CSCI-UA.0480-003
Parallel Computing

Lecture 22: CUDA - V

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com
Some Advanced Topics

• Overlapping computation and data transfer
• Asynchronous execution
• Multi-GPU systems
Overlapping Computation and Data-transfer: Streams

- A sequence of operations that execute on the device in the order in which they are issued by the host code
- Operations in different streams can be interleaved and, when possible, they can even run concurrently.
- A stream can be sequence of kernel launches and host-device memory copies
- Can have several open streams to the same device at once
- Need GPUs with concurrent transfer/execution capability
- Potential performance improvement: can overlap transfer and computation
Streams

- By default all transfers and kernel launches are assigned to stream 0
  - This means they are executed in order

![Diagram showing host thread and device with FIFO]

Device
Example: Default Stream

cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
increment<<<1,N>>>(d_a);
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);

• In the code above, from the perspective of the device, all three operations are issued to the same (default) stream and will execute in the order that they were issued.
• From the perspective of the host:
  • data transfers are blocking or synchronous transfers
  • kernel launch is asynchronous.

Isn’t this more efficient?

cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
increment<<<1,N>>>(d_a);
anyCPUfunction();
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
Example: Non-Default Stream

Non-default streams in CUDA C/C++ are declared, created, and destroyed in host code as follows:

```c
cudaStream_t stream1;
cudaError_t result;
result = cudaStreamCreate(&stream1);
result = cudaStreamDestroy(stream1);
```

To issue data transfer to non-default stream (non-blocking):

```c
result = cudaMemcpyAsync(d_a, a, N, cudaMemcpyHostToDevice, stream1);
```

To launch a kernel to non-default stream:

```c
increment<<<1,N,0,stream1>>>(d_a);
```
Important

• All operations in non-default streams are non-blocking with respect to the host code.
• Sometimes you need to synchronize the host code with operations in a stream.
• You have several options:
  – `cudaDeviceSynchronize()` → blocks host
    • Blocks until the device has completed all preceding requested tasks.
  – `cudaStreamSynchronize(stream)` → blocks host
    • Blocks until stream has completed all operations.
  – `cudaStreamQuery(stream)` → does not block host
    • Returns `cudaSuccess` if all operations in stream have completed, or `cudaErrorNotReady` if not (both of type `cudaError_t`)
Streams

• The amount of overlap execution between two streams depends on:
  – Device supports overlap transfer and kernel execution
  – Devices supports concurrent kernel execution
  – Device supports concurrent data transfer
  – The order on which commands are issued to each stream
Using streams to overlap device execution with data transfer

• Conditions to be satisfied first:
  – The device must be capable of concurrent transfer and execution.
  – The kernel execution and the data transfer to be overlapped must both occur in different, non-default streams.
  – The host memory involved in the data transfer must be pinned memory.
Pinned Pages

• Allocate page(s) from system RAM 
  (cudaMallocHost() or cudaHostAlloc())
  - Cannot be paged out
  - Enables highest memory copy performance 
    (cudaMemcpyAsync())

• If too much pinned pages, overall system performance may greatly suffer.
Using streams to overlap device execution with data transfer

for (int i = 0; i < nStreams; ++i) {

    int offset = i * streamSize;

    cudaMemcpyAsync(&d_a[offset], &a[offset], streamBytes, cudaMemcpyHostToDevice, stream[i]);

    kernel<<<Nblks, Nthreads, stream[i]>>>(d_a, offset);

    cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes, cudaMemcpyDeviceToHost, stream[i]);
}

So..

- Streams are a good way to overlap execution and transfer, hardware permits.
- Don’t confuse kernels, threads, and streams.
Asynchronous Execution

• Asynchronous = returns to host right-away and does not wait for device

• This includes:
  – Kernel launches;
  – Memory copies between two addresses to the same device memory;
  – Memory copies from host to device of a memory block of 64 KB or less;
  – Memory copies performed by functions that are suffixed with Async;
Asynchronous Execution

• Some CUDA API calls and all kernel launches are asynchronous with respect to the host code.
• This means error-reporting is also asynchronous.
• Asynchronous transfer (cudaMemcpyAsync()) version requires pinned host memory
• On all CUDA-enabled devices, it is possible to overlap host computation with asynchronous data transfers and with device computations.
Asynchronous Execution

cudaMemcpyAsync(a_d, a_h, size, cudaMemcpyHostToDevice, 0);
kernel<<<grid, block>>>(a_d);
cpuFunction();
Other Sources of Concurrency

- Some devices of compute capability 2.x and higher can execute multiple kernels concurrently.
- The maximum number of kernel launches that a device can execute concurrently is 32 on devices of compute capability 3.5 and 16 on devices of lower compute capability.
- A kernel from one CUDA context cannot execute concurrently with a kernel from another CUDA context. A CUDA context is the application.
- Kernels that use many textures or a large amount of local memory are less likely to execute concurrently with other kernels.
- Some devices of compute capability 2.x and higher can perform a copy from page-locked host memory to device memory concurrently with a copy from device memory to page locked host memory.
Multi-GPU systems: Flavors

- Multiple GPUs in the same node (e.g. PC)
- Multi-node system (e.g. MPI).

Multi-GPU configuration is here to stay!
Hardware Example: Tesla S870 Server
Hardware Example: Tesla S870 Server

Connected to a single-host
Hardware Example: Tesla S870 Server

Host System w/ 1 PCIe slot

Tesla S870

Host System w/ 1 PCIe slot

Connected to a two host systems
Future NVLINK
(Expected in PASCAL Architecture ... Circa 2016)
Why Multi-GPU Solutions

• Scaling-up performance
• Another level of parallelism
• Power
• Reliability
// Run independent kernel on each CUDA device
int numDevs = 0;
cudaGetDeviceCount(&numDevs);
...
for (int d = 0; d < numDevs; d++) {
    cudaSetDevice(d);
    kernel<<<blocks, threads>>>(args);
}
CUDA Support

• `cudaGetDeviceCount( int * count)`
  – Returns in *count the number of devices

• `cudaGetDevice( int * device )`
  – Returns in *device the device on which the active host thread executes the device code.
CUDA Support

• `cudaSetDevice(devID)`
  - Device selection within the code by specifying the identifier and making CUDA kernels run on the selected GPU.

```c
size_t size = 1024 * sizeof(float);
cudaSetDevice(0); // Set device 0 as current
float* p0;
cudaMalloc(&p0, size); // Allocate memory on device 0
MyKernel<<<1000, 128>>>(p0); // Launch kernel on device 0
cudaSetDevice(1); // Set device 1 as current
float* p1;
cudaMalloc(&p1, size); // Allocate memory on device 1
MyKernel<<<1000, 128>>>(p1); // Launch kernel on device 1
```
Conclusions

• There are many performance enhancement techniques in our arsenal:
  – Streams
  – Texture memory
  – Asynchronous execution
  – ...

• Multi-GPU system:
  – is an efficient way to reach higher performance
  – Performance gain is application-dependent and programmer-dependent!