CSC266 Introduction to Parallel Computing using GPUs
Heterogeneous Parallelism

Sreepathi Pai

November 15, 2017

URCS

Outline

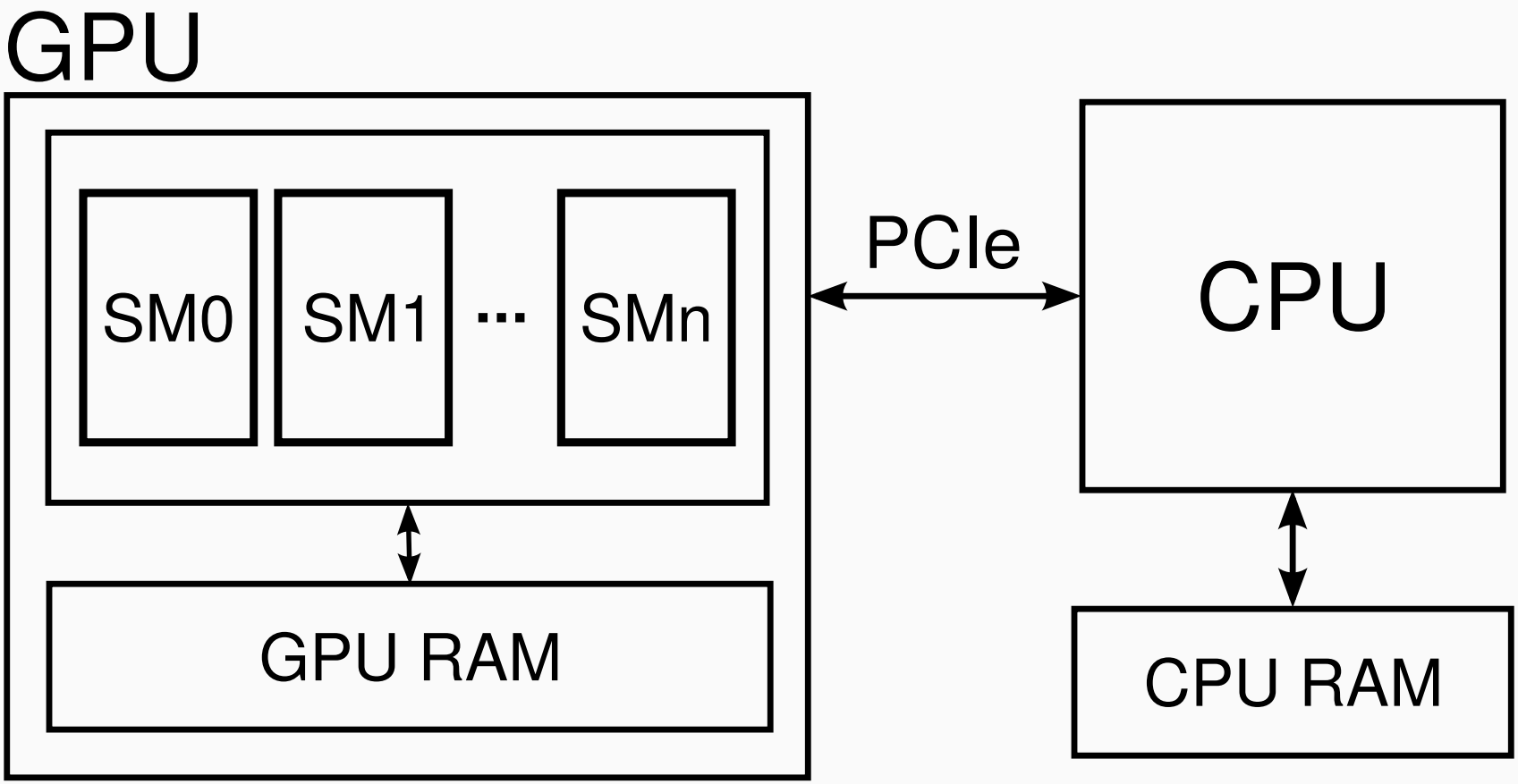
Heterogeneous Parallelism

Streams

Stream Synchronization

Outline

CPU/GPU systems



Using both the CPU and GPU

Reasons to use the CPU:

- Not enough parallelism
  - e.g. matrix multiply on small matrices
- Not SPMD
  - e.g. producer–consumer parallelism
  - task parallelism
- Legacy binary code (e.g. libraries)

Parallelism on the CPU

- Multiple Processes
  - Standard Unix way
  - `fork` and `join`
- Multiple Threads
  - Standard Windows way
  - Also `pthreads`
- And combinations thereof

Outline

The Default Stream

- Created automatically by CUDA
- All kernel launches and memory copies placed on default stream
    - If no stream specified
- All commands on the same stream execute in order

Default Streams in CUDA 7+

- Prior to CUDA 7, one default stream *per process*
  - All threads share the same stream
- After CUDA 7, *option* to have one default stream per thread
  - All threads have their own stream.
  - You have to opt-in during compilation `--default-stream per-thread`
  - These streams are still *blocking* streams

Creating Streams Explicitly

- Most GPU commands take an optional stream parameter
  - `kernel<<<blocks, threads, smem, stream>>>()`
  - `cudaMemcpyAsync(..., stream)`
- `stream` can be 0 (default) or explicitly created stream.
  - Also called *non-default* streams
- Creating non-default streams:

```
cudaCreateStream()
cudaCreateStreamWithFlags()
cudaCreateStreamWithPriority()
```

Terminology

- Non-blocking: CPU does not wait for operation to complete
  - All kernel calls
  - called "asynchronous" in NVIDIA docs
- Blocking: CPU waits for operation to complete
  - All memory copies involving host but not host *pinned* memory (even those marked `Async`)
  - called "synchronous" in NVIDIA docs
- Implicit Synchronization: Operation acts as a "barrier"
  - Operations on non-default streams vis-a-vis default stream
  - Example:

```
cudaStream_t a, b;

cudaCreateStream(&a)
cudaCreateStream(&b)

kernel_a<<<..., a>>>()  // Stream A
kernel_c<<<..., 0>>>()  // Default Stream
kernel_b<<<..., b>>>()  // Stream B
```

Non-blocking Streams

- Streams created with `cudaStreamCreate` implicitly synchronize with the default stream
  - Wait for all existing default stream operations to complete
  - Default stream waits for existing operations on streams to complete
- Non-blocking streams do not implicitly synchronize with default stream
  - Created with `cudaCreateStreamWithFlags`
  - What you usually want
  - Or avoid mixing default streams and non-default streams

```
cudaCreateStreamWithFlags(&a, cudaStreamNonBlocking)
cudaCreateStreamWithFlags(&b, cudaStreamNonBlocking)

kernel_a<<<..., a>>>()  // Stream A
kernel_c<<<..., 0>>>()  // Default Stream
kernel_b<<<..., b>>>()  // Stream B
```

Implicitly Serializing Operations

- Some CUDA API calls implicitly act as barriers
  - Block CPU until all operations on GPU are completed
  - Prevent any other operations from starting until API is completed
- Example:
  - `cudaMalloc`
  - No official list
- Diagnose by looking at timeline in NVProfiler

Stuff I'm not going to cover in detail

- Unified memory interactions with streams
  - Remember managed memory cannot be accessed on both CPU and GPU at same time
  - See `cudaStreamAttachMemAsync` documentation.
- Priority Streams
  - Kernels on high-priority streams "pre-empt" over running kernels
  - Only two levels of priority supported in Kepler
  - Not widely supported

Outline

Test for completion

- Have non-blocking items on stream finished?

  `cudaStreamQuery(stream)`

- returns `cudaSuccess` if all operations on stream are complete

Wait for completion (all)

- Wait for all non-blocking items on stream to finish?
- Waiting behaviour (can be changed):
  - Busy-wait
  - Yield

```
cudaStreamSynchronize(stream)
```

- You can also use `cudaDeviceSynchronize`
  - Not recommended

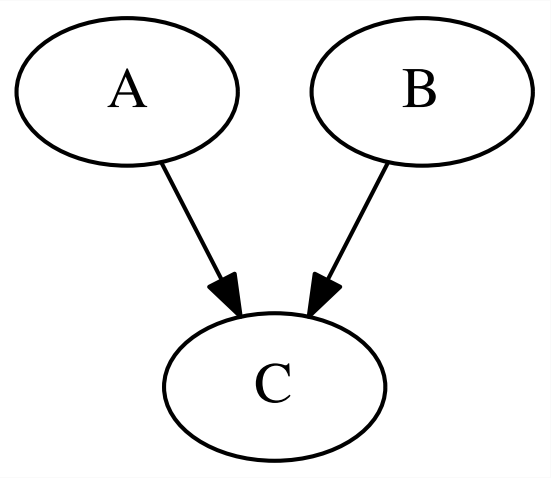
Targeted Waiting

- Waiting for specific events (e.g. `kernel_a` below)

```
cudaCreateStreamWithFlags(&a, cudaStreamNonBlocking);
cudaMallocHost(&repeat, sizeof(int));

do {
   *repeat = 0;

   kernel_a<<<..., a>>>(repeat)
   kernel_b<<<..., a>>>()

   cudaStreamSynchronize(a);

} while(*repeat > 0);
```

Synchronizing Across Streams

- Waiting in one stream for operations in *another* stream
  - A in stream 1
  - B in stream 2
  - C in ?
- Across different processes?
  - Not covered in course



Events

- Third kind of operation in stream
  - Other two are Kernels, memory copies
- Placed in order in stream with kernels and memory copies

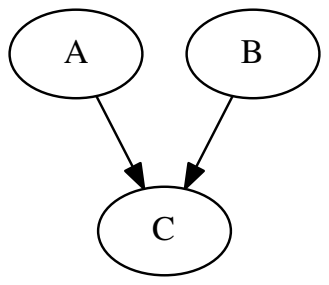
```
 ...
cudaEvent_t ev;

cudaCreateEventWithFlags(&ev, flags)
cudaCreateStreamWithFlags(&s, ...);
 ...

cudaEventRecord(ev, s); // places ev in stream s
```

- Interesting flags:
  - `cudaEventBlockingSync`: causes CPU to yield (instead of busy-waiting)
  - `cudaEventDisableTiming`: does not record timing data

Targeted Waiting with Events

```
cudaCreateStreamWithFlags(&a, cudaStreamNonBlocking);
cudaCreateEventWithFlags(&ev, cudaEventDisableTiming);
cudaMallocHost(&repeat, sizeof(int));

do {
   *repeat = 0;

   kernel_a<<<..., a>>>(repeat)
   cudaRecordEvent(ev, a); // asynchronous
   kernel_b<<<..., a>>>()

   cudaEventSynchronize(ev);
} while(*repeat > 0);

cudaStreamSynchronize(a); // wait for everything to complete
```

Waiting Across Streams




```
cudaCreateStreamWithFlags(&a, cudaStreamNonBlocking);
cudaCreateStreamWithFlags(&b, cudaStreamNonBlocking);

cudaCreateEventWithFlags(&ev, cudaEventDisableTiming);

A<<<...., a>>>();

B<<<...., b>>>();
cudaEventRecord(ev_b, b):

cudaStreamWaitEvent(a, ev_b); // places wait for ev_b in a
C<<<...., a>>>(); // note implicitly waits for a
```

Callbacks

- Can also create callbacks on stream operations
- Fourth type of operation in stream queue
- Called when all currently queued operations on a stream are completed
    - Blocks all further operations on stream until callback finished

```
f (cudaStream_t t, cudaError_t status, void *userData) {
  ...
}

A<<<..., a>>>();
cudaStreamAddCallback(a, f, NULL, 0); // B will execute after f
B<<<...., a>>>(); // placed on queue asynchronously
```

Further Reading

- How to Overlap Data Transfers in CUDA/C++