU and GPU memory**
  - Determine physical memory location from pointer value
  - Enables libraries to simplify their interfaces (e.g. cudaMemcpy)

<table>
<thead>
<tr>
<th>Before UVA</th>
<th>With UVA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Separate options for each permutation</td>
<td>One function handles all cases</td>
</tr>
<tr>
<td>cudaMemcpyHostToDevice</td>
<td></td>
</tr>
<tr>
<td>cudaMemcpyDeviceToDevice</td>
<td></td>
</tr>
<tr>
<td>cudaMemcpyDeviceToDevice</td>
<td></td>
</tr>
<tr>
<td>cudaMemcpyDeviceToHost</td>
<td></td>
</tr>
<tr>
<td>cudaMemcpyDeviceToDevice</td>
<td>cudaMemcpyDefault</td>
</tr>
<tr>
<td>(data location becomes an implementation detail)</td>
<td></td>
</tr>
</tbody>
</table>
UVA Memory Copy Step-by-Step (1/2)

- Copy without specifying in which memory space src / dst are

Requirements
- Needs to be a 64bit application
- Fermi-class GPU
- Linux or Windows TCC
- CUDA 4.0

- Call cudaGetDeviceProperties() for all participating devices, check cudaDeviceProp::unifiedAddressing flag
Between host and multiple devices:

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

Between two devices:

- `cudaMemcpy(gpu0_buf, gpu1_buf, buf_size, cudaMemcpyDefault)`
Zero-Copy Memory With UVA

- Pointers returned by cudaHostAlloc() can be used directly from within kernels running on UVA enabled devices (i.e. there is no need to obtain a device pointer via cudaHostGetDevicePointer())
Peer-to-Peer Communication Between GPUs

- **Direct Transfers**
  - `cudaMemcpyp()` initiates DMA copy from GPU₀ memory to GPU₁ memory
  - Works transparently with CUDA Unified Virtual Addressing (UVA)

- **Direct Access**
  - GPU₀ reads or writes GPU₁ memory (load/store)
  - Data cached in L2 of the target GPU

- **Performance Expectations**
  - High bandwidth: saturates PCIe (> 6GB/s observed)
  - Low latency: 1 PCIe transaction + 1 memory fetch (~2.5us)
P2P Memory Copy Step-by-Step (1/4)

- Copy data between GPUs without going through host memory

- Requirements
  - Needs to be a 64bit application
  - Fermi-class Tesla GPU
  - Linux or Windows TCC
  - CUDA 4.0
  - Drivers v270.41.19 or later
  - GPUs need to be on same IOH
- Check for peer access between participating GPUs:
  
cudaDeviceCanAccessPeer(&can_access_peer_0_1, gpuid_0, gpuid_1);
cudaDeviceCanAccessPeer(&can_access_peer_1_0, gpuid_1, gpuid_0);

- Enable peer access between participating GPUs:
  
cudaSetDevice(gpuid_0);
cudaDeviceEnablePeerAccess(gpuid_1, 0);
cudaSetDevice(gpuid_1);
cudaDeviceEnablePeerAccess(gpuid_0, 0);
Now we can do a UVA memory copy just like before:

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

- `cudaMemcpy()` knows that our buffers are on different devices (UVA), will do a P2P copy now

- Note that this will transparently fall back to a normal copy through the host if P2P is not available
P2P Memory Copy Step-by-Step (4/4)

- **Shutdown peer access at the end:**
  
  ```c
  cudaSetDevice(gpuId_0);
  cudaDeviceDisablePeerAccess(gpuId_1);
  cudaSetDevice(gpuId_1);
  cudaDeviceDisablePeerAccess(gpuId_0);
  ```

- **Optional**

  No need to shutdown if you’re requiring the same peer access throughout your program, but if it’s limited to a specific phase you can potentially reduce overhead and free resources by explicitly disabling it once you’re done.
P2P Direct Access Step-by-Step (1/3)

- Starting point – same as for P2P memory copy

- Same initialization
  - cudaDeviceCanAccessPeer()
  - cudaDeviceEnablePeerAccess()

- Same shutdown
  - cudaDeviceDisablePeerAccess()

- Same system requirements
P2P Direct Access Step-by-Step (2/3)

- Basic copy kernel as example, taking two buffers:

```c
__global__ void SimpleKernel(float *src, float *dst)
{
    const int idx = blockIdx.x * blockDim.x + threadIdx.x;
    dst[idx] = src[idx];
}
```
After P2P initialization, this kernel can now read and write data in the memory of multiple GPUs:

cudaSetDevice(gpuid_0); SimpleKernel<<<blocks, threads>>> (gpu0_buf, gpu1_buf);
cudaSetDevice(gpuid_0); SimpleKernel<<<blocks, threads>>> (gpu1_buf, gpu0_buf);
cudaSetDevice(gpuid_1); SimpleKernel<<<blocks, threads>>> (gpu0_buf, gpu1_buf);
cudaSetDevice(gpuid_1); SimpleKernel<<<blocks, threads>>> (gpu1_buf, gpu0_buf);

UVA makes sure the kernel knows whether its argument is from local memory, another GPU or zero-copy from the host
Summary

- P2P and UVA can be used to both simplify and accelerate your CUDA programs
- Faster memory copies between GPUs with less host overhead
- Kernels can directly read and write memory of other GPUs
- Transparent addressing of memory on different devices
Further reading

- simpleP2P Sample, SDK 4.0 (Demonstrates both P2P & 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
Questions?