CSC266 Introduction to Parallel Computing using GPUs
Introduction to CUDA

Sreepathi Pai

October 18, 2017

URCS

Outline

Background

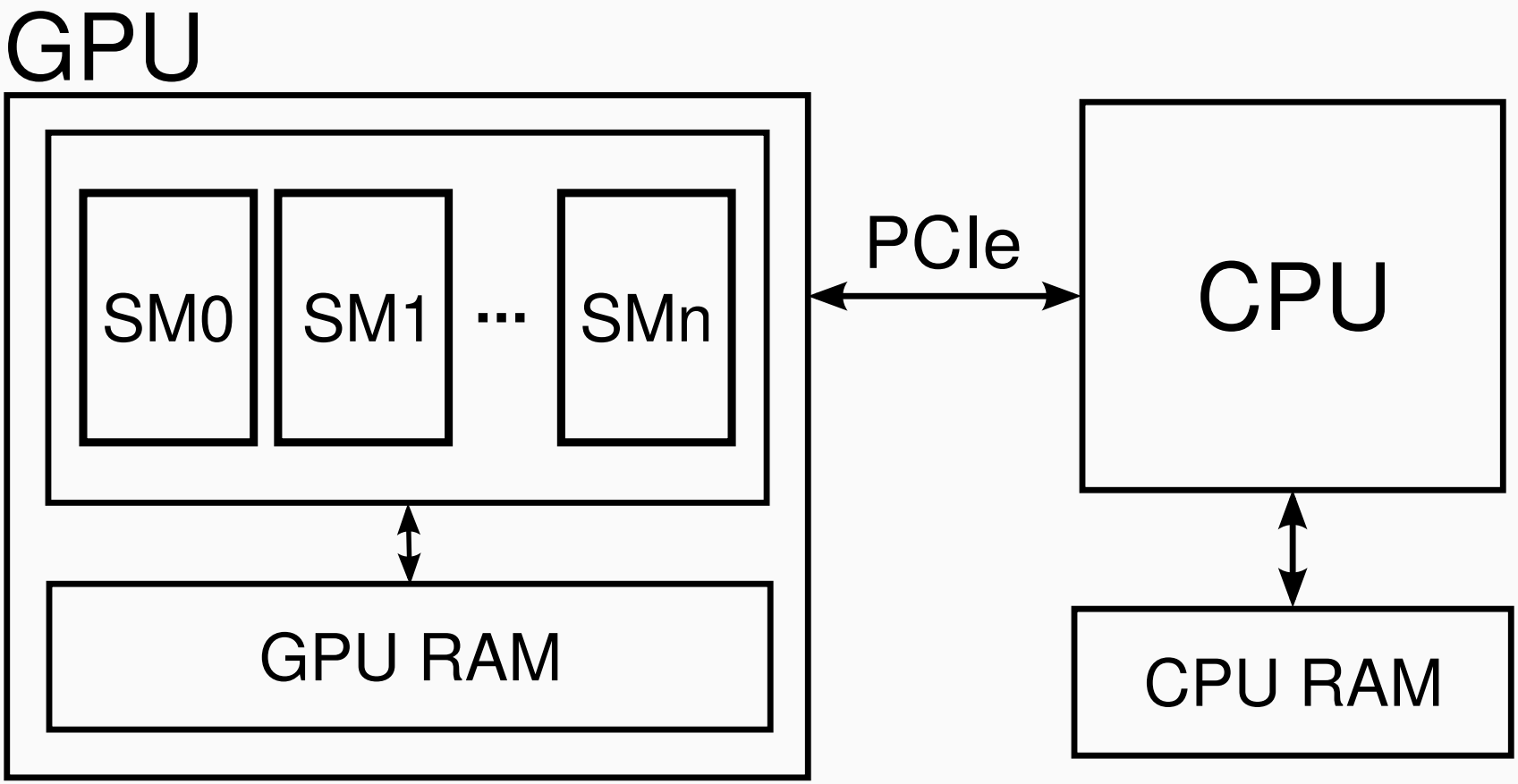
Memory

Code

Execution Model

Outline

CUDA programmer's view of the system



Data access in Shared memory vs Distributed systems

- Shared Memory system
  - Same address space
  - Data in the system accessed through load/store instructions
  - E.g., multicore
- Distributed Memory System (e.g. MPI)
  - (Usually) different address space
  - Data in the system accessed through message-passing
  - E.g., clusters

Is a GPU-containing system a distributed system?

- Does data live in the same address space?
- Is data in the entire system accessed through load/store instructions?

Outline

Pointers

- Addresses are contained in pointers
- GPU addresses are C/C++ pointers in CPU code
  - True, in CUDA
  - False, in OpenCL (`cl::Buffer` in CPU)

Allocating Host Memory

- Data lives in CPU memory
- Read/Written by CPU using load/store instructions
- Allocated by `malloc` (or equivalent)
- Freed by `free` (or equivalent)
- Pointers cannot be deferenced by GPU

Allocating GPU Memory

- Data lives in GPU memory
- Read/Written by GPU using load/store instructions
- Allocated by `cudaMalloc` (or `cudaMallocManaged`)
- Freed by `cudaFree`
- Pointers cannot be deferenced by CPU
- Data transferred using copies (`cudaMemcpy`)

Allocating Pinned Memory

- Data lives in CPU memory
- Read/Written by CPU using load/store instructions
- Read/Written by GPU using load/store instructions over PCIe bus
- Same pointer value
- Allocated by `cudaMallocHost` (or `cudaHostMalloc`)
- Freed by `cudaFree`
- No transfers needed!

Mapping Host-allocated Memory to GPUs

- Data lives in CPU memory
- Read/Written by CPU using load/store instructions
- Read/Written by GPU using load/store instructions over PCIe bus
- Allocated by `malloc`
- Mapped by `cudaHostRegister`
- GPU uses different pointer (`cudaHostGetDevicePointer`)
- Freed by `free`
- No transfers needed!

Managed Memory

- Data lives in CPU memory or GPU memory
- Read/Written by CPU using load/store instructions
- Read/Written by GPU using load/store instructions
- But not by both at the same time!
- Same pointer value
- Freed by `cudaFree`
- No manual transfers needed!
- Data transferred "automagically" behind scenes

Summary

Pointer from	Host	GPU	Same Pointer
CPU `malloc`	Y	N	N
`cudaMalloc`	N	Y	N
`cudaHostMalloc`	Y	Y	Y
`cudaHostRegister/GetDevicePointer`	Y	Y	N
`cudaMallocManaged`	Y	Y	Y

Outline

Host code and device code

- CPU code callable from CPU (`__host__`)
- GPU code callable from CPU (`__global__`)
- GPU code callable from GPU (`__device__`)
- Code callable from both CPU and GPU (`__host__`, `__device__`)
- CPU code callable from GPU (N/A)

CUDA source code layout

```
__global__
void vector_add(int *a, int *b, int *c, int N) {
    ...
}

int main(void) {
   ...
   vector_add<<<...>>>(a, b, c, N);
}
```

CUDA Compilation Model (Simple)

- All code lives in CUDA source files (`.cu`)
- `nvcc` compiler separates GPU and CPU code
  - Inserts calls to appropriate CUDA runtime routines
- GPU code is compiled to PTX or binary
  - PTX code will be compiled to binary at runtime
- CPU code is compiled by GCC (or clang)

Fat binary

- End result of `nvcc` run is a single executable
  - On Linux, standard ELF executable
- Contains code for both CPU and GPU
- CUDA automatically sets up everything
  - OpenCL does not
  - No OpenCL equivalent of `nvcc`

Outline

Vector Addition again

```
__global__
void vector_add(int *a, int *b, int *c, int N) {
    ...
}

int main(void) {
   ...
   vector_add<<<...>>>(a, b, c, N);
}
```

Execution starts on the CPU

- Program starts in `main`, as usual
- On first call to CUDA library, a GPU context is created
  - GPU Context == CPU Process
  - Can also create one automatically
- Default GPU is chosen automatically *per thread*
  - If multiple GPUs
  - Usually the newest, ties broken by the fastest
  - This is where default allocations and launches occur
  - Can be changed *per thread* (`cudaSetDevice`)

Memory Allocation and Copies

- `cudaMalloc`, etc. used to allocate memory
  - CPU waits for allocation
- `cudaMemcpy`, etc. used to copy memory across
  - CPU waits by default for copy to finish
  - LATER LECTURES: non-blocking copying APIs

Launch

- Determine a thread block size: say, 256 threads
- Divide work by thread block size
  - Round up
  - $\lceil N/256 \rceil$
- Configuration can be changed *every* call

```
int threads = 256;
int Nup = (N + threads - 1) / threads;
int blocks = Nup / threads;

vector_add<<<blocks, threads>>>(...)
```

Kernel Launch Configuration

- GPU kernels are SPMD kernels
  - Single-program, multiple data
  - All threads execute the same code
- Number of threads to execute is specified at launch time
  - As a *grid* of B *thread blocks* of T threads each
  - Total threads: $B \times T$
- Reason: Only threads within the same thread block can communicate with each other (cheaply)
  - Other reasons too, but this is the only algorithm-specific reason

Distributing work in the kernel

```
__global__
vector_add(int *a, int *b, int *c, int N) {
   int tid = threadIdx.x + blockIdx.x * blockDim.x;

   if(tid < N) {
      c[tid] = a[tid] + b[tid];
   }
}
```

- Maximum 2^{32} threads supported
- *gridDim*, *blockDim*, *blockIdx* and *threadIdx* are CUDA-provided variables

Blocking and Non-blocking APIs

- Blocking API (or operation)
  - CPU waits for operation to finish
  - e.g. simple `cudaMemcpy`
- Non-blocking API (or operation)
  - CPU does not wait for operation to finish
  - e.g. kernel launches
  - You can wait explicitly using special CUDA APIs

Helpful Tips

- Each CUDA API call returns a status code
  - Check this *always*
  - If an error occurred, this will contain error code
  - Error may be related to this API call or *previous non-blocking API calls!*
- Use `cuda-memcheck` tool to detect errors
  - Slows down program, but can tell you of many errors