A Brief Overview of the World of Parallel Programming

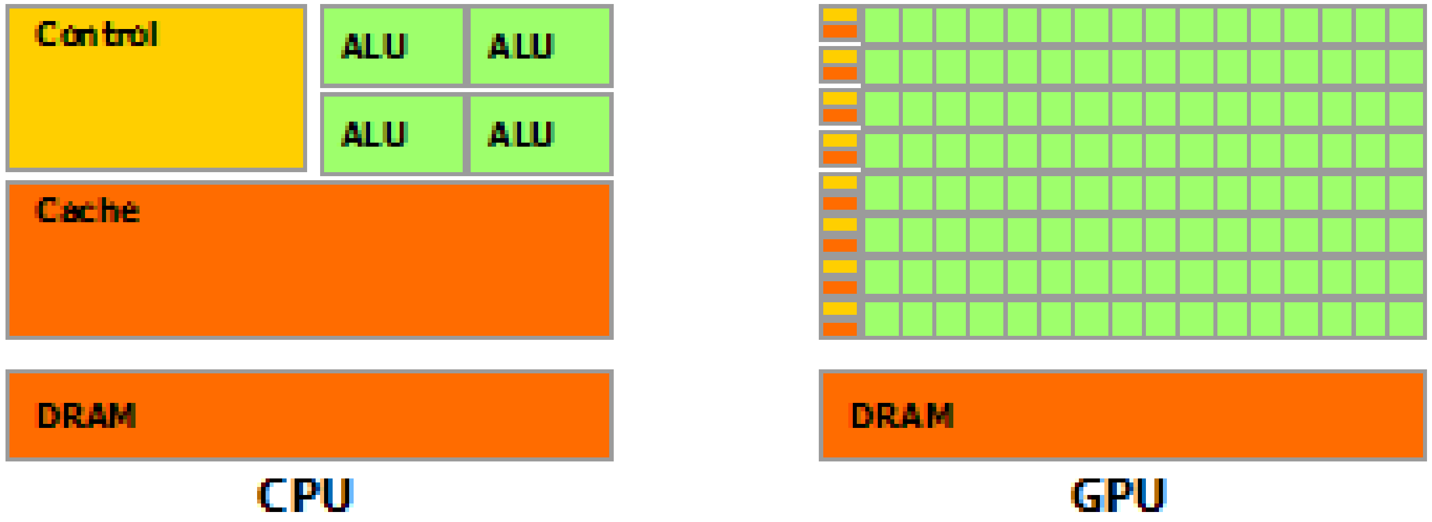
Mike Elliott

mre@m79.net

CPUs and GPUs

- CPU – Core
  - Local cache
  - Registers
  - control unit
- GPU – Compute unit
  - Cache
  - Registers
  - SIMD
  - Threading

CPU vs GPU Organization



CPU – Low Latency

- Large cache – on chip
- Sophisticated control
  - Branch prediction
  - Data forwarding
- Powerful ALU

GPU – High Throughput

- Throughput oriented design
- Small caches
- Simple control
- Energy efficient ALUs
- Massive number of threads to tolerate latencies

Heterogeneous Programming

- Use both CPU and GPU
- Use CPU for serial parts
  - Where latency matters
- Use GPU for parallel parts
  - Where throughput matters

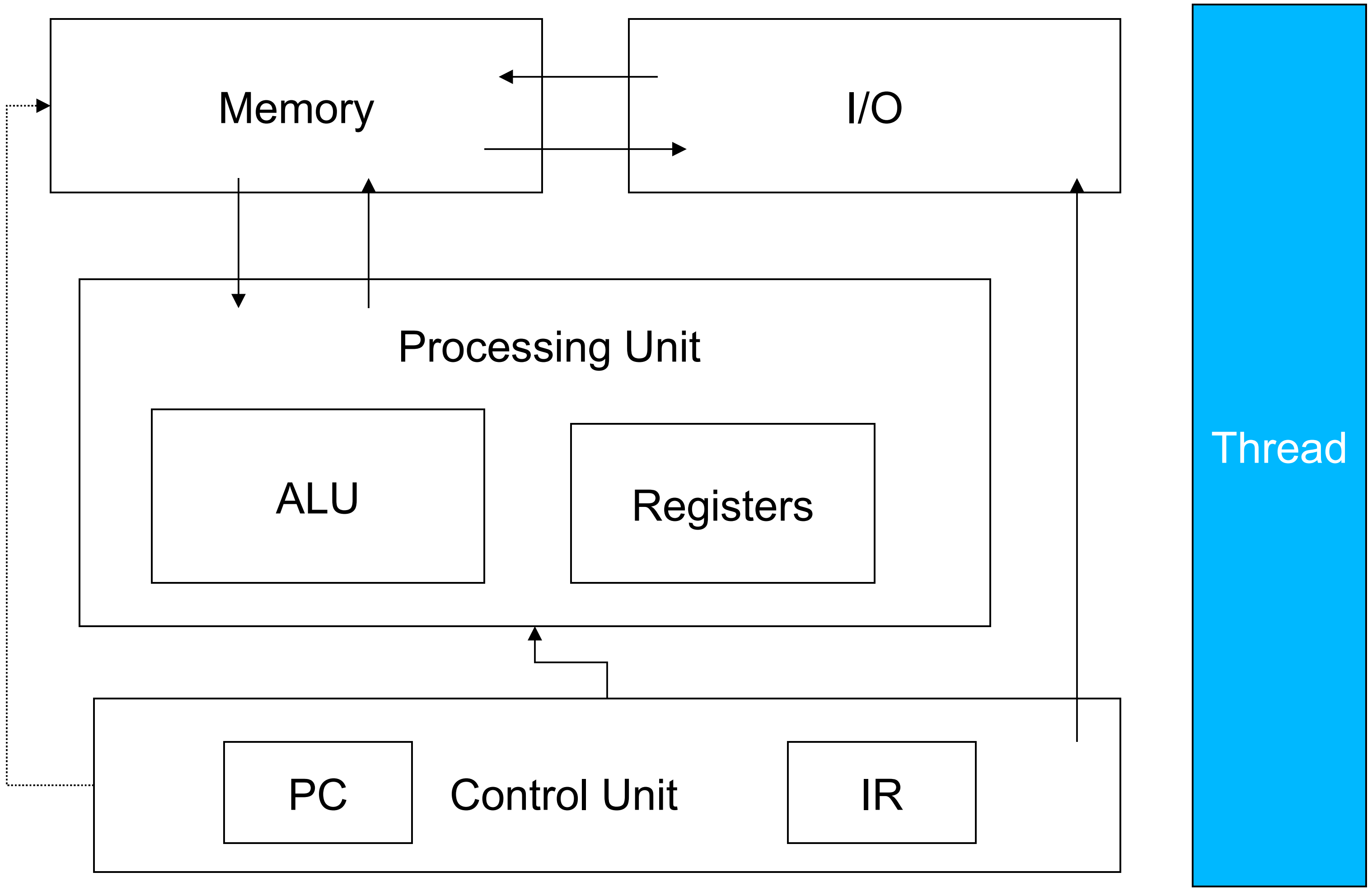
CUDA C

- Compute Unified Device Architecture
  - Created by Nvidia
- Hierarchical thread organization
- Heterogeneous host + device programming
  - Serial code host
  - Parallel code (kernel) device

CUDA Architecture

- Three dimensional grid
  - This constitutes the “kernel”
- Three dimensional blocks in the grid
  - Holds local memory and threads
  - Maximum threads determined by hardware
- Threads of execution
  - “Warp” of 32 threads

Von Neumann Architecture



SPMD

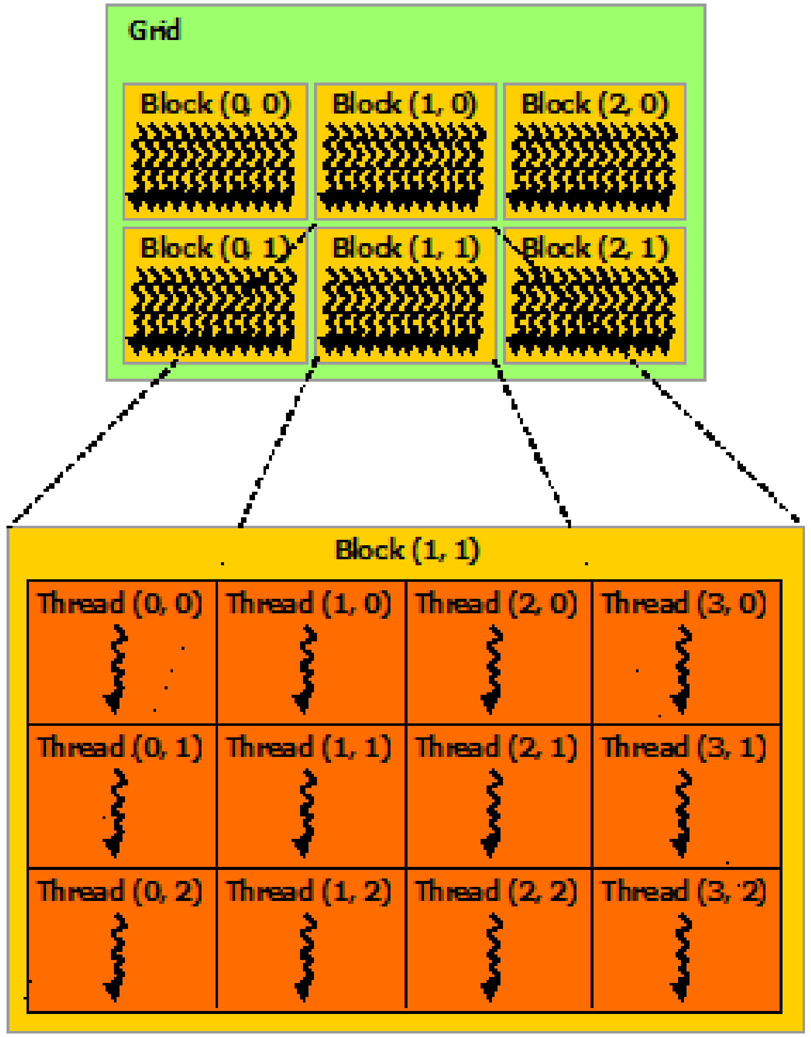
- Single Program, Multiple Data
- All threads run the exact same program
- Each thread knows its indices
  - Compute memory addresses
  - Make control decisions
  - int ndx = blockIdx.x * blockDim.x + threadIdx.x

Thread Blocks

Threads are organized into blocks

- Threads within a block cooperate via
  - Shared memory
  - Synchronization
  - Atomic operations
- Threads in different blocks cannot affect each other

Grid of Thread Blocks



CUDA Function Declarations

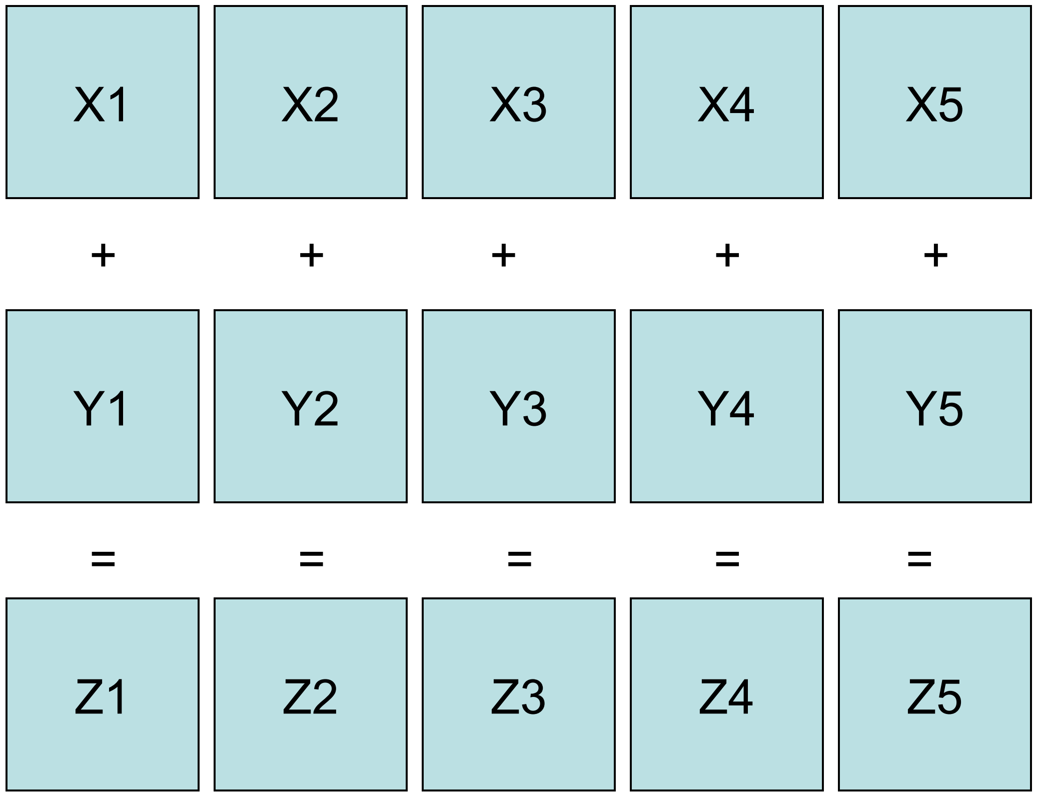
	executed	callable
__device__ float deviceFunction()	device	device
__global__ void kernelFunction()	device	host
__host__ float hostFunction()	host	host

__global__ denotes a kernel function

must return void

__device__ and __host__ can be used together

Vector Addition



Vector Addition (kernel)

```
// CUDA C
//
__global__
void vecAddKern( float* result, float* a, float* b, int
    count) {
  const int ndx = blockIdx.x * blockDim.x +
    threadIdx.x;
  if (ndx < count)
     result[ndx] = a[ndx] + b[ndx];
}
```

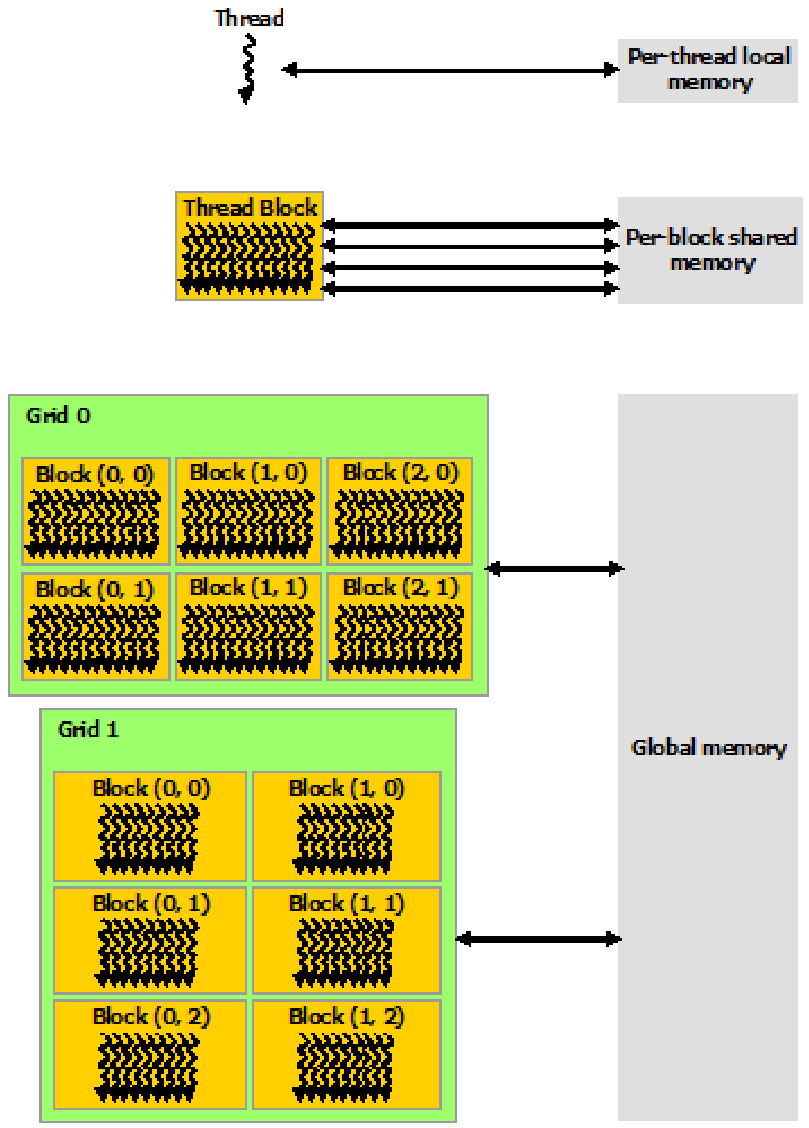
Vector Addition (host)

```
// CUDA C
__host__
void vecAdd( float* result, float* a, float* b, int
    count) {
   // magic to allocate memory, etc., here
  dim3 dimGrid = (ceil(count/256), 1, 1 );
  dim3 dimBlock = (256, 1, 1 );
   vecAddKern<<<dimGrid, dimBlock>>>( rr, aa,
    bb, count);
  // magic to deallocate memory, store result, etc.,
    here
}
```

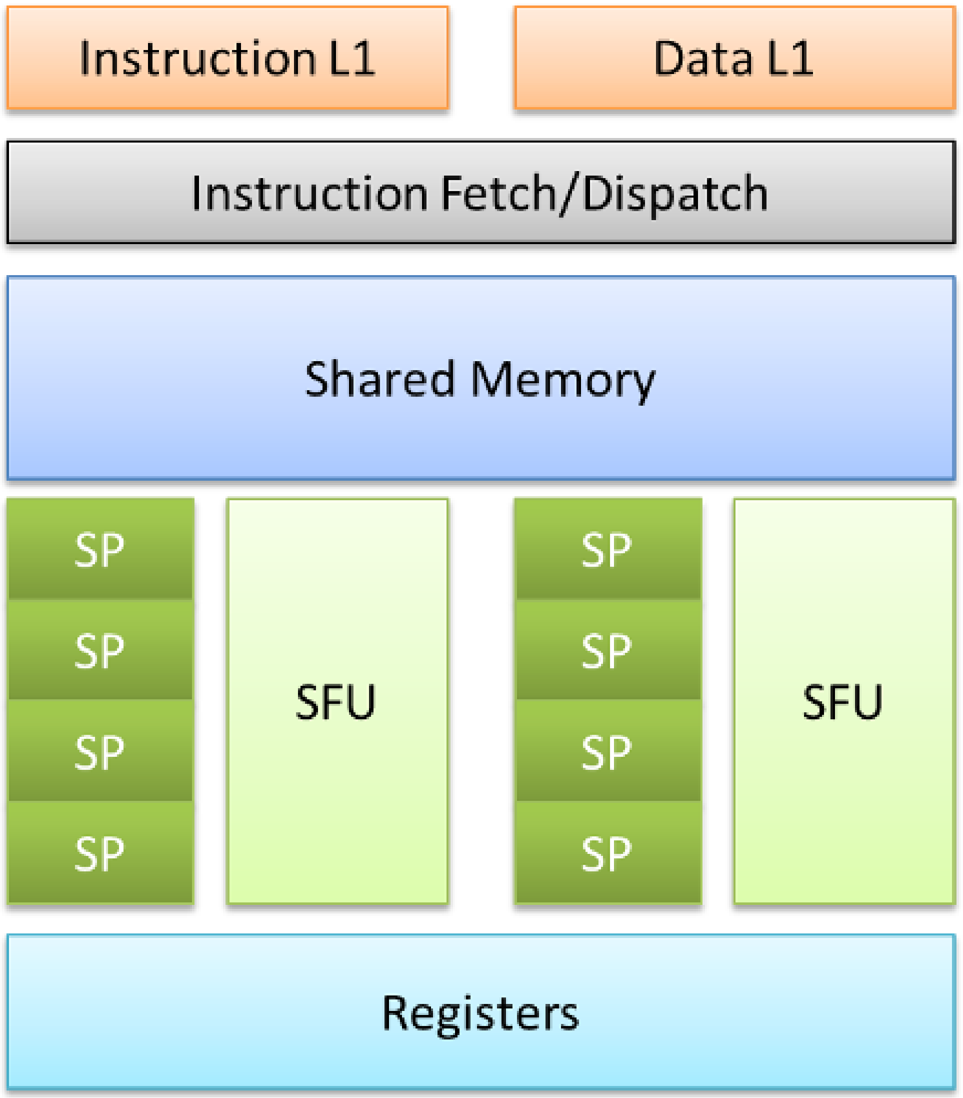
Host / Device Memory

- Host can transfer data to and from device
- Within a device, blocks can access device global memory
- Within a block, threads can access block local memory
- Within a thread, a set of registers exists which are local to that thread

Memory Hierarchy



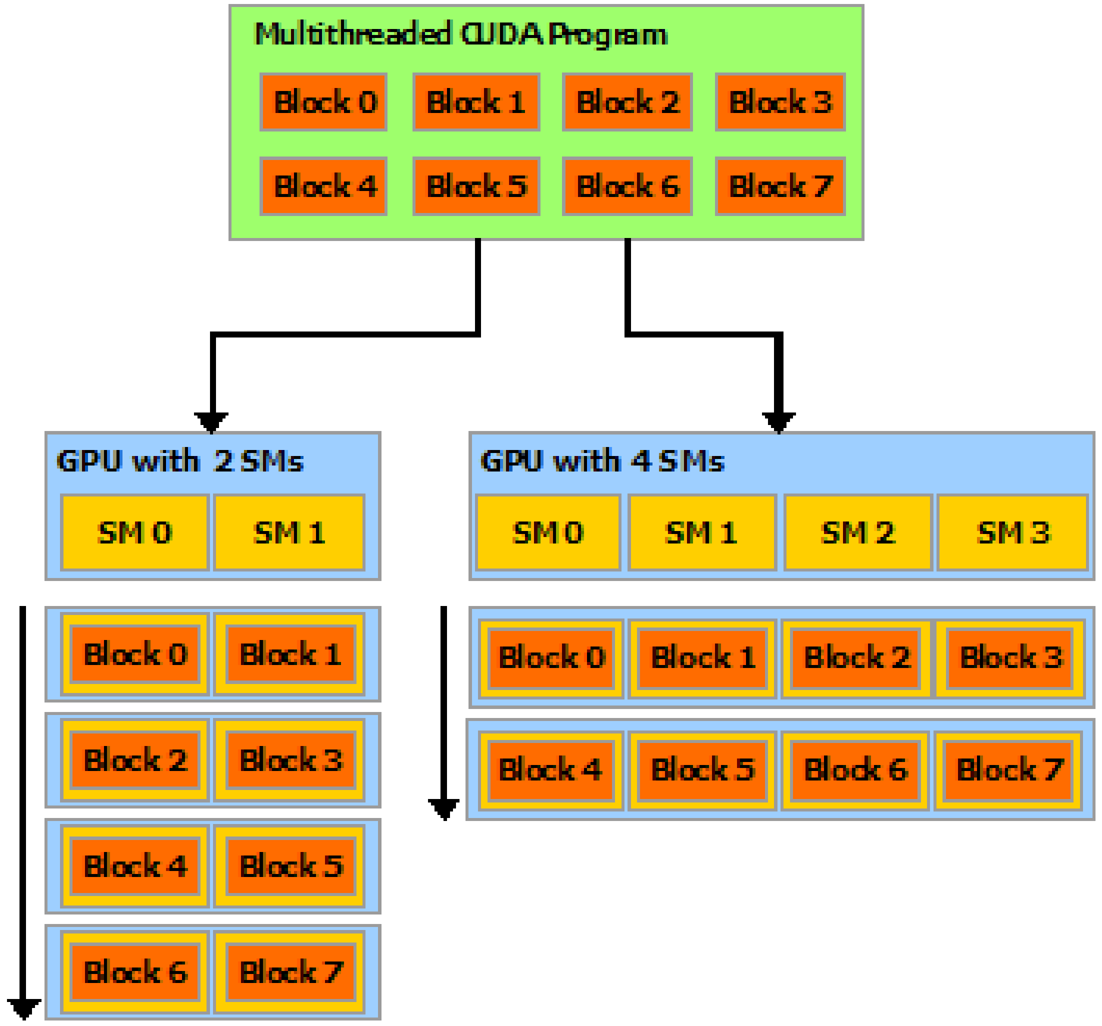
Streaming Multiprocessors



Executing Thread Blocks

- Threads assigned in blocks
  - Up to 8 blocks per SM (typical)
  - Up to 1536 total threads (typical)
- SM manages block/thread indices
- SM schedules thread execution

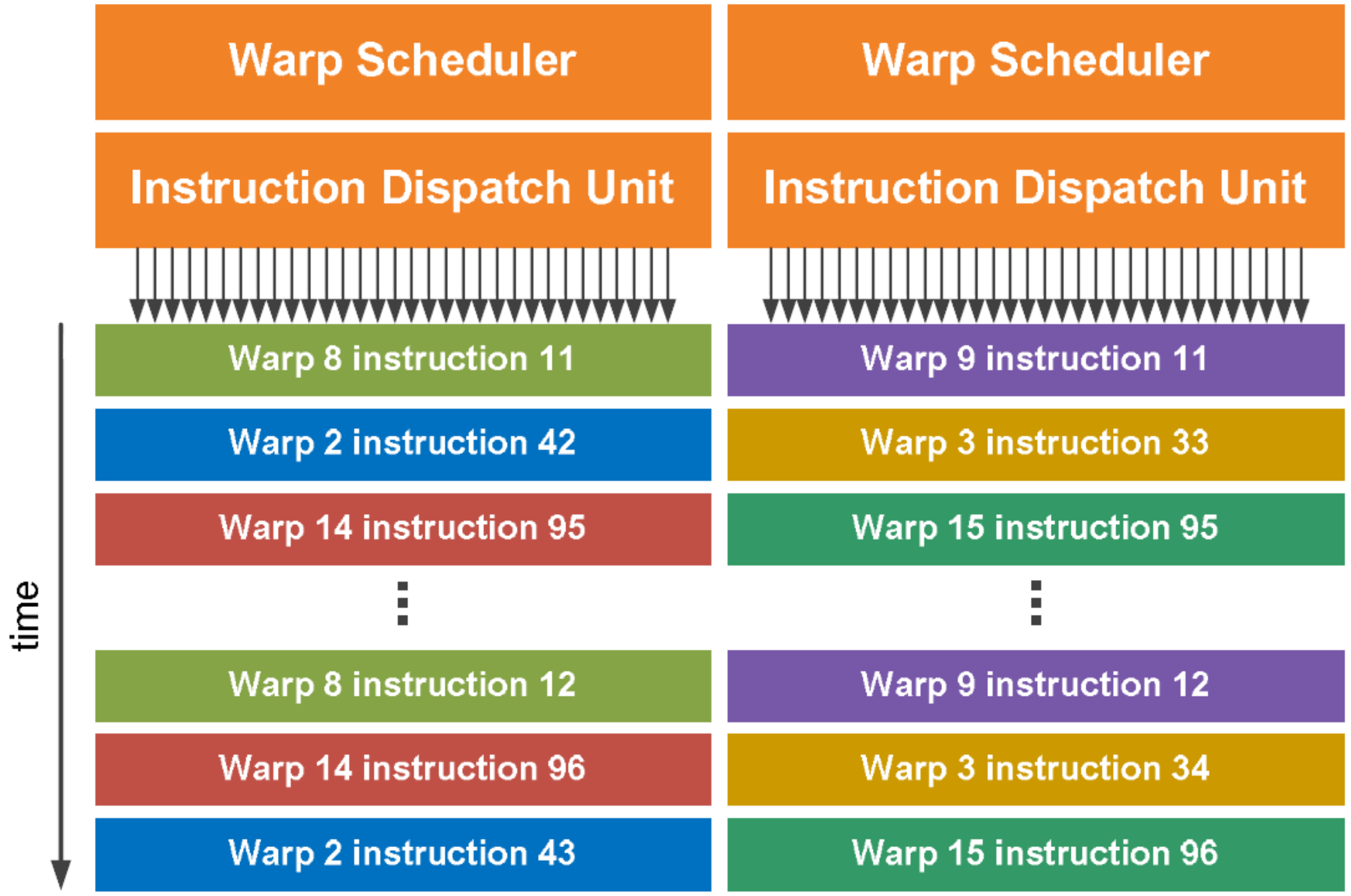
Scalability



Warps As Scheduling Unit

- 32 threads are assigned to a warp (typical)
- An implementation detail
  - Not part of the CUDA programming model
- Warps execute in Single Instruction Multiple Data mode

Warp Scheduling



Zero Overhead Warp Scheduling

- At any time, 1 of a small number of warps are executed by the SM
- Next operands ready, warp is eligible for execution
- Eligible warps selected based on priority
- All threads in a warp execute the same instruction when warp is selected

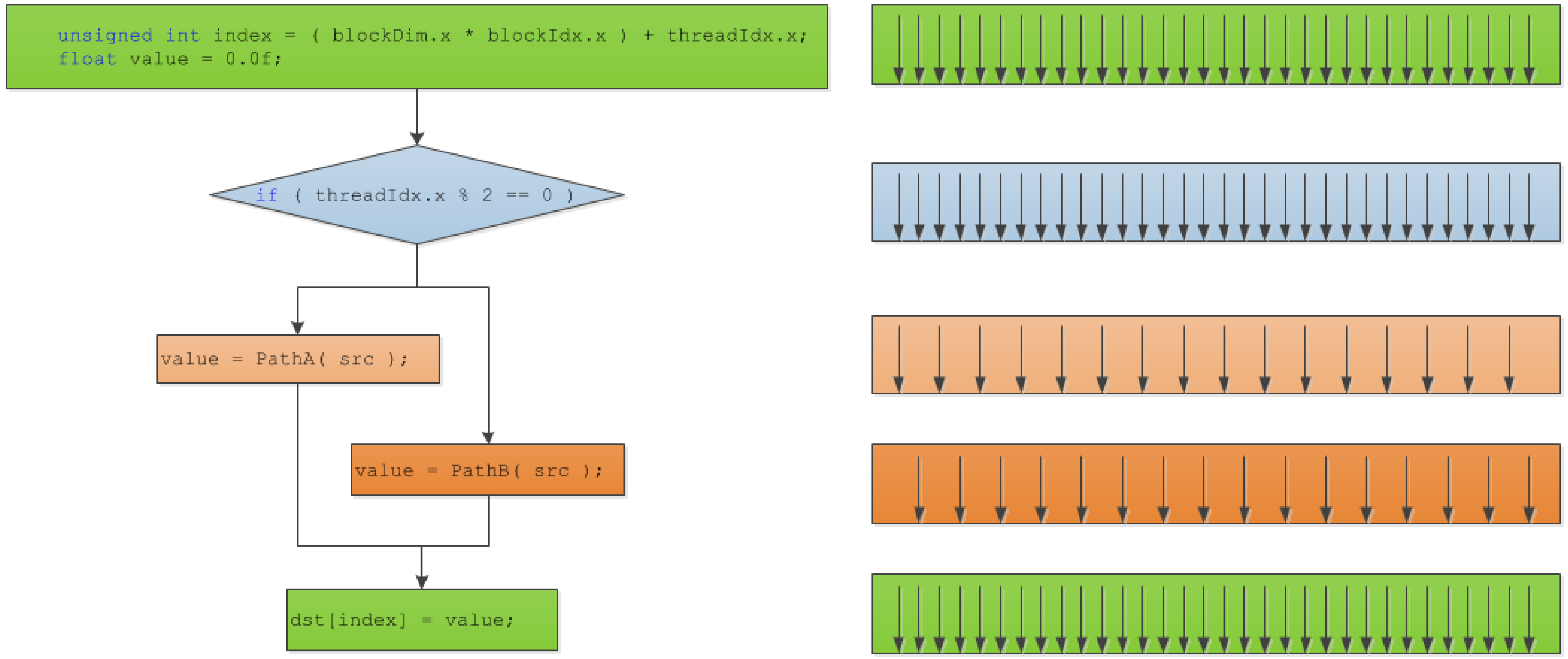
Warp Organization

- Threads are allocated sequentially into warps
  - Warp 0 starts with thread 0, etc.
- This is actually predictable and can be used to optimize code
- Warp size can change from generation to generation
- Never rely on ordering within warps or between warps

Control Flow

- Main concern with branching is divergence
- Threads within a single warp take different paths
- Different paths are serialized in current GPUs
  - Control paths taken by threads are traversed one at a time until all are completed

Control Divergence



```
unsigned int index = ( blockDim.x * blockIdx.x ) + threadIdx.x;
float value = 0.0f;
```

Divergence Examples

- With control divergence:
  - if (threadIdx.x > 2) …
  - Two different control paths in a warp
- Without control divergence:
  - if (threadIdx.x / WARP_SIZE > 2) . . .
  - Two different control paths in the block

Open CL

- Initiated by Apple
- Cross platform programming in CPUs, GPUs, FPGAs, DSPs, etc.
- Draws heavily on CUDA
  - More complex and busy due to need to maximize portability
- Many design decisions made to ease the task of vendors adapting to OpenCL

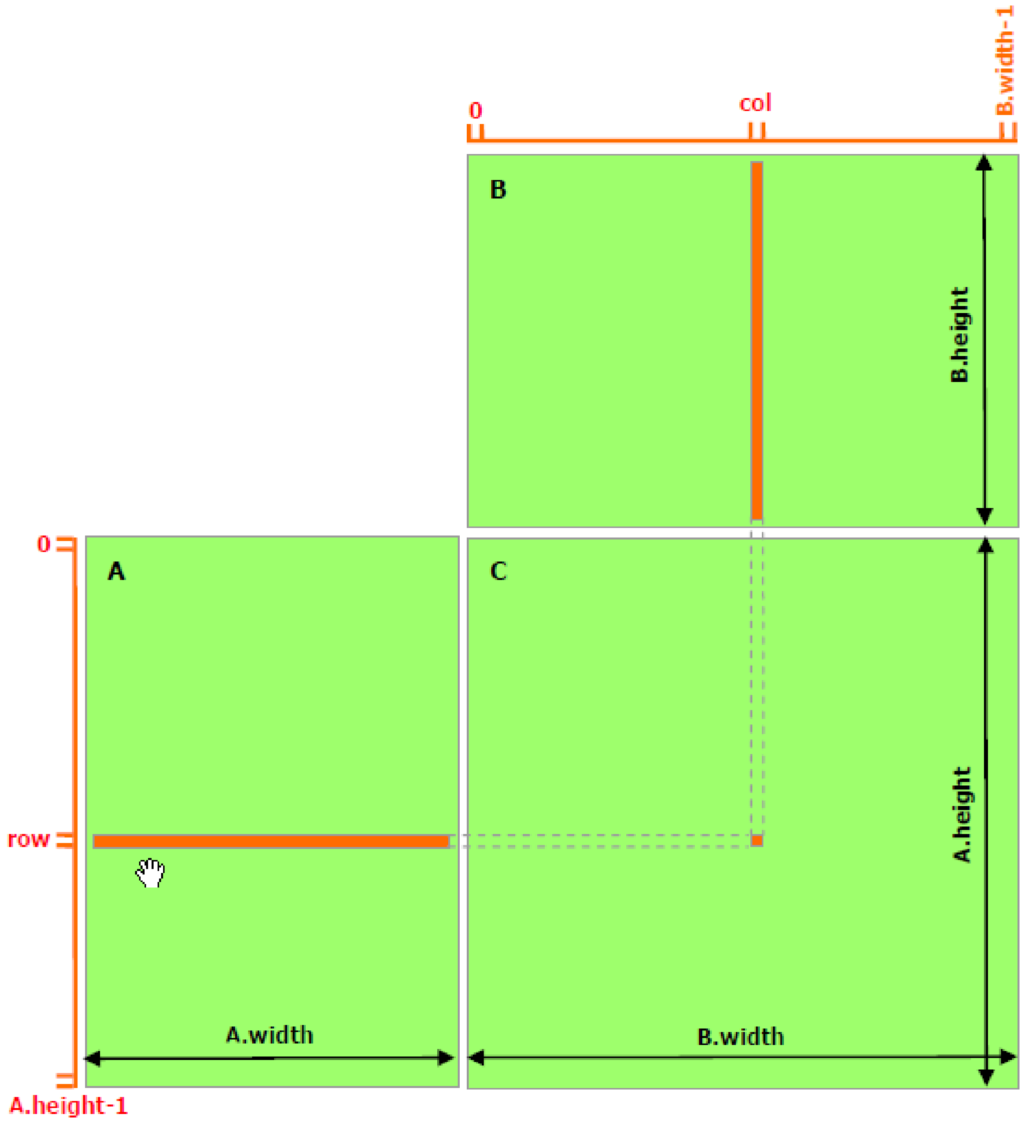
Open ACC

- Can be used to write data parallel programs in Fortran, C and C++
- Uses compiler directives (pragmas)
  - #pragma in C or C++
- Programmers can often begin with a fully sequential version then annotate with pragmas for parallelization
- Pragmas can be ignored (maybe)

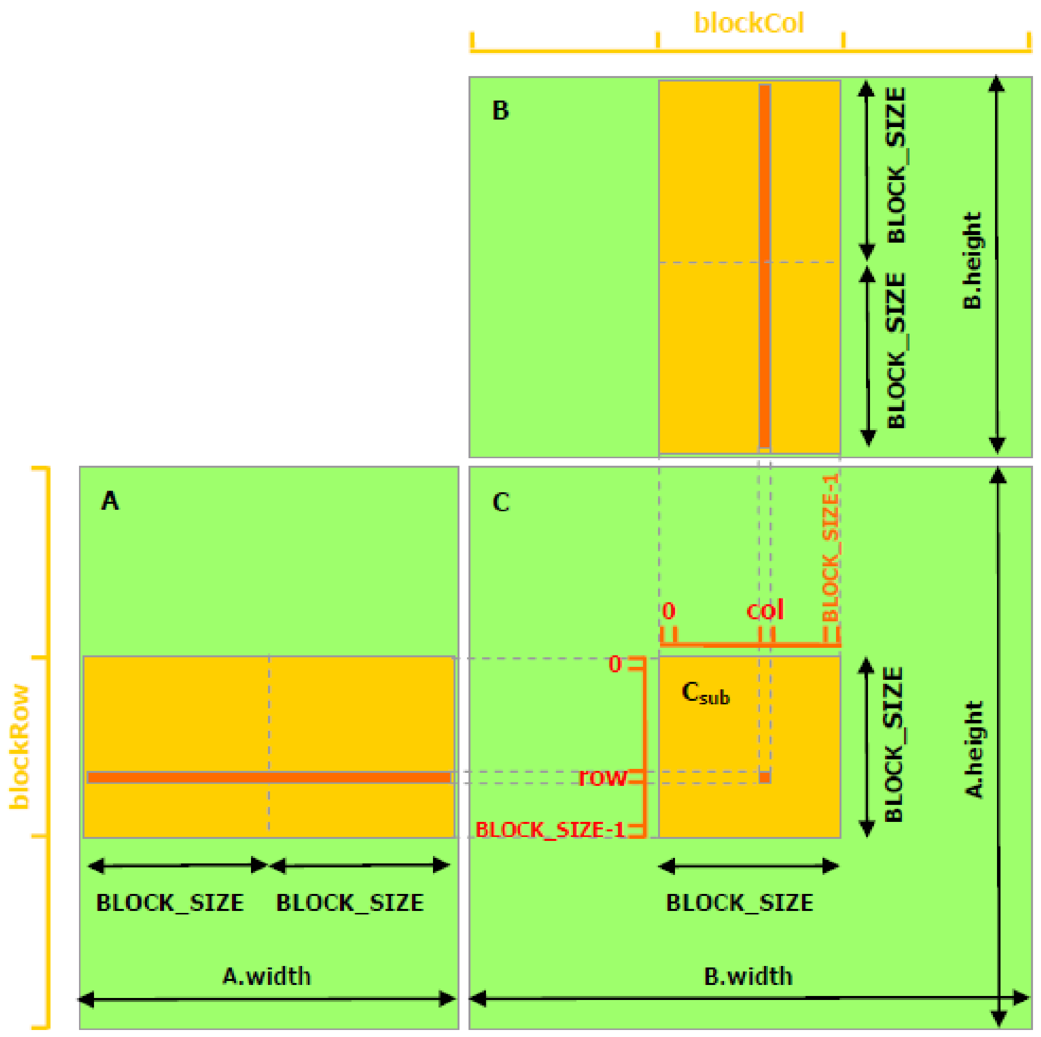
C++ AMP

- From Microsoft
- Similar to Open ACC
- Allows programmers to assert more control
- Can use multidimensional indices so that the data doesn't have to be manually linearized

Matrix Multiplication



Tiled Matrix Multiplication



blockCol
B
BLOCK_SIZE
BLOCK_SIZE
B.height
A
C
blockRow
0
col
BLOCK_SIZE-1
0
row
BLOCK_SIZE-1
Csub
BLOCK_SIZE
A.height
BLOCK_SIZE
BLOCK_SIZE
BLOCK_SIZE
A.width
B.width