GPU CUDA Programming

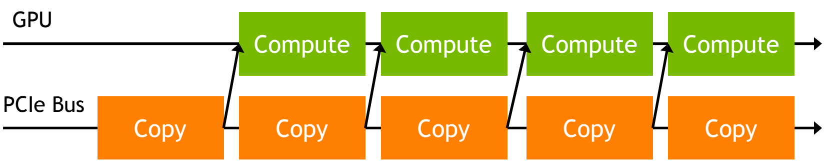
이 정 근 (Jeong-Gun Lee)

한림대학교 소프트웨어융합대학, 임베디드 SoC 연구실

www.onchip.net
Email: **Jeonggun.Lee@hallym.ac.kr**

Overlapping Comm. & Computation

- Three sequential steps for a single kernel execution
- Multiple kernels
  – **Asynchrony** is a first-class citizen of most GPU programming frameworks
  – **Computation-communication overlap** is a common technique in GPU programming



Abstract Concurrency

- Different kinds of action overlap are possible in CUDA?
  1. Overlapped host computation and device computation

     CPU function GPU kernel
  2. Overlapped host computation and host-device data transfer

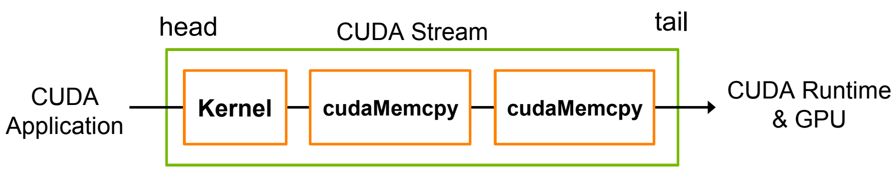
     CPU GPU
  3. Overlapped host-device data transfer and device computation

     GPU GPU
  4. Concurrent device computation

     GPU Kenel 1 kernel 2
- CUDA "**Streams**" to achieve each of these types of overlap

CUDA Streams

- CUDA Streams: a FIFO queue of CUDA actions to be performed
  – Placing a new action at the head of a stream is **asynchronous**
  – Executing actions from the tail as CUDA resources allow
  – Every action (kernel launch, cudaMemcpy, etc) runs in an implicit or explicit stream



FIFO

CUDA Streams

- Two types of streams in a CUDA program
  - The **implicitly** declared stream (**NULL stream**)
  - **Explicitly** declared streams (**non-NULL streams**)

- Up until now, all code has been using the NULL stream by default

```
cudaMemcpy(...);
kernel<<<...>>>(...);
cudaMemcpy(...);
```

- Non-NULL streams require manual allocation and management by the CUDA programmer

CUDA Streams

- To create a CUDA stream:

```
cudaError_t cudaStreamCreate(cudaStream_t *stream);
```

- To destroy a CUDA stream:

```
cudaError_t cudaStreamDestroy(cudaStream_t stream);
```

- To wait for all actions in a CUDA stream to finish:

```
cudaError_t cudaStreamSynchronize(cudaStream_t stream);
```

- To check if all actions in a CUDA stream have finished:

```
cudaError_t cudaStreamQuery(cudaStream_t stream);
```

CUDA Streams

- **cudaMemcpyAsync**: Asynchronous memcpy

```
cudaError_t cudaMemcpyAsync(void *dst, const void *src,
                            size_t count, cudaMemcpyKind kind,
                            cudaStream_t stream = 0);
```

- **cudaMemcpyAsync** does the same as cudaMemcpy, but may return before the transfer is actually complete
- **cudaMemcpyAsync** requires "pinned host memory"
  - Memory that is resident in physical memory pages, and cannot be swapped out, also referred as page-locked

CUDA Streams

- Performing a cudaMemcpyAsync:

```
int *h_arr, *d_arr;
cudaStream_t stream;
cudaMalloc((void **)&d_arr, nbytes);
cudaMallocHost((void **)&h_arr, nbytes);
cudaStreamCreate(&stream);

cudaMemcpyAsync(d_arr, h_arr, nbytes, cudaMemcpyHostToDevice,
stream);
...
cudaStreamSynchronize(stream);
cudaFree(d_arr); cudaFreeHost(h_arr); cudaStreamDestroy(stream);
```

① page-locked memory allocation

② Call return before transfer complete

③ Do something while data is being moved

④ Sync to make sure operations complete

CUDA Streams

- Associate kernel launches with a non-NULL stream
  - Note that kernels are always asynchronous

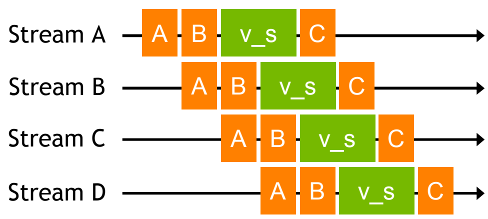
```
kernel<<<nblocks, threads, smem_size, stream>>>(...);
```

- The effects of cudaMemcpyAsync and kernel launching
  - Operations are put in the stream queue for execution
  - Actually operations may not happen yet
- Host-side timer to time those operations
  - Not the actual time of the operations

CUDA Streams

- Vector sum example, A + B = C

- Partition the vectors and use CUDA streams to overlap copy and compute



CUDA Streams

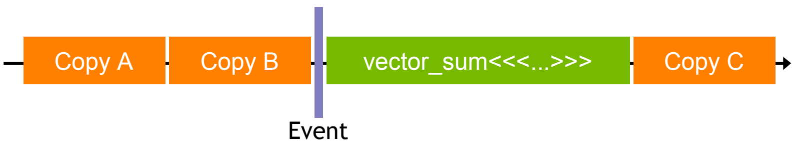
- How can this be implemented in code?

```
for (int i = 0; i < nstreams; i++) {
   int offset = i * eles_per_stream;
   cudaMemcpyAsync(&d_A[offset], &h_A[offset], eles_per_stream *
                            sizeof(int), cudaMemcpyHostToDevice, streams[i]);
   cudaMemcpyAsync(&d_B[offset], &h_B[offset], eles_per_stream *
                            sizeof(int), cudaMemcpyHostToDevice, streams[i]);
   ……
   vector_sum<<<..., streams[i]>>>(d_A + offset, d_B + offset, d_C + offset);
   cudaMemcpyAsync(&h_C[offset], &d_C[offset], eles_per_stream *
                            sizeof(int), cudaMemcpyDeviceToHost, streams[i]);
}

for (int i = 0; i < nstreams; i++)
  cudaStreamSynchronize(streams[i]);
```

CUDA Events

- Timing asynchronous operations
  - Host-side timer: only measure the time for the call, not the actual time for the data movement or kernel execution
- Events to streams, which mark specific points in stream execution



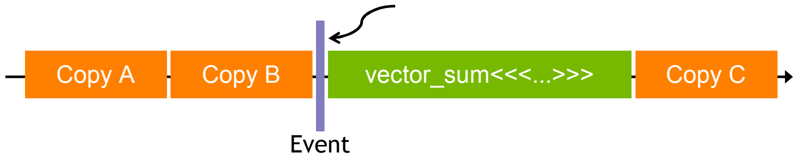
- Events are manually created and destroyed:

cudaError_t cudaEventCreate(cudaEvent_t *event);
cudaError_t cudaEventDestroy(cudaEvent_t *event);

CUDA Events

- To add an event to a CUDA stream:

  cudaError_t cudaEventRecord(cudaEvent_t event, cudaStream_t stream);



  – Event marks the point-in-time after all preceding actions in stream complete, and before any actions added after cudaEventRecord run

- Host to wait for some CUDA actions to finish

  cudaError_t cudaEventSynchronize(cudaEvent_t event);

  – Wait for all the operations before this events to complete, but not those after

CUDA Events

- Check if an event has been reached without waiting for it:

cudaError_t cudaEventQuery(cudaEvent_t event);

- Get the elapsed milliseconds between two events:

cudaError_t cudaEventElapsedTime(float *ms, cudaEvent_t start, cudaEvent_t stop); 성능측정!

CUDA Events

- In codes:

```
float time;
cudaEvent_t start, stop;
cudaEventCreate(&start); cudaEventCreate(&stop);

cudaEventRecord(start);
kernel<<<grid, block>>>(arguments);
cudaEventRecord(stop);
v cudaEventSynchronize(stop);

v cudaEventElapsedTime(&time, start, stop);
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

Implicit and Explicit Synchronization

- Two types of host-device synchronization:
  - **Implicit synchronization** causes the host to wait on the GPU, but as a side effect of other CUDA actions
  - **Explicit synchronization** causes the host to wait on the GPU because the programmer has asked for that behavior

Implicit and Explicit Synchronization

- Five CUDA operations that include implicit synchronization:
  1. A pinned host memory allocation (cudaMallocHost, cudaHostAlloc)
  2. A device memory allocation (cudaMalloc)
  3. A device memset (cudaMemset)
  4. A memory copy between two addresses on the same device (cudaMemcpy(..., cudaMemcpyDeviceToDevice))
  5. A modification to the L1/shared memory configuration (cudaThreadSetCacheConfig, cudaDeviceSetCacheConfig)

Implicit and Explicit Synchronization

- Four ways to explicitly synchronize in CUDA:
  1. Synchronize on a device

     cudaError_t cudaDeviceSynchronize();
  2. Synchronize on a stream

     cudaError_t cudaStreamSynchronize();
  3. Synchronize on an event

     cudaError_t cudaEventSynchronize();
  4. Synchronize across streams using an event

     cudaError_t cudaStreamWaitEvent(cudaStream_t stream, cudaEvent_t event);

Implicit and Explicit Synchronization

- **cudaStreamWaitEvent** adds inter-stream dependencies
  - Causes the specified stream to wait on the specified event before executing any further actions
  - event does not need to be an event recorded in stream

```
cudaEventRecord(event, stream1);
...
cudaStreamWaitEvent(stream2, event);
...
```

  - No actions added to stream2 after the call to cudaStreamWaitEvent will execute until event is satisfied

Suggested Readings

1. Chapter 6 in *Professional CUDA C Programming*
2. Justin Luitjens. *CUDA Streams: Best Practices and Common Pitfalls*. GTC 2014. http://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best- practices-common-pitfalls.pdf
3. Steve Rennich. *CUDA C/C++ Streams and Concurrency.* 2011. http://on-demand.gputechconf .com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf

Thanks !