Some *Advanced* Topics About GPGPU Programming

Wei Wang

Coordinating Host & Device

- Kernel launches are asynchronous
  - Control returns to the CPU immediately
- CPU needs to synchronize before consuming the results
  - cudaMemcpy(): Blocks the CPU until the copy is complete Copy begins when all preceding CUDA calls have completed
  - cudaMemcpyAsync(): Asynchronous, does not block the CPU
  - cudaDeviceSynchronize(): Blocks the CPU until all preceding CUDA calls have completed
- For consecutive kernel launches, it usually does not require cudaDeviceSynchronize(). Consecutive kernels are queued at GPU to be released one by one
  - cudaDeviceSynchronize() is used to ensure host and device code are synchronized.

Device Management

- Application can query and select GPUs
  - cudaGetDeviceCount(int *count)
  - cudaSetDevice(int device)
  - cudaGetDevice(int *device)
  - cudaGetDeviceProperties(cudaDeviceProp *prop, int device)
- Multiple host threads can share a device
- A single host thread can manage multiple devices
  - cudaSetDevice(i) to select current device
  - cudaMemcpy(…) for peer-to-peer copies

Thread Warps

- In CUDA, GPU scheduler always schedule threads in the units of warps
  - A warp usually has 32 threads; although in theory, this number can change based on GPU architecture
  - Only threads in the same block will be put into a warp
  - If there are fewer than 32 threads, idle threads will be scheduled and issued
- How thread warps are mapped to the SMs are usually not the concern of programmers, e.g.,
  - Each Nvidia Fermi GPU SM has 48 cores, 16 Load/Store units, 8 SFU and one warp scheduler
  - Each cycle, the warp scheduler may schedule two wraps (2*32 trheads) into the execution pipeline
  - It is up to the scheduler to decide what two wraps are schedule to maximize the utilization of hardware

Thread Warp cont'd

- Threads in one warp are executed in lock step, instruction by instruction – this is the nature of SIMD architecture
  - Branches in a warp introduces idling
  - E.g., a warp of threads execute the following code; thread 0-15 execute CODE_SEGMENT_1, thread 16-31 execute CODE_SEGMENT_2
    - First, thread 0-15 execute segment 1, and thread 16-31 runs idle
    - Second, thread 16-31 execute segment 2, and thread 0-15 runs idle

```
Some kernel:
    …
    if (condition){
        CODE_SEGMENT_1;
    }
    else{
        CODE_SEGMENT_2;
    }
    …
```

  - All threads in a warp should always execute the same code path, no divergence, to minimize idling

Thread Warp cont’d

- Because a thread warp must have 32 threads
  - If the thread count per block is not the multiple of 32, then idle threads will be scheduled and issued
  - Thread count per block should be close to the multiple of warp size
  - E.g., for a total of 320 threads, with each thread take 110 cycles to execute
    - 32 blocks and 10 threads/per blocks result in:
      - 32 wraps
      - each warp has 10 real threads and 22 idle threads;
      - A total of 32 * 110 = 3520 cycles
    - 10 blocks and 32 threads/per block result in:
      - 10 warps
      - each warp has 32 real threads and 0 idle threads;
      - A total of 10*110 = 1100 cycles

Thread Warp cont’d

- Threads in a warp should access the memory in the same pattern
  - Some load and store operations
  - Preferable accessing consecutive memory regions
- What threads are in the same warp?
  - From CUDA programming guide: "The way a block is partitioned into warps is always the same; each warp contains threads of consecutive, increasing thread IDs with the first warp containing thread 0."

Global Synchronization in CUDA

- Unfortunately, there is no easy way to do global synchronizations in CUDA at the moment
- Barrier only syncs threads within a block
- For global barrier, programmers usually resorts to multiple kernels
  - One kernel represents one step
  - Between kernels, tasks are synchronized back on CPU

Global Synchronization in CUDA cont'd

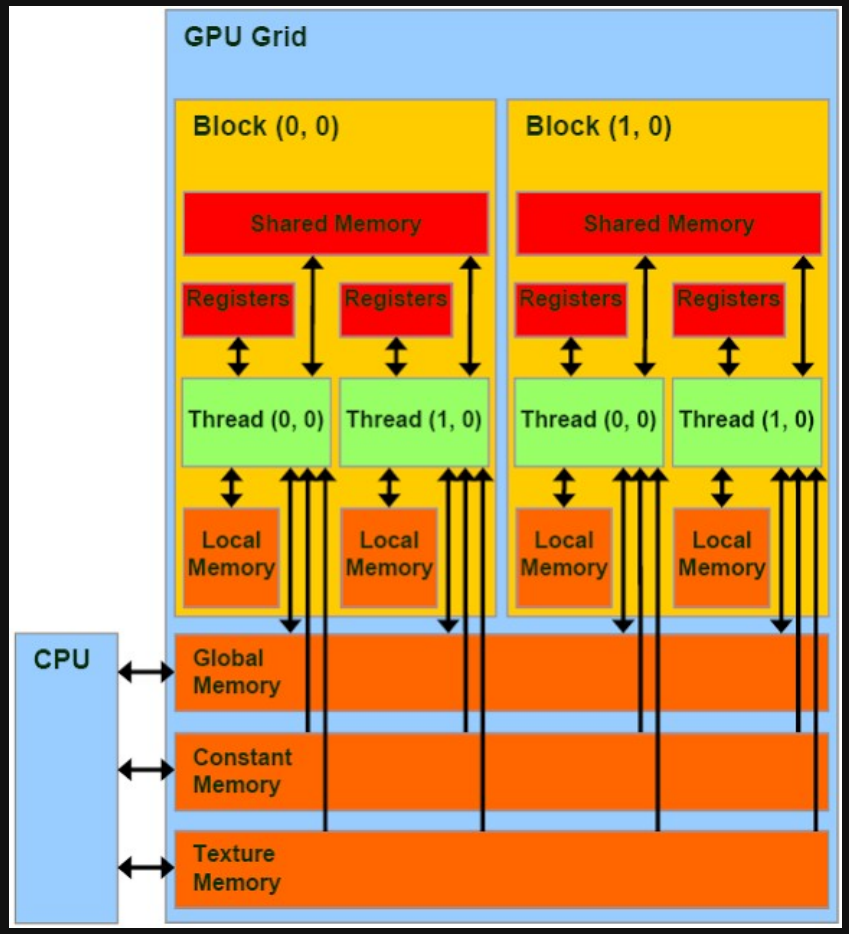
- CUDA provides global atomic operations, which theoretically allow implementation of global synchronization primitives
  - But implementations can be tricky
- CUDA has a memory fence
  - __threadfence()

Caching on GPU

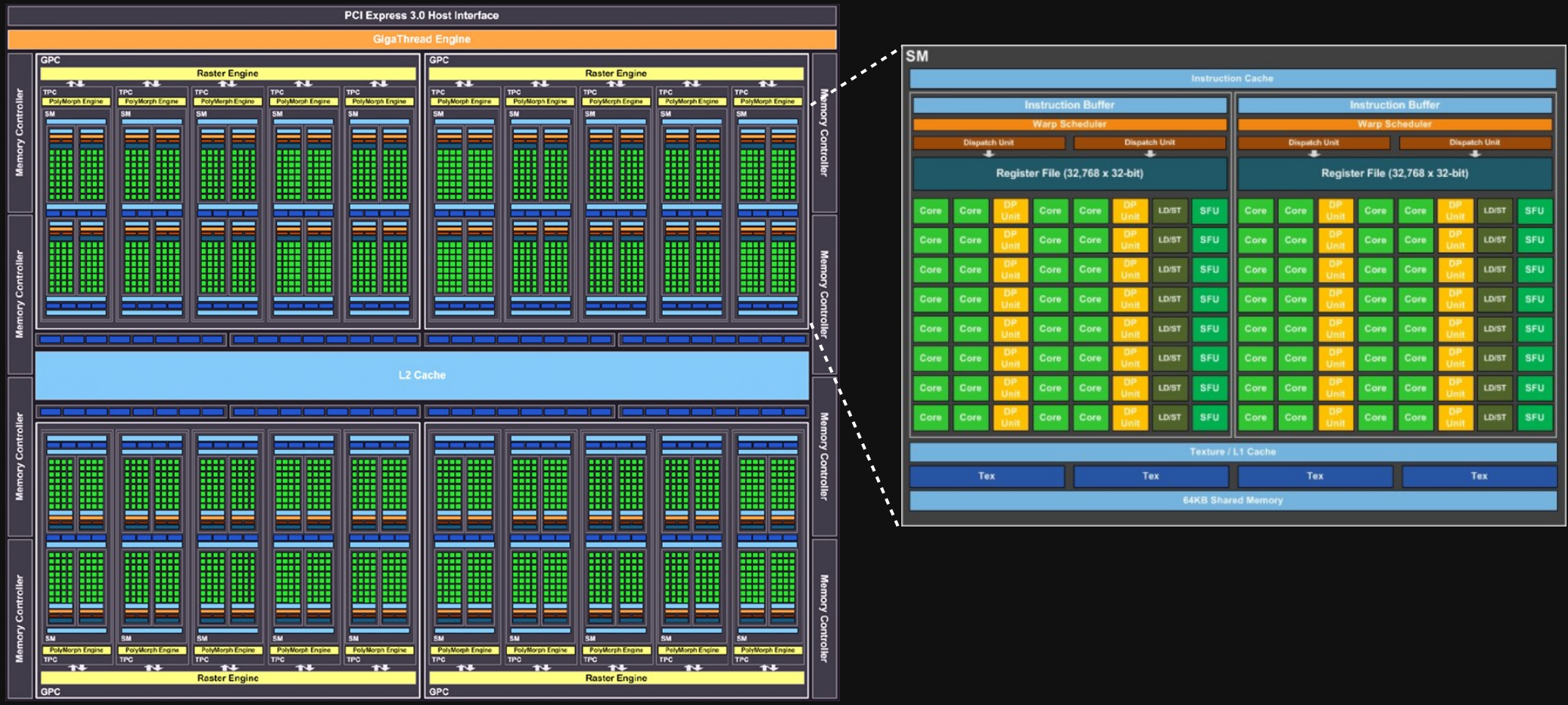
- Traditionally, GPU does not have hardware managed caches
  - Graphic applications do not need hardware managed caches
  - Saved transistors are devoted to CUDA cores
  - There are software managed caches: shared memory, texture cache and constant cache
- New generations of GPU provides L1 and L2 caches
  - Motived by GPGPU workloads
  - One L1 cache per SM, shared with shared memory or texture cache
  - One global L2 cache shared by all SMs

A Historical View of Memory Structure of GPU

- Shared memory:
  - Practically a software managed L1 cache
- Local memory:
  - a storage for local variables that cannot be put in registers
  - Originally not cached, now cached through new L1 and L2 cache
  - Today local memory is mostly a concept than a real storage
- Global memory:
  - Main memory of a GPU
  - Originally not cached, now cached through new L1 and L2 cache
- Constant memory:
  - Used to stored constant data, read-only
  - Can be cached in constant cache
  - Incorporated into main memory and L1/L2 cache in newer GPUs
- Texture memory:
  - Used to store read-only data
  - Can be cached in texture cache
  - Incorporated into main memory and L1/L2 cache in newer GPUs



Memory Structure of Current GPU (Nvidia Pascal)



Constant Memory

- Used to store read-only data
- Constant memory has limited size
- Constant memory data are cached in constant cache (now incorporated into L1/L2 caches)
  - Traditionally, most data are not cached, i.e., data are discarded after use
  - Reused data are declared as constant memory for fast reuse
- Constant memory data are declared with key word __constant__
- Although all data are cached now, GPU may still optimized read-only operations.
  - Therefore, it may still be beneficial to use constant memory

Texture Memory

- Used to store texture data for graphic applications
  - Read-only data
- Texture memory data are cached in texture cache (now incorporated into L1/L2 caches) for fast access
  - Unlike constant memory, texture memory data are expressed in 1D, 2D or 3D arrays to represent 2D/3D data locality
  - 1D/2D/3D Data are preloaded to texture cache to improve performance
- Texture memory data are declared with texture keyword, and need to be explicitly bound with data in main memory using function cudaBindTexture
- Although all data are cached now, GPU may still optimized texture-like memory reads
  - Therefore, it may still be beneficial to use texture memory

Programming with OpenCL

- OpenCL is another famous framework for programming heterogeneous hardware, including GPU
- OpenCL shares similar concepts with CUDA, e.g.,
  - Routines running on GPU are also called kernels
  - CUDA warp/thread are called weavefront/work-items on OpenCL
  - Similar functions to copy data between CPU and GPU memory
- OpenCL aims at supporting more than just GPU, including FPGA and DSPs
- CUDA is for Nvidia GPU
- The actual performance of OpenCL and CUDA depends on hardware support, library implementation quality and application code quality

ROCm

- AMD’s alternative to CUDA
  - Radeon Open Compute ecosystem
- Understands OpenCL code
- HIP
  - A common programming interfaces that supports CUDA and ROCm
  - Syntax very similar to CUDA