Harness the Power of GPUs: An Introduction to GPGPU Programming
Lecture 6: Streams and Dynamic Parallelism

Guido Juckeland
Visiting Scholar
Technische Universität Dresden, Germany

June 18, 2014
Types of parallelism

- Bit-wise parallelism
- Instruction level parallelism
- Data parallelism
- Task parallelism
Why task parallelism?

- So far only serial GPU usage

- **Standard** `cudaMemcpy` is synchronous

- Use cases:
  - Overlapping data transfers and kernel execution
  - Run multiple (small) kernels concurrently
  - Increased device utilization
Host-device concurrency

```c
//do stuff
kernel<<<..., ...>>>();
//do more stuff
cudaDeviceSync();
```
CUDA Streams

- Stream = Queue of device operations (kernels, memcpy)

- Default stream (stream 0)
  - Used so far (used when no stream is specified)
  - Blocks all concurrent kernel execution on the device, i.e. only one kernel from stream 0 can run at any time
  - Launched kernels are executed in launch order
Managing streams

• Creating streams
  cudaStream_t stream1;
cudaStreamCreate(&stream1);

• Destroying streams
  cudaStreamDestroy(stream1);

• Specifying streams
  – On kernel launches as 4th argument in <<<,>>>
    kernel<<<grid, block, sharedMemory, stream>>>(args);
  – On memcpy’s as 5th argument
    cudaMemcpyAsync(dst, src, size, direction, stream);
Asynchronous data transfers

- Require pinned memory on the host side
  \[ \text{cudaMallocHost()} \]

- Concurrent data transfers need to be in different stream from kernel
  \[ \text{cudaStream_t stream1, stream2;}
  \text{cudaStreamCreate ( &stream1);}
  \]
  
  \[ \text{...}
  \text{cudaMalloc ( &dev1, size );}
  \text{cudaMallocHost ( &host1, size ); // pinned memory}
  \]
  
  \[ \text{...}
  \text{cudaMemcpyAsync ( dev1, host1, size, H2D, stream1 );}
  \text{kernel2 << grid, block, 0, stream2 >>> ( ..., dev2, ... );} \]

Concurrent kernels

• **Launch kernels in different streams**
  
  kernel1 <<< grid1, block1, 0, stream1 >>>> ( ... ) ;
  kernel2 <<< grid2, block2 0, stream2 >>>> ( ... ) ;

• Pre Kepler devices hat poor kernel concurrency
• HyperQ in Kepler enables better utilization
Limitations

- Activity in one stream is still serialized
- Only one ongoing data transfer per direction at a time
- Activity in stream0 stops all concurrent device activity
Synchronization

- `cudaDeviceSync()`
  - Wait for all outstanding activity on the device to finish
- `cudaStreamSync(stream)`
  - Wait for all outstanding activity in the stream to finish

- Event based synchronization
  - Uses event to mark completion of activity
Using events

- Example: kernel2 depends on results of kernel1 and new input data
  
  ```c
  cudaEvent_t event;
  cudaEventCreate (&event); // create event

  // run kernel1 in stream1
  kernel1 <<< , , , stream1 >>> ( d1_in, d1_out );

  // record event at end of kernel in stream1
  cudaEventRecord (event, stream1);

  // run data transfer in stream2
  cudaMemcpyAsync ( d2_in, in, size, H2D, stream2 );

  // wait for event in stream1
  cudaStreamWaitEvent ( stream2, event );

  // enqueue kernel2 in stream2 to run after the event is received
  kernel2 <<< , , , stream2 >>> ( d2_in, d2_out );
  ```
Using events on the host side

- `cudaEventSynchronize(event)`
  - Blocks host execution until event occurs
- `cudaEventQuery(event)`
  - Tests if event has occurred (cudaSuccess or cudaErrorNotReady)
- `cudaEventElapsedTime(&time,startEvent,endEvent)`
  - Returns elapsed time in ms between start and end event
Timing kernel execution

cudaEvent_t start, end;
float time;

cudaEventCreate(&start);
cudaEventCreate(&end);

// prepare kernel

cudaEventRecord(start, 0); // not using streams
kernel<<<...,....>>>();
cudaEventRecord(end, 0);
cudaEventElapsedTime(&time,start,stop);
Dynamic Parallelism

- Up until Kepler
  - All work is host initiated
  - Streams for task parallelism
  - Device functions for better modularization → will be inlined by compiler

- Dynamic parallelism allows kernels to launch new kernels
  - Enables recursion
  - Useful for adaptive grids
Dynamic parallelism for library calls

```c
__global__ void libraryCall(float *a, float *b, float *c)
{
    // All threads generate data
    createData(a, b);
    __syncthreads();

    // Only one thread calls library
    if(threadIdx.x == 0) {
        cublasDgemm(a, b, c);
        cudaMemcpy(dOut, c, dOutSize, cudaMemcpyDeviceToHost);
        cudaMemcpy(a, dOut, dOutSize, cudaMemcpyHostToDevice);
    }

    // All threads wait for dgemm
    __syncthreads();

    // Now continue
    consumeData(c);
}
```

Dynamic parallelism for kernel calls

```c
__global__ void qsort(int *data, int l, int r)
{
    int pivot = data[0];
    int *lptr = data+l, *rptr = data+r;

    // Partition data around pivot value
    partition(data, l, r, lptr, rptr, pivot);

    // Launch next stage recursively
    if(l < (rptr-data))
        qsort<<<...>>>(data, l, rptr-data);
    if(r > (lptr-data))
        qsort<<<...>>>(data, lptr-data, r);
}
```

QUESTIONS?