CUDA SKILLS

Yu-Hang Tang

June 23-26, 2015

CSRC, Beijing

- **day1.pdf** at **/home/ytang/slides**
- Referece solutions coming soon

Online CUDA API documentation

http://docs.nvidia.com/cuda/index.html

RECAP

GPUs are massively parallel, energy-efficient processors

There are a variety of ways to harness the power of GPUs

- Language extensions, directives, libraries, scripts

NVCC is the C++ compiler for CUDA GPUs

Kernels are functions that run in parallel on GPUs

- need to be launched from host CPU (or otherwise by Dynamic Parallelism)

Threads are organized in a grid of blocks

THE SIMT ARCHITECTURE

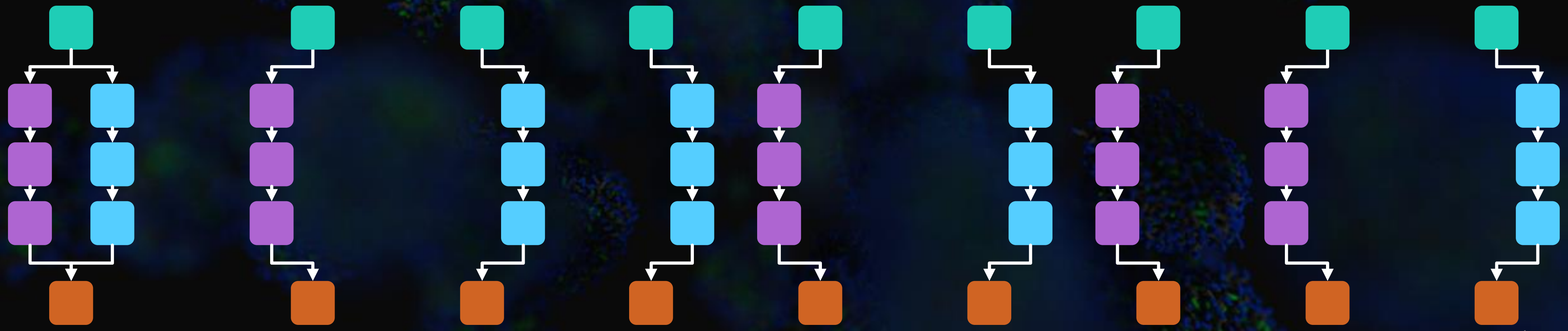
- Software
  - Kernels run in **warps** of 32 parallel threads
  - All threads in a warp must execute the same instruction at any given time
- Hardware – Kepler architecture
  - Each GPU contains several, e.g. 15, **Stream Multiprocessors** (SMs)
  - Each SM contains 192 cores, divided into 6 groups (i.e. 32 cores per group)
  - Each SM can hold up to 2048 CUDA threads, i.e. 64 warps
    - 2 blocks if block size is 1024, 4 blocks if block size is 512, etc…
  - For each cycle, the SM can pick up to 4 warps and issue up to 2 instructions per warp
- Why: to hide instruction & memory latency

OCCUPANCY

- Occupancy = number of actual threads per SM / max number of threads per SM
- Why high occupancy is crucial for reaching peak performance: to hide latency
  - Instruction latency: 9+ cycles
  - Memory latency: 8 - 2000+ cycles
  - CUDA core execute instructions **in-order**
- How much occupancy do we actually need?

BRANCH DIVERGENCE

- A warp can only execute **ONE** common instruction at a time
- Divergence happens when threads of a warp disagree on their execution path due to data-dependent conditional branch
  - if, for, while/do, switch
- Warp serially executes each branch path, disabling threads that are not on that path until all paths complete



QUIZ

- Divergent or not?

```
__global__ void foo( ... ) {
    if ( threadIdx.x % 2 ) {
        ...
    } else {
        ...
    }
}
```

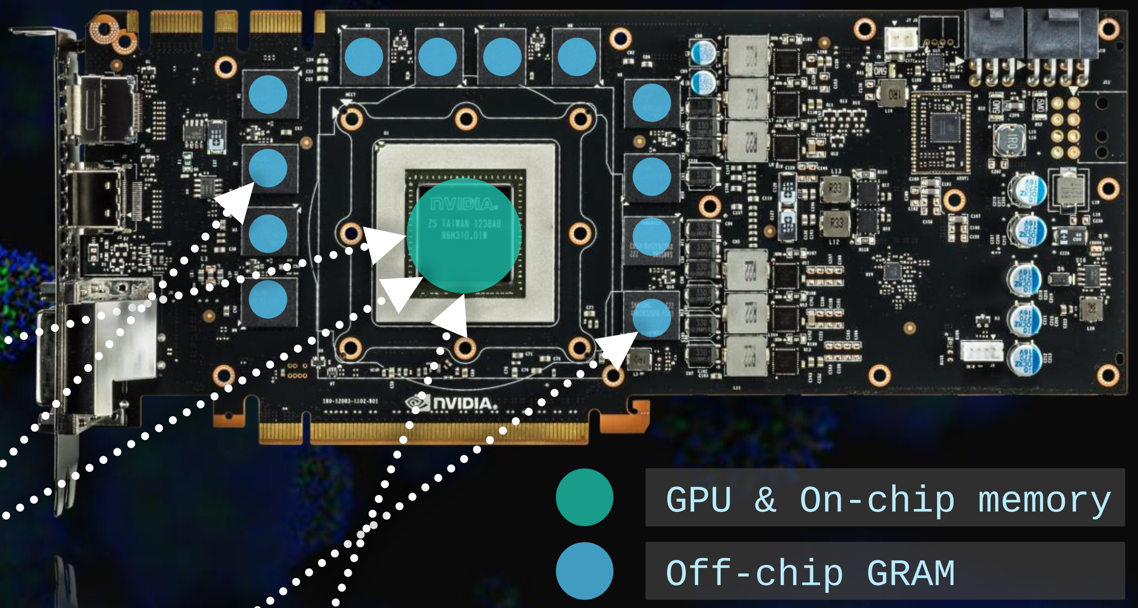
```
__global__ void foo( int *bar ) {
    if ( bar[threadIdx.x] ) {
        ...
    } else {
        ...
    }
}
```

```
__global__ void foo( ... ) {
    if ( ( threadIdx.x / warpSize ) % 2 ) {
        ...
    } else {
        ...
    }
}
```

```
__global__ void foo( int *bar ) {
    int tid = threadIdx.x;
    for( int i = 0; i < bar[tid]; i++ ) {
        ...
    }
}
```

NOT ALL MEMORIES ARE BORN EQUAL

- HW
  - Register files
  - On-chip L1/L2 cache
  - On-chip texture units
  - Off-chip GRAM
- SW
  - registers
  - per-thread private local memory
  - per-block shared memory
  - global memory: accessible to all threads
  - constant and texture cache: global read-only access



GPU MEMORY FACT SHEET

	Reg(32bit)	Global	Shared	Const	Texture
Capacity	255/thread	2-12 GB	16-48 KB/SM	~10s KB	~10s KB
Latency	2	~1000	8	8 (hit)	~60 (hit)
Bandwidth	-	High	Very High	Low	Very High
Scope	thread	global	block	global	global

GLOBAL MEMORY

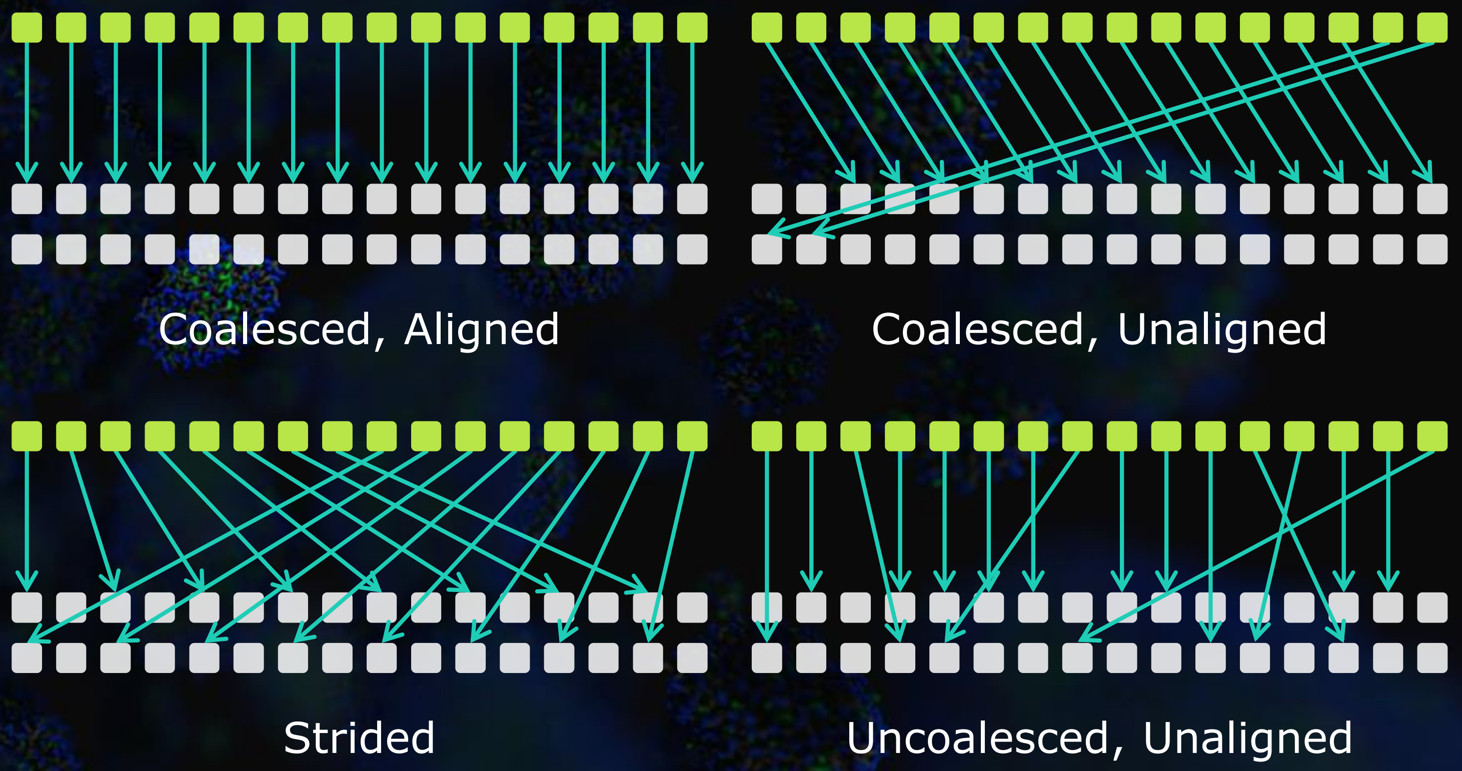
- Allocatable from host/device
  - `cudaError_t cudaMalloc ( void** devPtr, size_t size );`
  - `cudaError_t cudaFree ( void* devPtr ) ;`
  - `device-side malloc/new/free/delete`
- Accessible from device

```
ptr[ index ] = value;
```

- Copiable from host
  - `cudaError_t cudaMemcpy ( void* dst, const void* src, size_t count, cudaMemcpyKind kind );`
  - `cudaError_t cudaMemset ( void* devPtr, int  value, size_t count );`
- UVA
  - Single address space for the host and all devices.

ACCESS PATTERN

- Coalesced: adjacent threads access consecutive memory locations
- Aligned: Starting address of memory access is multiple of 32 bytes (write, non-caching read) or 128 bytes (caching read)
- Strided: memory access spaced uniformly



QUIZ

- Coalesced or not?

```
__global__ void foo( int *bar ) {
    bar[thread_id()] = ...;
}
```

```
__global__ void foo( double *bar ) {
    double e = bar[thread_id()+16];
}
```

```
__global__ void foo( int *bar ) {
    bar[thread_id()+8] = ...;
}
```

```
__global__ void foo( int2 *bar ) {
    int e = bar[thread_id()].x;
}
```

```
__global__ void foo( int *bar ) {
    bar[thread_id()+13] = ...;
}
```

```
__global__ void foo( float4 *bar ) {
    float e = bar[thread_id()].z;
}
```

```
__global__ void foo( int *bar ) {
    int e = bar[thread_id()+16];
}
```

```
__global__ void foo( int *map, int *bar ) {
    int e = bar[ map[thread_id()] ];
}
```

CONSTANT MEMORY

- Const memory
  - __constant__
  - low latency on hit, low bandwidth (broadcast only)
  - const *: compiler automatically offload to constant cache
- Non-coherent cache
  - const __restrict
  - high bandwidth, medium latency
  - compiler automatically offload to non-coherent cache

SHARED MEMORY

- Shared
  - visible to all threads of the block and within the lifetime of the block.
  - allocated using the __shared__ qualifier
  - much faster than global memory
  - banked access, broadcasting
- Static allocation
  - __shared__ int array[32];
- Dynamical allocation
  - <<<numBlocks, threadsPerBlock, **sharedMemSize** >>>
- Template allocation

READ-ONLY DATA CACHE (TEXTURE CACHE)

- Underlying memory region is assumed to be immutable during kernel launch.
  - 48 KB per SM
- Better random access performance
  - 32-byte load granularity, cached
  - hardware takes care of multi-dimensional data locality
- Usage

```
__global__ void foo( int *bar, int *map ) {
    int x = __ldg( bar + map[ threadIdx.x ] );
}

__global__ void foo2( const int* __restrict bar, int *map ) {
    int x = bar[ map[ threadIdx.x] ];
}
```

EXAMPLE 5: IMAGE FILTERING REVISITED

- Shared memory version
  - copy tiles to shared memory first
- __syncthreads()
- Non-coherent cache version
  - decorate image as const * __restrict

ATOMICS

- Race condition

```
__shared__ int sum;
int b = ...;
sum += b;
```

```
__shared__ int sum;
int b = ...;
register r = sum;
r += b;
sum = r;
```

```
__shared__ int sum;
int b0 = ...;
register r0 = sum;
r0 += b0;
int b1 = ...;
register r1 = sum;
sum = r0;
r1 += b1;
sum = r1;
```

- Atomicity: a guarantee that the operation will be performed without interference from other threads.
- Performs a read-**modify**-write atomic operation on one 32-bit or 64-bit word residing in global or shared memory
  - `modify = add, sub, exchange, etc...`
- Only atomicExch() and atomicAdd() for `float` values

WARP SHUFFLE

- Exchange a variable between threads within a warp (C.C. > 3.0)
- Signature
  - `type __shfl(type var, int srcLane, int width=warpSize);`
  - type = int / float

`__shfl()`	Direct copy from indexed lane
`__shfl_up()`	Copy from a lane with lower ID relative to caller
`__shfl_down()`	Copy from a lane with higher ID relative to caller
`__shfl_xor()`	Copy from a lane based on bitwise XOR of own lane ID

EXAMPLE 6: PARALLEL REDUCTION

- Reduction: a **summary** of data
  - summary = summation, mean, max, min, etc.
- Parallel summation: $S_n = \sum_{i=0}^{n-1} a_i$
  - The serial way: `for(int i = 0 ; i < n ; i++) sum += a[i];`
  - How to reduce in parallel?

Thank you for coming to this workshop!

ACKNOWLEDGEMENT

Y.H.T. appreciate invitation from Professor Wei Cai and support from CSRC.