CUDA
Asynchronous Concurrent Execution

Bedřich Beneš, Ph.D.
Purdue University
Department of Computer Graphics

What is it?

- Asynchronous Concurrent Execution allows for concurrent execution between host and device.
- Control is returned to the host before the device has completed the task.

Async. concurrent execution

- Kernels
- Mem copy on device
- Mem copy device-host (64kB or less)
- Mem copy by functions with suffix Async
- Mem set call functions

CUDA Synchronization

Kernel calls with device are synchronous

cudaMemcpy(dA,hA,size,cudaMemcpyHostToDevice);
VecAdd<<<n,1>>>(dA,dB,n);
CpuDoSomething(); //Does not wait
cudaMemcpy(dB,hB,size,cudaMemcpyDeviceToHost);
CUDA Synchronization

- There is asynchronous memory copy
  
cudaMemcpyAsync(HostToDevice)
cudaMemcpyAsync(DeviceToHost)

- GPU/CPU can perform memory transfer and GPU execution simultaneously

Page-Locked Host Memory

- Also called pinned memory
- Allocated by cudaMemcpyAsync()
- Freed by cudaMemcpy()
- Is guaranteed not to be paged
- H2D and D2H can be 2x faster
- Use it carefully! May run out of memory.

Stream

- A stream is a sequence of GPU operations that is executed in a specific order
- CUDA operations within stream run in the defined order
- CUDA operations in different streams can run concurrently or may be interleaved

Streams

Stream creation:

cudaStream_t stream;
cudaStreamCreate(&stream);
Streams and Events

Example:
cudaEvent_t start;
cudaEventCreate(&start);
cudaEventRecord(start, stream)

Event start is recorded into stream stream

Simple stream

• Let’s take large data and divide it into small sets that will be processed in a queue

for (int i=0; i<n; i++) {
cudaMemcpyAsync(d_a, h_a+i, size,
cudaMemcpyHostToDevice, stream);
cudaMemcpyAsync(d_b, h_b+i, size,
cudaMemcpyHostToDevice, stream);
Kernel<<<n/256, 256, 0, stream>>>(d_a, d_b, d_c);
cudaMemcpyAsync(h_c+i, d_c, size,
cudaMemcpyDeviceToHost, stream);
}

Simple stream

• Now the stream is ready
• We can wait for the completion by

cudaStreamSynchronize(stream);

Simple stream

• We can measure the execution time

cudaEvent_t stop;
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time, start, stream)

• Free the memory

cudaFreeHost(h_a);
cudaFreeHost(h_b);
cudaFreeHost(h_c);
Simple stream

And destroy the stream

cudaStreamDestroy(stream);

Multiple Streams

- All the fun starts with multiple streams
- CUDA allows for interleaved kernel execution and data transfer
- So if we can schedule data transfer AND kernel execution at the same time, it runs in parallel

Multiple Streams

<table>
<thead>
<tr>
<th>Stream 0</th>
<th>Stream 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memcpy A to Device</td>
<td>Memcpy A to Device</td>
</tr>
<tr>
<td>Memcpy B to Device</td>
<td>Memcpy B to Device</td>
</tr>
<tr>
<td>Kernel invocation</td>
<td>Kernel invocation</td>
</tr>
<tr>
<td>Memcpy C to Device</td>
<td>Memcpy C to Device</td>
</tr>
<tr>
<td>Memcpy A to Device</td>
<td>Memcpy A to Device</td>
</tr>
<tr>
<td>Memcpy B to Device</td>
<td>Memcpy B to Device</td>
</tr>
<tr>
<td>Kernel invocation</td>
<td>Kernel invocation</td>
</tr>
<tr>
<td>Memcpy C to Device</td>
<td>Mempy C to Device</td>
</tr>
</tbody>
</table>

Visual Profiler

- And this is how it looks in Visual Profiler
Example

Serial

cudaMemcpyAsync(H2D) Kernel<<<>>>> cudaMemcpyAsync(D2H)

Concurrent – kernel and memcpy overlap

cudaMemcpyAsync(H2D) K D2H
K D2H
K D2H
K D2H

Stream concurrency

2-way

H2D K D2H
K

3-way

H2D K D2H
H2D K D2H
H2D K D2H

Stream concurrency

• 4-way

H2D K K D2H
H2D K K D2H
H2D K K D2H
H2D K K D2H

Stream concurrency

• 4-way+CPU

H2D K K D2H
H2D K K D2H
H2D K K D2H
H2D K K D2H

CPU
Stream CPU Concurrency

- Possible overlap
  
  ```
  cudaMemcpy(&dev1, size);
  double* h1 = (double*) malloc(&h1, size);
  cudaMemcpy(d1, h1, size, H2D);
  k1<<<grid, block, 0>>>( ..., dev2, ... );
  SomeCPUFunc(); // Could run in parallel
  k2<<<grid, block, 0>>>( ..., dev3, ... );
  cudaMemcpy(h4, d4, size, D2H );
  ```

How to use multiple streams?

- A task done on a large data
- Divide it into small chunks
- Use multiple streams to process each
- Kernels can do exactly the same
- By their interleaving we gain speed

Mapping Streams to Engines

<table>
<thead>
<tr>
<th>Stream 0</th>
<th>Copy Engine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memcpy A to Device</td>
<td>Memcpy A to Device</td>
</tr>
<tr>
<td>Memcpy B to Device</td>
<td>Memcpy B to Device</td>
</tr>
<tr>
<td>Kernel invocation</td>
<td>Memcpy C to Device</td>
</tr>
<tr>
<td>Memcpy C to Device</td>
<td>Memcpy A to Device</td>
</tr>
<tr>
<td>Memcpy C to Device</td>
<td>Memcpy B to Device</td>
</tr>
<tr>
<td>Memcpy C to Device</td>
<td>Memcpy C to Device</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stream 1</th>
<th>Kernel Engine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memcpy A to Device</td>
<td>Kernel Invocation</td>
</tr>
<tr>
<td>Memcpy B to Device</td>
<td>Kernel Invocation</td>
</tr>
<tr>
<td>Kernel invocation</td>
<td>Kernel Invocation</td>
</tr>
<tr>
<td>Memcpy C to Device</td>
<td>Kernel Invocation</td>
</tr>
</tbody>
</table>
Device Work Scheduling

- The hardware does not know about streams
- It is an abstraction
- As always, when we have parallelization, we need synchronization

Explicit Synchronization

```c
cudaEventCreate(&e); // create event
cudaMemcpyAsync(d_in, in, size, H2D, s1); // 1) H2D copy of new input
cudaEventRecord(e, s1); // record event
cudaMemcpyAsync(out, d_out, size, D2H, s2); // 2) D2H copy of previous result
cudaStreamWaitEvent(s2, e);
kernel <<< , , , s2>>>(d_in, d_out);
```

Explicit Synchronization

Create 'Events', within streams, to use for synchronization
- `cudaEventRecord(event, streamid)`
- `cudaEventSynchronize(event)`
- `cudaStreamWaitEvent(stream, event)`
- `cudaEventQuery(event)`

Reading

- J. Sanders & E. Kandrot
  *CUDA by Example*
  Morgan Kaufmann 2011
  Chapter 10