CSC367 Parallel computing

Lecture 21: CUDA Streams

Final test

- The class final test will be held on Monday, April 1st from 2 pm to 3 pm in EX320.
- All material not covered in the midterm will be included in the final. See Piazza for more information.
- Similar to the mid-term, the final will have a few multiple choice questions and a few descriptive questions.
- The test will not include the upcoming few slides on CUDA streams!

Remaining logistics

- Next Friday we will have a lecture on CLOUD COMPUTING! To encourage attendance we wont post slides online.
- Please fill the Scinet survey until Monday May 6th.
- I will keep the course Piazza instance alive even after the course to send emails about possible TA openings for this course next year and research and graduate positions that relate to parallel computing. So don't un-enroll if you are interested to hear about them!
- If you applied for summer or undergraduate research positions in my group, I will touch base with you in early May.
- If interested in grad school and our research, please email me!

Occupancy

- Hiding latency: when one warp is stalled, execute a different warp
- Need a metric related to how many active warps on a SM
  - Tells us how effectively the H/W is kept busy
- Occupancy = ratio of number of active warps per SM to max number of possible active warps
- Higher occupancy does not necessarily mean higher performance
  - At some point additional occupancy doesn't improve performance
  - However, low occupancy is always bad – poor memory latency hiding

Occupancy

- Register availability is a limiting factor – shared between all threads
  - Ability to hold local variables (with low-latency access) for lots of threads
  - If each block uses many registers, the number of blocks that can be resident on an SM is reduced => lowers the occupancy of the SM
  - e.g., 8192 registers per SM, 1024 threads resident per SM
    - => for 100% occupancy, each thread can only use 8 registers max
- Depends on CUDA compute capability
  - In older CUDA, occupancy calculator is basically a spreadsheet
    - Calculate occupancy based on block size and shared memory usage
  - In newer versions (after CUDA6.5), runtime functions for this
    - Occupancy calculator API: see documentation

Pinned Host Memory

- So far, allocate memory with cudaMalloc()
  - ... and life was beautiful
- CUDA runtime offers its own mechanism to allocate host memory
  - cudaHostAlloc() or cudaMallocHost()
  - Deallocate with cudaFreeHost()

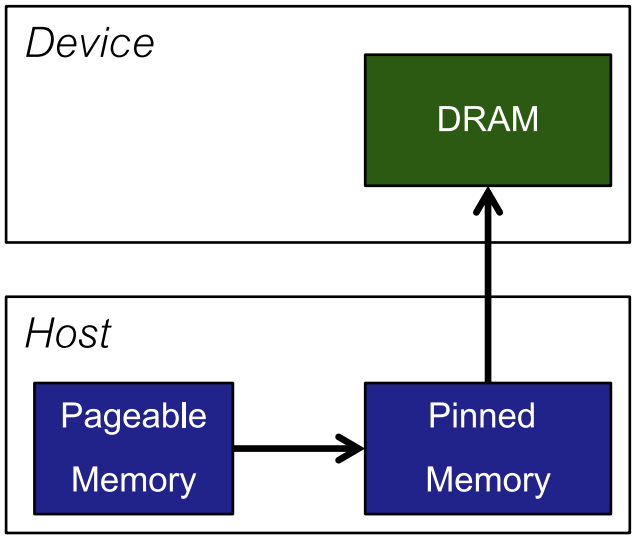
```
cudaError_t status = cudaMallocHost((void**)&h_mPinned, bytes);
if (status != cudaSuccess)
  printf("Error allocating pinned host memory\n");
...
cudaFreeHost(h_mPinned);
```

- Why not just use malloc()?
  - Host memory is subject to paging out to disk by the OS (more in CSC369)

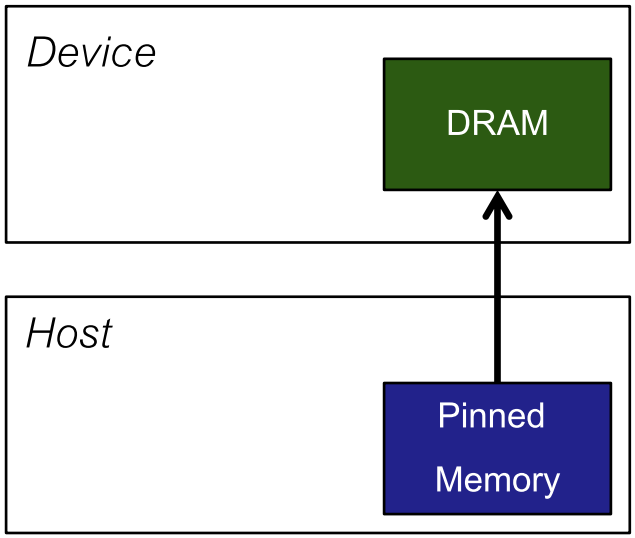
Why allocate pinned/locked memory?

- It's now safe to allow an application access to the physical address
- GPU can use DMA to copy data to or from the host (no CPU intervention)
- Pinned memory transfers are *typically* faster
- Besides, GPU cannot access data directly from pageable memory, pinned memory is used as a "staging area" (must pin before transfer)

Pageable data transfer



Pinned data transfer



CUDA Streams

- A sequence of operations that execute (on the device) in the order they are issued in the host code
  - e.g., copy host-to-device, kernel, copy device-to-host
- Operations in different streams can be run concurrently!
- All kernels and data transfers run in a stream
  - If no stream specified explicitly, the default stream is used
  - Default stream – slightly different semantics than other streams

Default stream

- Typical execution: copy in, launch kernel, copy out

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

- Copies are blocking, kernel is asynchronous!
- We could run independent host code during kernel execution

```
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
lots_o_computations<<<1,N>>>(d_a);
some_other_computations(); //runs on CPU in parallel with kernel
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

- Which one finishes first? Kernel or some_other_computations?
  - Doesn't matter, device-to-host copy must wait for kernel to finish anyway

Non-default streams

- Declare, create and destroy on host side (check errors as usual though)

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

- Data transfers in non-default streams: use cudaMemcpyAsync()
  - Similar to cudaMemcpy(), but non-blocking on the host
  - cudaMemcpy2DAsync() and cudaMemcpy3DAsync() variants exist (see docs)

```
cudaMemcpyAsync(d_a, a, N, cudaMemcpyHostToDevice, stream1);
```

- Launch a kernel in a non-default stream

```
lots_o_computations<<<1,N,0,stream1>>>(d_a);
```

- All operations (transfers and kernels) are non-blocking!

Streams and synchronization

- Might need to synchronize the host code with operations in a stream!
- Blunt way: cudaDeviceSynchronize()
  - Block the host code on this line until all previous device operations complete
  - Can be overkill, affects performance
- Another more subtle way:
  - cudaStreamSynchronize(str): block the host until all ops from 'str' complete

When can we overlap kernel with transfers?

- Device must be able to do "concurrent copy and execution"!
  - Check the deviceOverlap property of the cudaDeviceProp structure!
  - Most recent cards have this ability, since it's pretty crucial
- Kernel execution and data transfer must occur in different streams
  - Same stream => same order they are issued
- The host memory being transferred must be allocated as pinned
  - Otherwise can't issue async memcpy

Example: using multiple streams

- Here are 2 ways to issue the operations *from the host*
- Not the order of execution on the GPU though!
  - Within same stream, order is enforced
  - Can overlap operations across streams
  - Both produce correct results
  - What's the difference?
    - Must understand GPU scheduling!
    - Copy engines and kernel engines
    - Inter-engine: dependencies enforced
    - Intra-engine: execute in order of issue