EE382N (20): Computer Architecture - Parallelism and Locality

Lecture 08 – GPUs (III) (and parts of Lecture 07)

Mattan Erez

The University of Texas at Austin

GPU Teaching Kit

Accelerated Computing

Recap

Streaming model

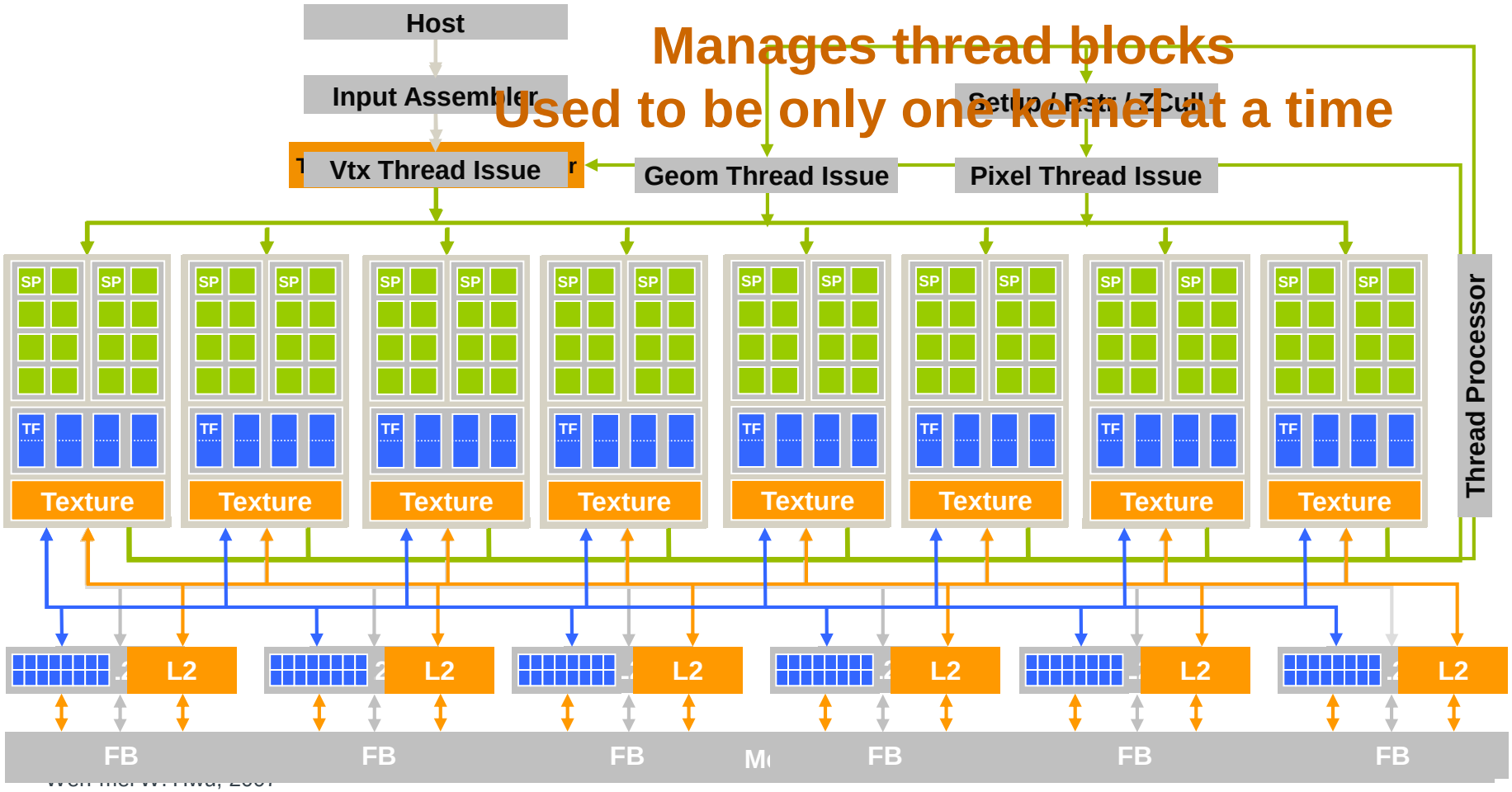
1. Use many "slimmed down cores" to run in parallel

2. Pack cores full of ALUs (by sharing instruction stream across groups of fragments)
   - Option 1: Explicit SIMD vector instructions
   - Option 2: Implicit sharing managed by hardware

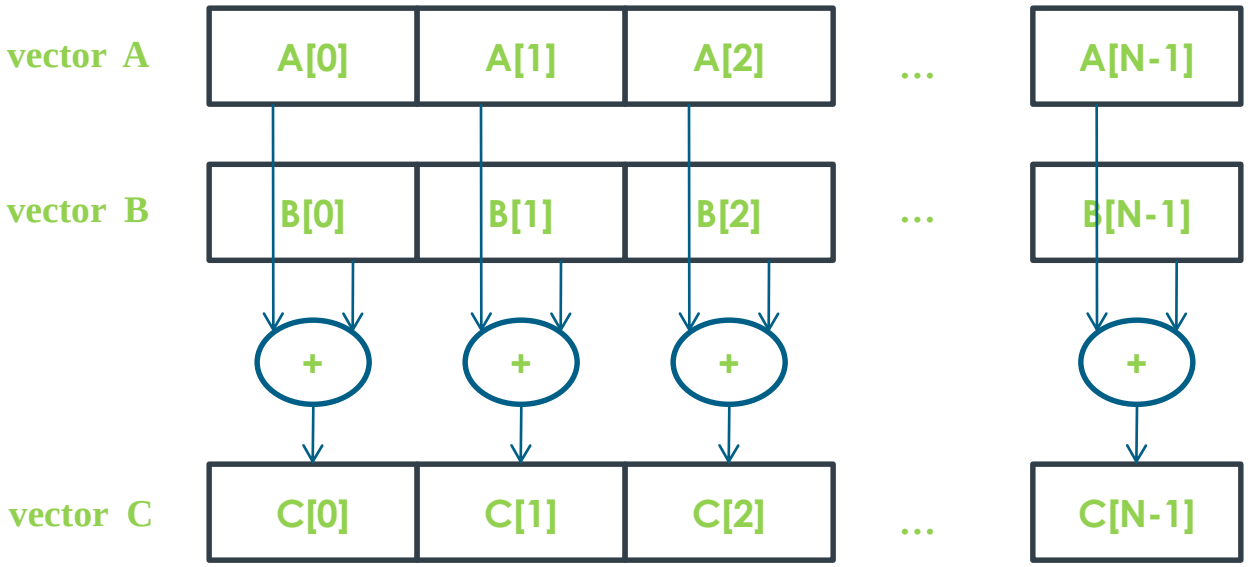
3. Avoid latency stalls by interleaving execution of many groups of fragments
   - When one group stalls, work on another group

Make the Compute Core The Focus of the Architecture

- The future of GPUs is programmable processing
- So – build the architecture around the processor



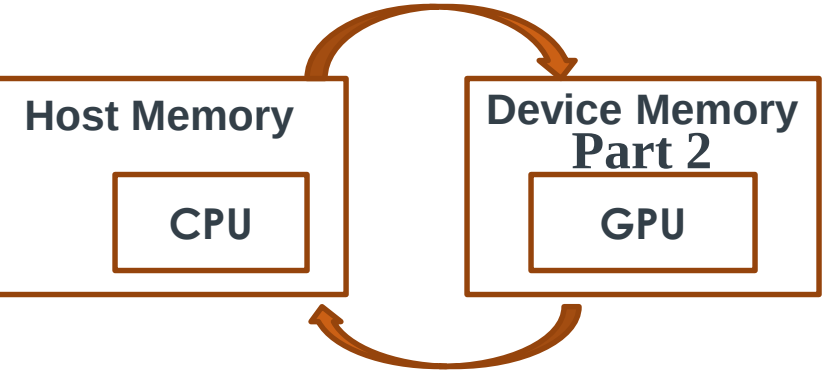
Data Parallelism - Vector Addition Example



Vector Addition – Traditional C Code

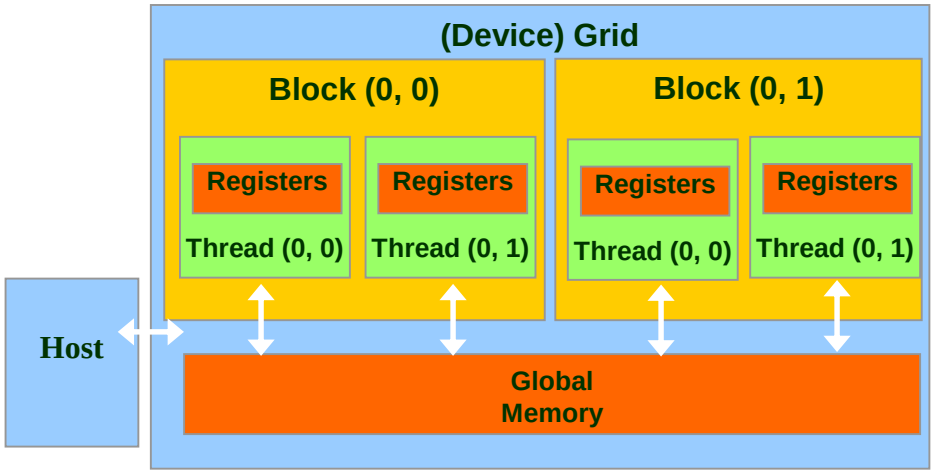
```
// Compute vector sum C = A + B
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int i;
    for (i = 0; i<n; i++) h_C[i] = h_A[i] + h_B[i];
}

int main()
{
    // Memory allocation for h_A, h_B, and h_C
    // I/O to read h_A and h_B, N elements
    …
    vecAdd(h_A, h_B, h_C, N);
}
```


Heterogeneous Computing vecAdd CUDA Host Code




```
#include <cuda.h>
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int size = n* sizeof(float);
    float *d_A, *d_B, *d_C;
    // Part 1
    // Allocate device memory for A, B, and C
    // copy A and B to device memory

    // Part 2
    // Kernel launch code – the device performs the actual vector addition

    // Part 3
    // copy C from the device memory

}
```

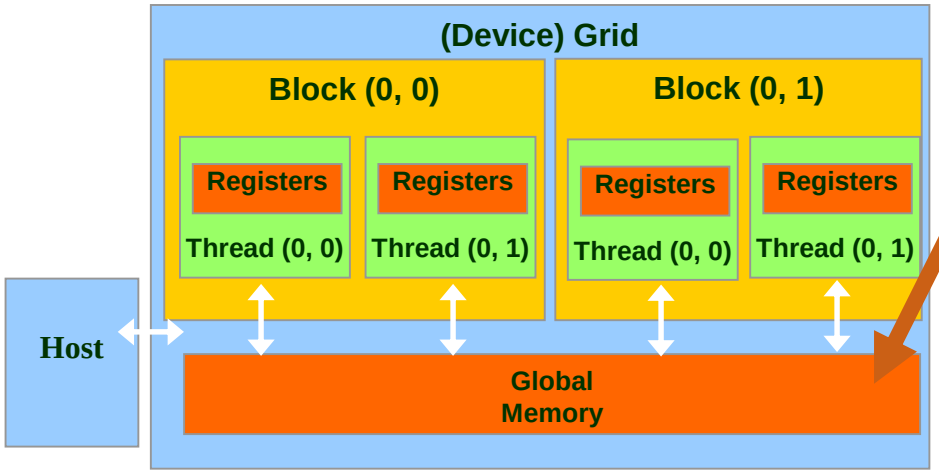

Partial Overview of CUDA Memories



- Device code can:
  - R/W per-thread registers
  - R/W all-shared global memory
- Host code can
  - Transfer data to/from per grid global memory

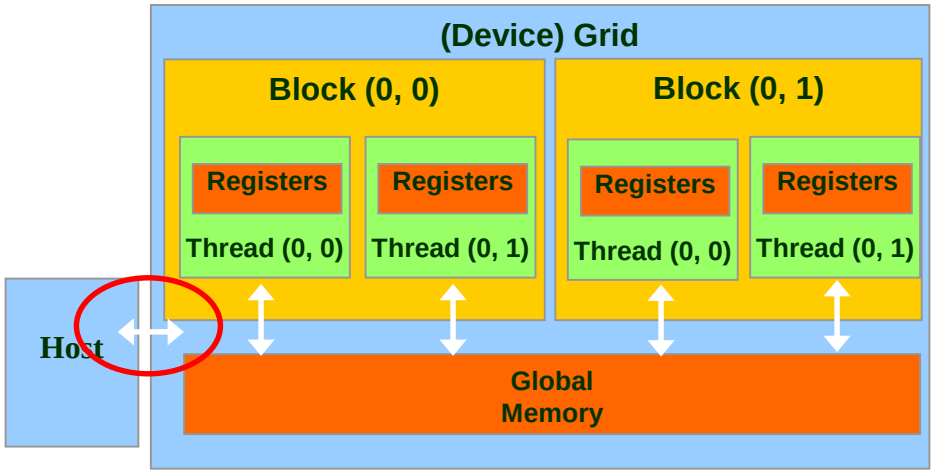
will cover more memory types and more sophisticated memory models later.

CUDA Device Memory Management API functions



- cudaMalloc()
  - Allocates an object in the device global memory
  - Two parameters
    - **Address of a pointe**r to the allocated object
    - **Size of** allocated object in terms of bytes
- cudaFree()
  - Frees object from device global memory
  - One parameter
    - **Pointer** to freed object

Host-Device Data Transfer API functions



- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type/Direction of transfer
  - Transfer to device is asynchronous

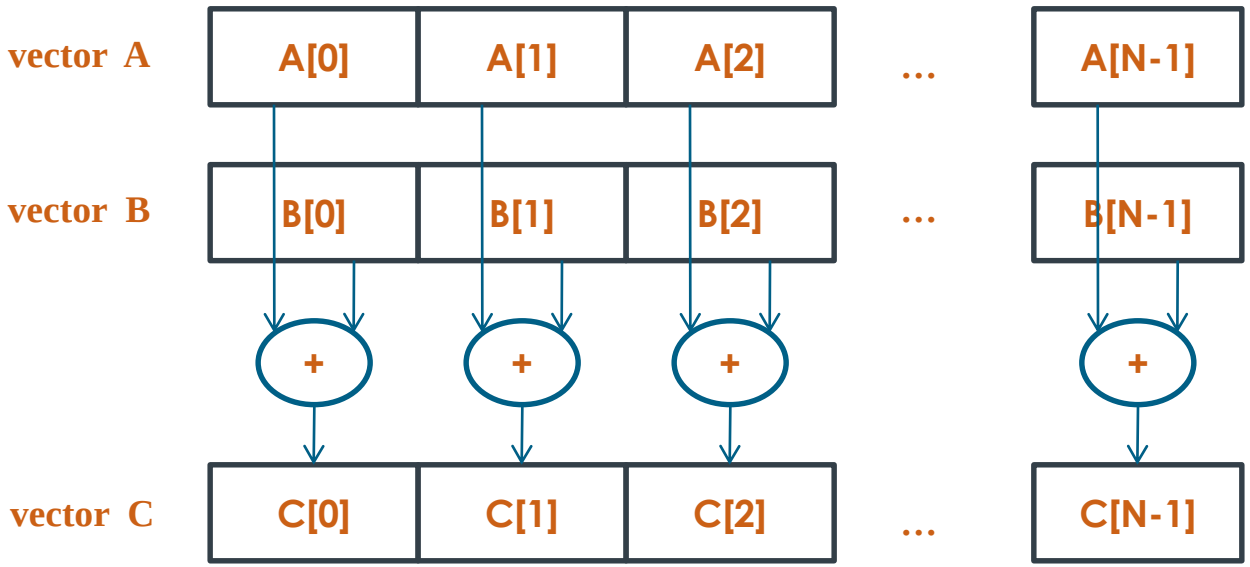
Vector Addition Host Code

```
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int size = n * sizeof(float); float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_C, size);

    // Kernel invocation code – to be shown later

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```


In Practice, Check for API Errors in Host Code

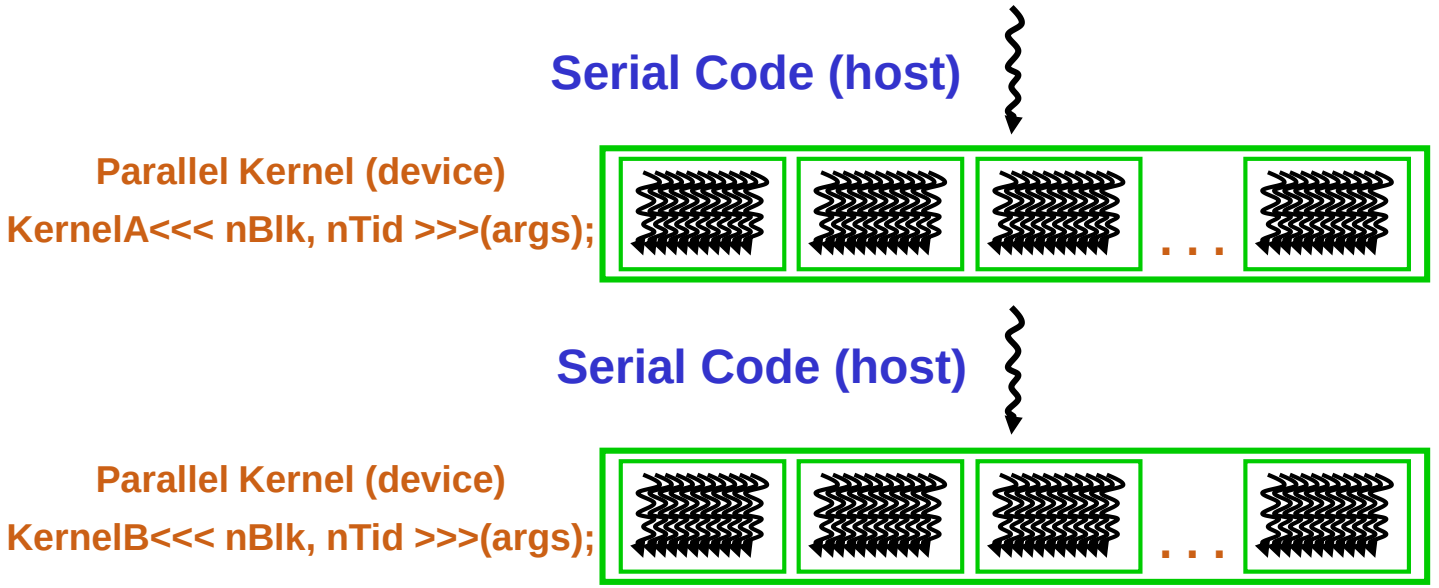
```
cudaError_t  err = cudaMalloc((void **) &d_A, size);

if (err != cudaSuccess)  {
   printf("%s in %s at line %d\n",   cudaGetErrorString(err), __FILE__,
   __LINE__);
   exit(EXIT_FAILURE);
}
```


Data Parallelism - Vector Addition Example

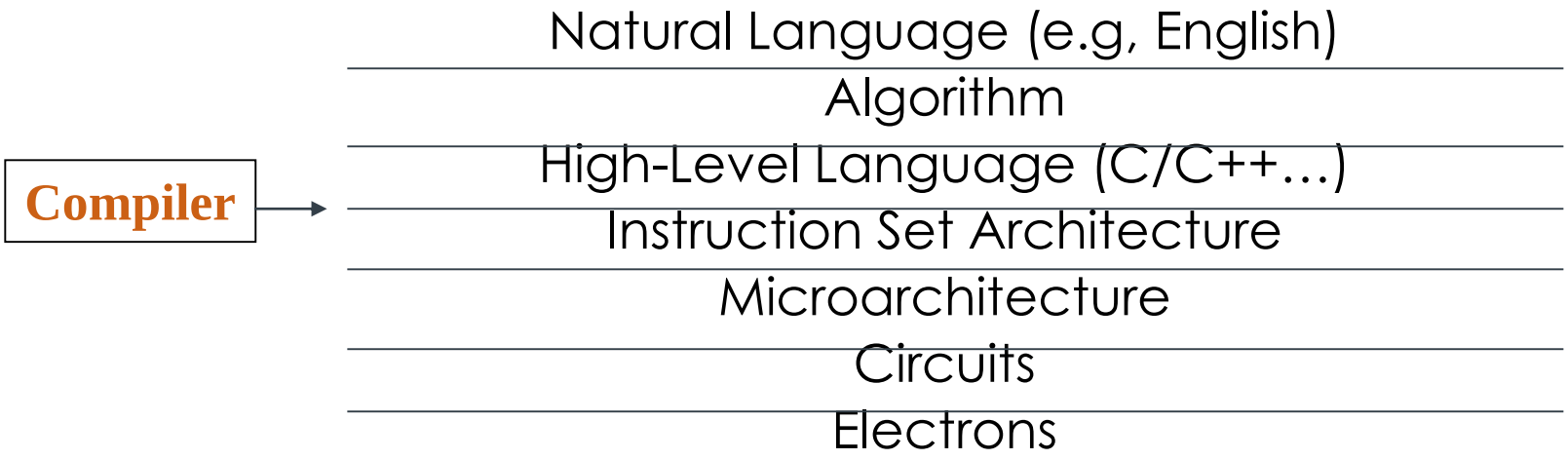


CUDA Execution Model

- Heterogeneous host (CPU) + device (GPU) application C program
  - Serial parts in **host** C code
  - Parallel parts in **device** SPMD kernel code



From Natural Language to Electrons



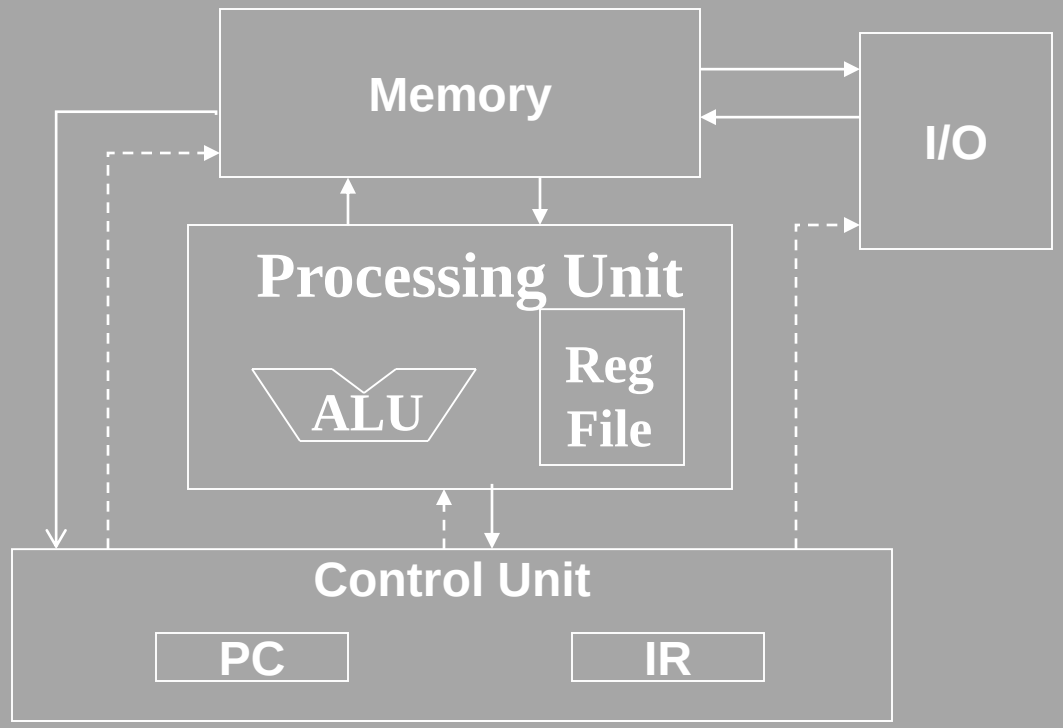
A program at the ISA level

- A program is a set of instructions stored in memory that can be read, interpreted, and executed by the hardware.
  - Both CPUs and GPUs are designed based on (different) instruction sets

- Program instructions operate on data stored in memory and/or registers.

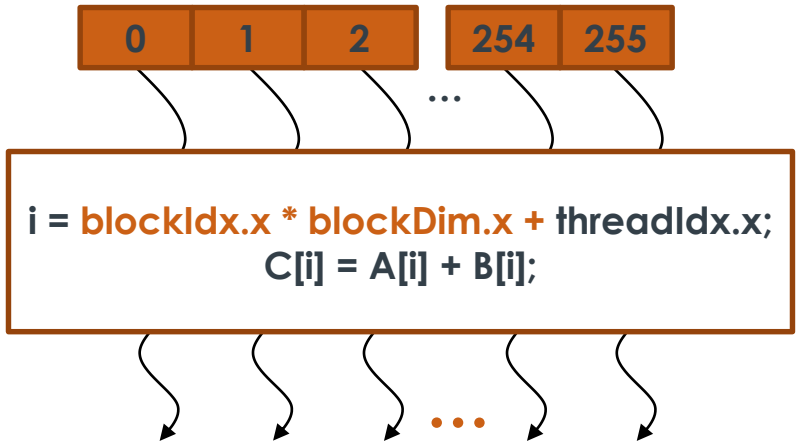
A Thread as a Von-Neumann Processor

A thread is a “virtualized” or “abstracted” Von-Neumann Processor

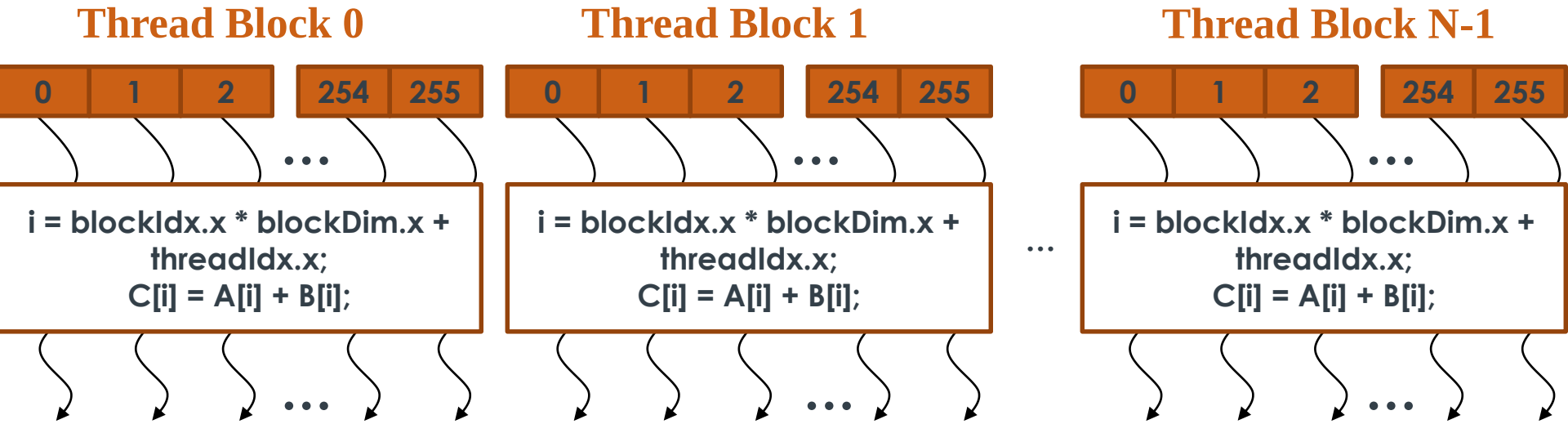


Arrays of Parallel Threads

- A CUDA kernel is executed by a grid (array) of threads
  - All threads in a grid run the same kernel code (Single Program Multiple Data)
  - Each thread has indexes that it uses to compute memory addresses and make control decisions



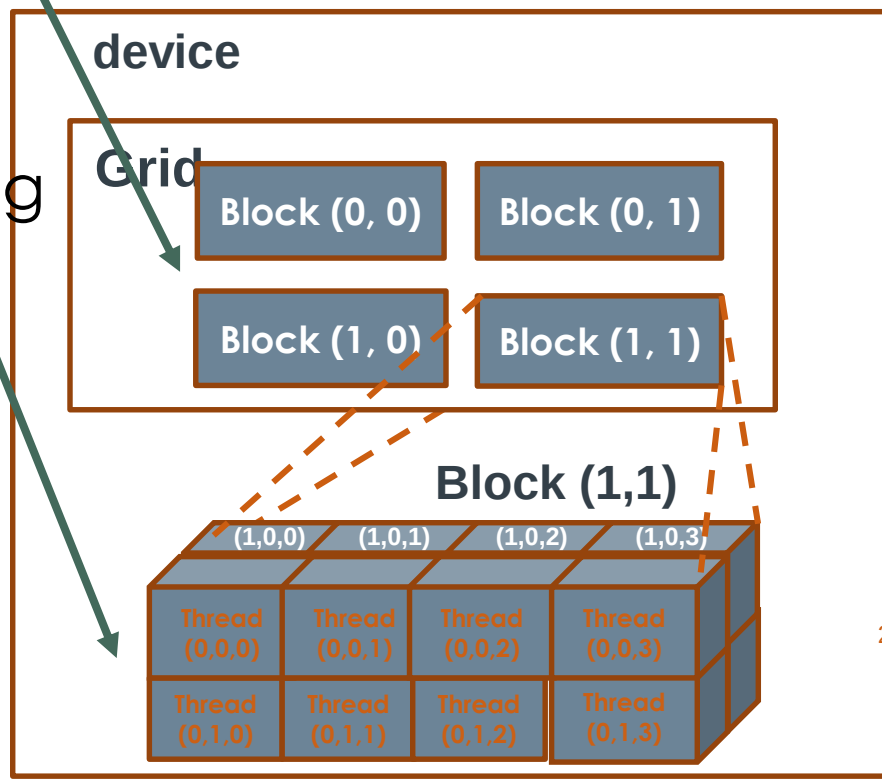
Thread Blocks: Scalable Cooperation



- Divide thread array into multiple blocks
  - Threads within a block cooperate via **shared memory**, **atomic operations** and **barrier synchronization**
  - Threads in different blocks do not interact

blockIdx and threadIdx

- Each thread uses indices to decide what data to work on
  - blockIdx: 1D, 2D, or 3D (CUDA 4.0)
  - threadIdx: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



Example: Vector Addition Kernel

Device Code

```
// Compute vector sum C = A + B
// Each thread performs one pair-wise addition

__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
    int i = threadIdx.x+blockDim.x*blockIdx.x;
    if(i<n) C[i] = A[i] + B[i];
}
```

Example: Vector Addition Kernel Launch (Host Code)

Host Code

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
 // d_A, d_B, d_C allocations and copies omitted
 // Run ceil(n/256.0) blocks of 256 threads each
  vecAddKernel<<<ceil(n/256.0),256>>>(d_A, d_B, d_C, n);
}
```

The ceiling function makes sure that there are enough threads to cover all elements.

More on Kernel Launch (Host Code)

Host Code

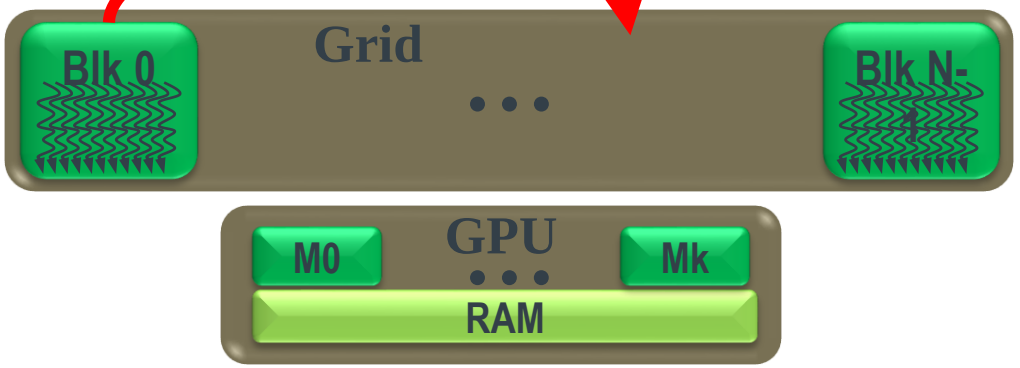
```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
  dim3 DimGrid((n-1)/256 + 1, 1, 1);
  dim3 DimBlock(256, 1, 1);
  vecAddKernel<<<DimGrid,DimBlock>>>(d_A, d_B, d_C, n);
}
```

This is an equivalent way to express the ceiling function.

Kernel execution in a nutshell

```
__host__
void vecAdd(…)
{
dim3 DimGrid(ceil(n/256.0),1,1);
dim3 DimBlock(256,1,1);
vecAddKernel<<<DimGrid,DimBlock>>>(d_A,d_B
,d_C,n);
}
```

```
__global__
void vecAddKernel(float *A,
   float *B, float *C, int n)
{
int i = blockIdx.x * blockDim.x
      + threadIdx.x;

if( i<n ) C[i] = A[i]+B[i];
}
```

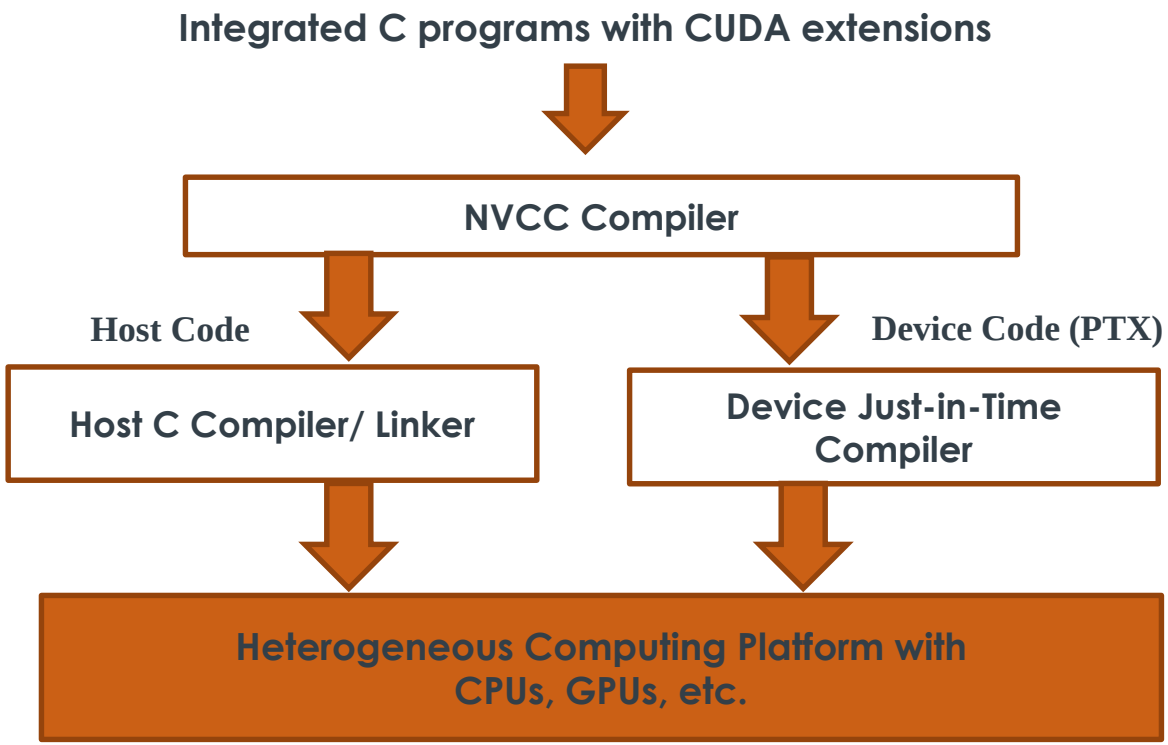


More on CUDA Function Declarations

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

- `__global__` defines a kernel function
  - Each "__" consists of two underscore characters
  - A kernel function must return `void`
- `__device__` and `__host__` can be used together
- `__host__` is optional if used alone

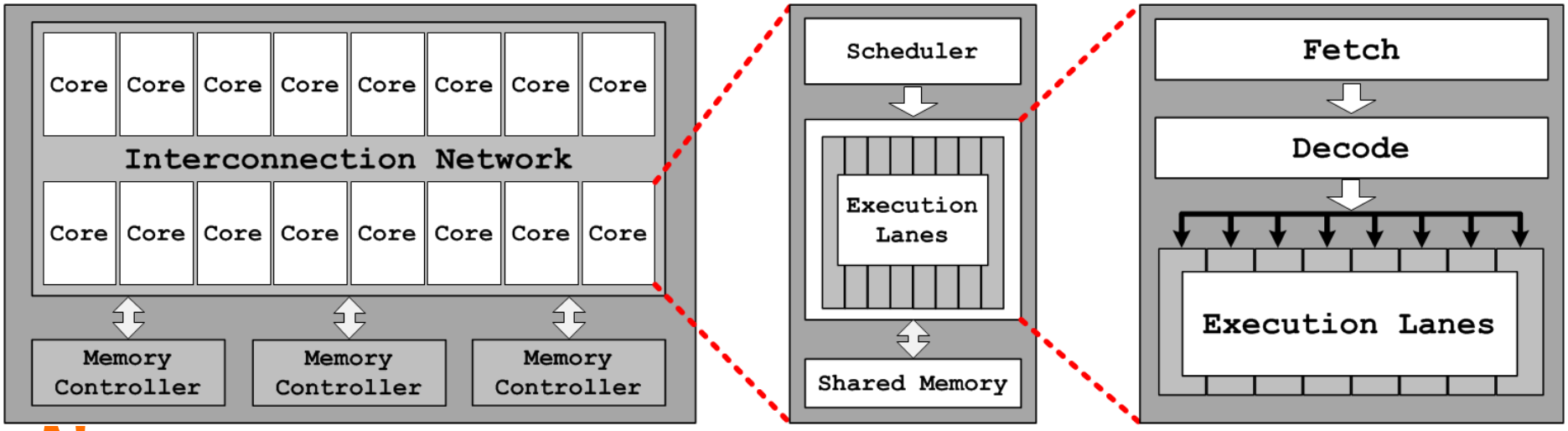
Compiling A CUDA Program



Graphic Processing Units (GPUs)

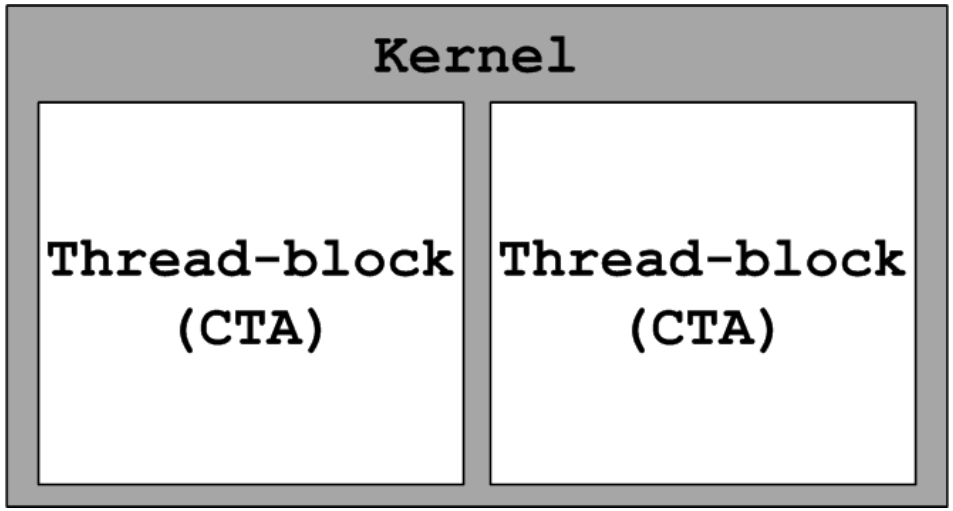
- General-purpose many-core accelerators
  - Use simple in-order shader cores in 10s / 100s numbers
  - Shader core == **S**treaming **M**ultiprocessor (**SM**) in NVIDIA GPUs

- Scalar frontend (fetch & decode) + parallel backend
  - Amortizes the cost of frontend and control



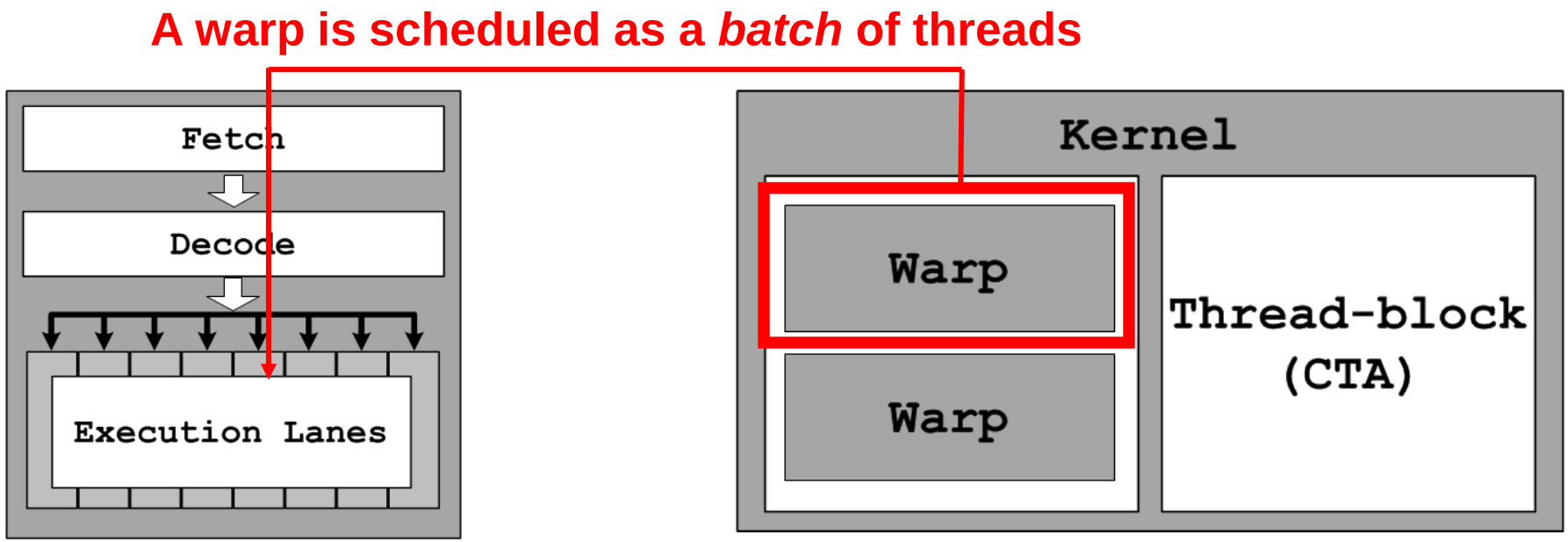
CUDA exposes hierarchy of data-parallel threads

- **SPMD model**: single ***kernel*** executed by all *scalar* threads
- **Kernel / Thread-block**
  - Multiple **thread-blocks** (cooperative-thread-arrays (**CTA**s)) compose a **kernel**



CUDA exposes hierarchy of data-parallel threads

- **SPMD model**: single ***kernel*** executed by all *scalar* threads
- **Kernel / Thread-block / *Warp* / *Thread***
  - Multiple **warps** compose a **thread-block**
  - Multiple **threads (32)** compose a warp



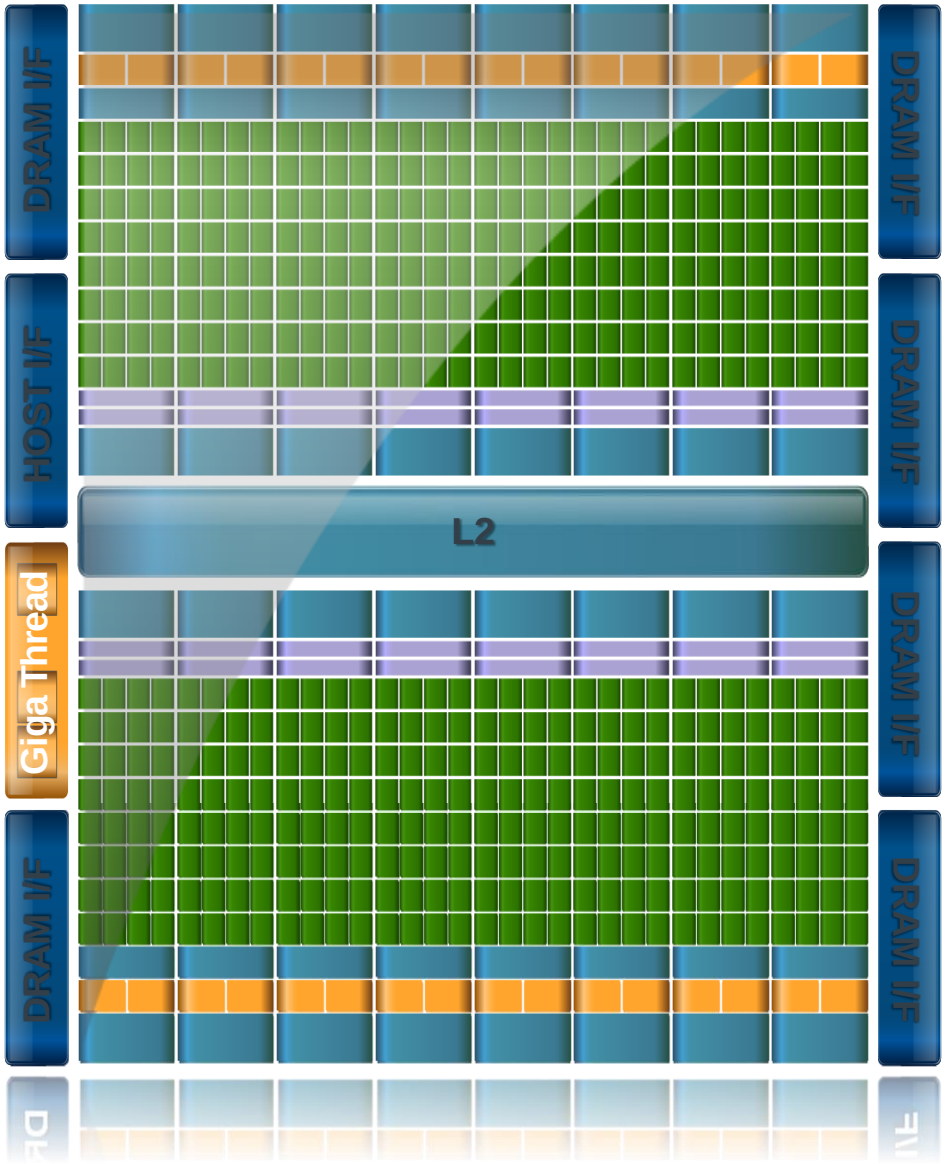
SIMT for balanced *programmability* and *HW-eff.*

- **SIMT**: Single-Instruction Multiple-**Thread**
  - Programmer writes code to be executed by ***scalar*** threads
  - Underlying hardware uses ***vector*** SIMD pipelines (lanes)
    - HW/SW groups ***scalar*** threads to execute in ***vector*** lanes

- Enhanced programmability using SIMT
  - Hardware/software supports ***conditional branches***
    - Each thread can follow its *own control flow*
  - ***Per-thread load/store*** instructions are also supported
    - Each thread can reference *arbitrary address regions*

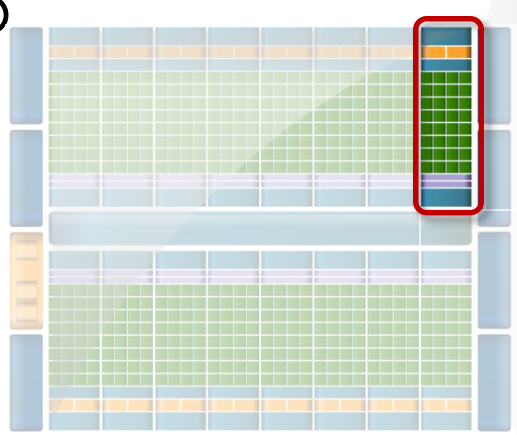
Older GPU, but high-level same

- Expand performance sweet spot of the GPU
  - Caching
  - Concurrent kernels
  - FP64
  - 512 cores
  - GDDR5 memory
- Bring more users, more applications to the GPU
  - C++
  - Visual Studio Integration
  - ECC



Streaming Multiprocessor (SM)

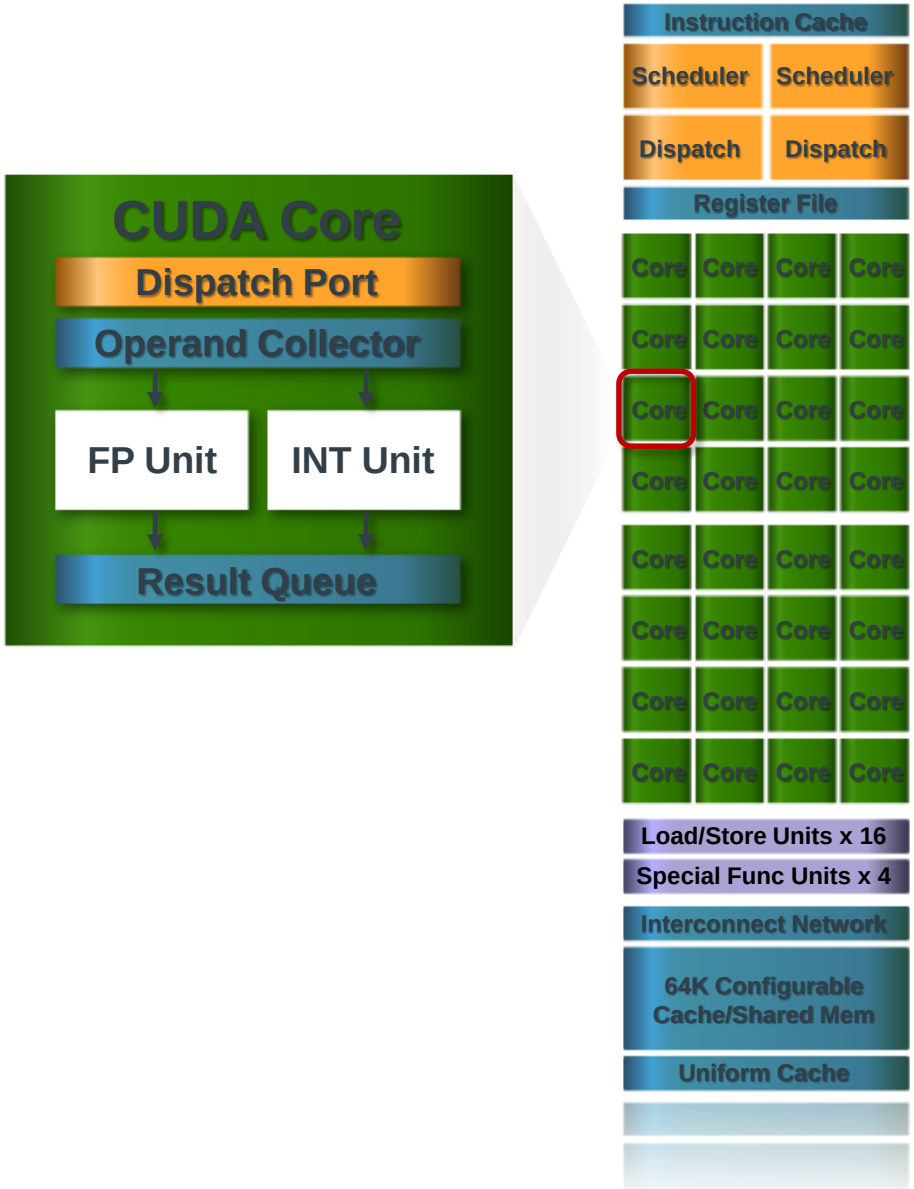
- Objective – optimize for GPU computing
  - New ISA
  - Revamp issue / control flow
  - New CUDA core architecture
- 16 SMs per Fermi chip
- 32 cores per SM (512 total)
- 64KB of configurable L1$ / shared memory



	FP32	FP64	INT	SFU	LD/ST
Ops / clk	32	16	32	4	16

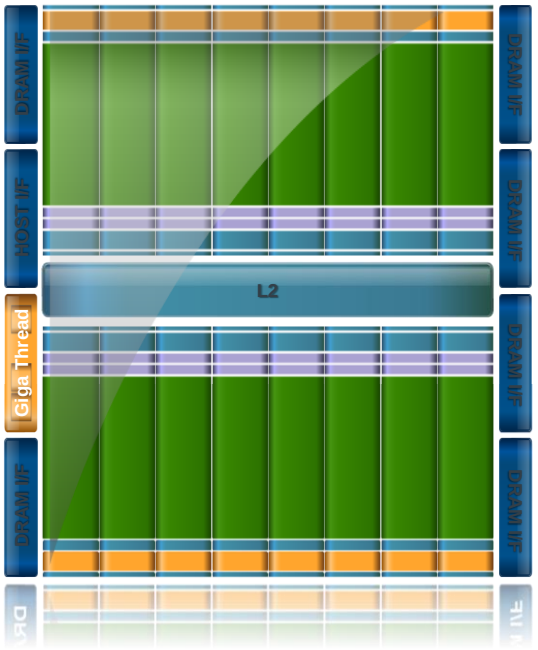
SM Microarchitecture

- New IEEE 754-2008 arithmetic standard
- Fused Multiply-Add (FMA) for SP & DP
- New integer ALU optimized for 64-bit and extended precision ops



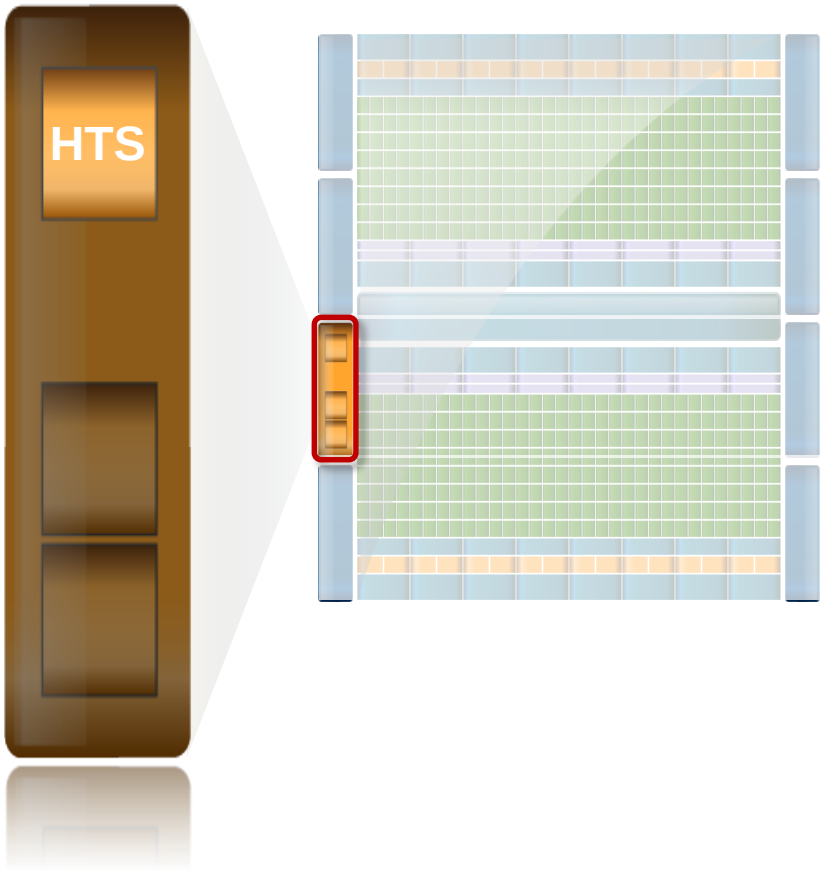
Memory Hierarchy

- True cache hierarchy + on-chip shared RAM
  - On-chip shared memory: good fit for regular memory access
    - dense linear algebra, image processing, …
  - Caches: good fit for irregular or unpredictable memory access
    - ray tracing, sparse matrix multiply, physics …
- Separate L1 Cache for each SM (16/48 KB)
  - Improves bandwidth and reduces latency
- Unified L2 Cache for all SMs (768 KB)
  - Fast, coherent data sharing across all cores in the GPU



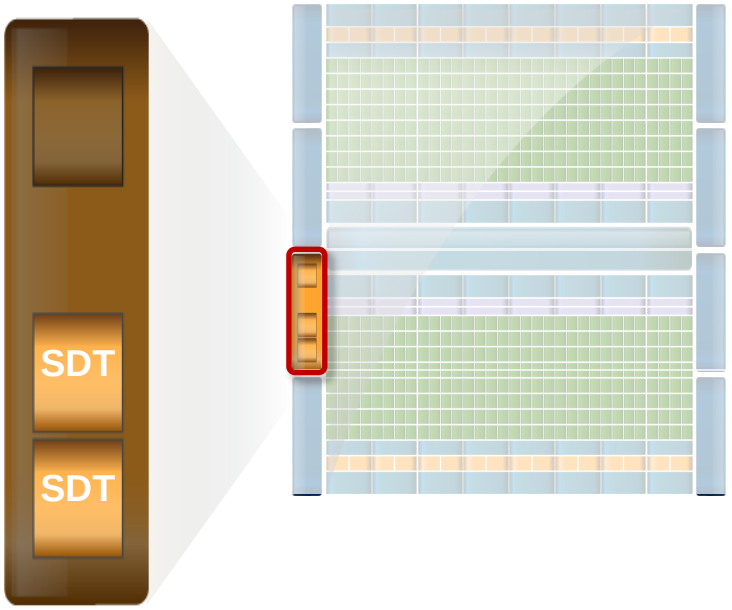
GigaThread™ Hardware Thread Scheduler

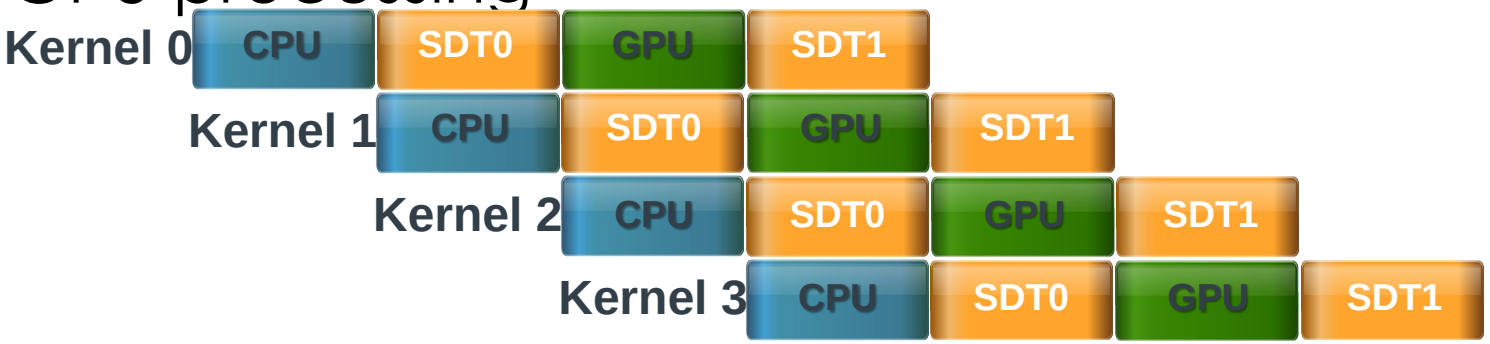
- Hierarchically manages tens of thousands of simultaneously active threads
- 10x faster context switching on Fermi
- Overlapping kernel execution



GigaThread Streaming Data Transfer Engine

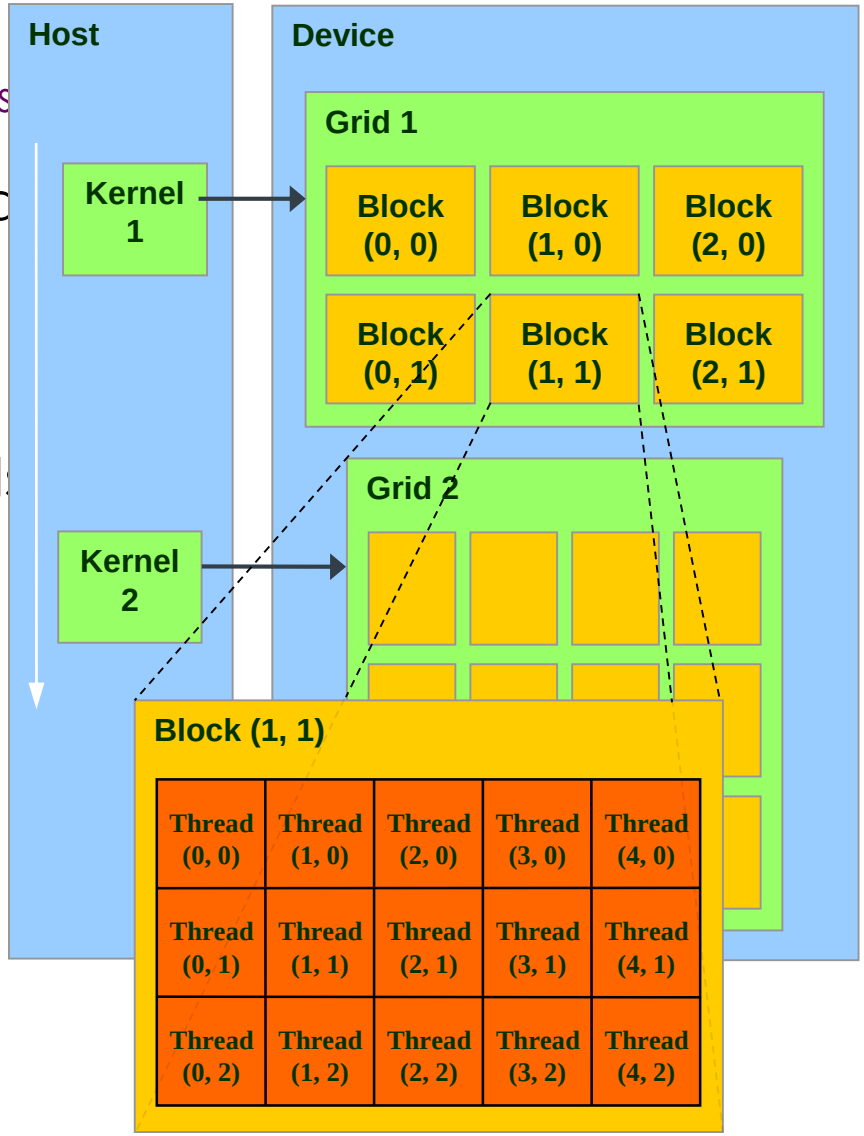
- Dual DMA engines
- Simultaneous CPU→GPU and GPU→CPU data transfer
- Fully overlapped with CPU/GPU processing



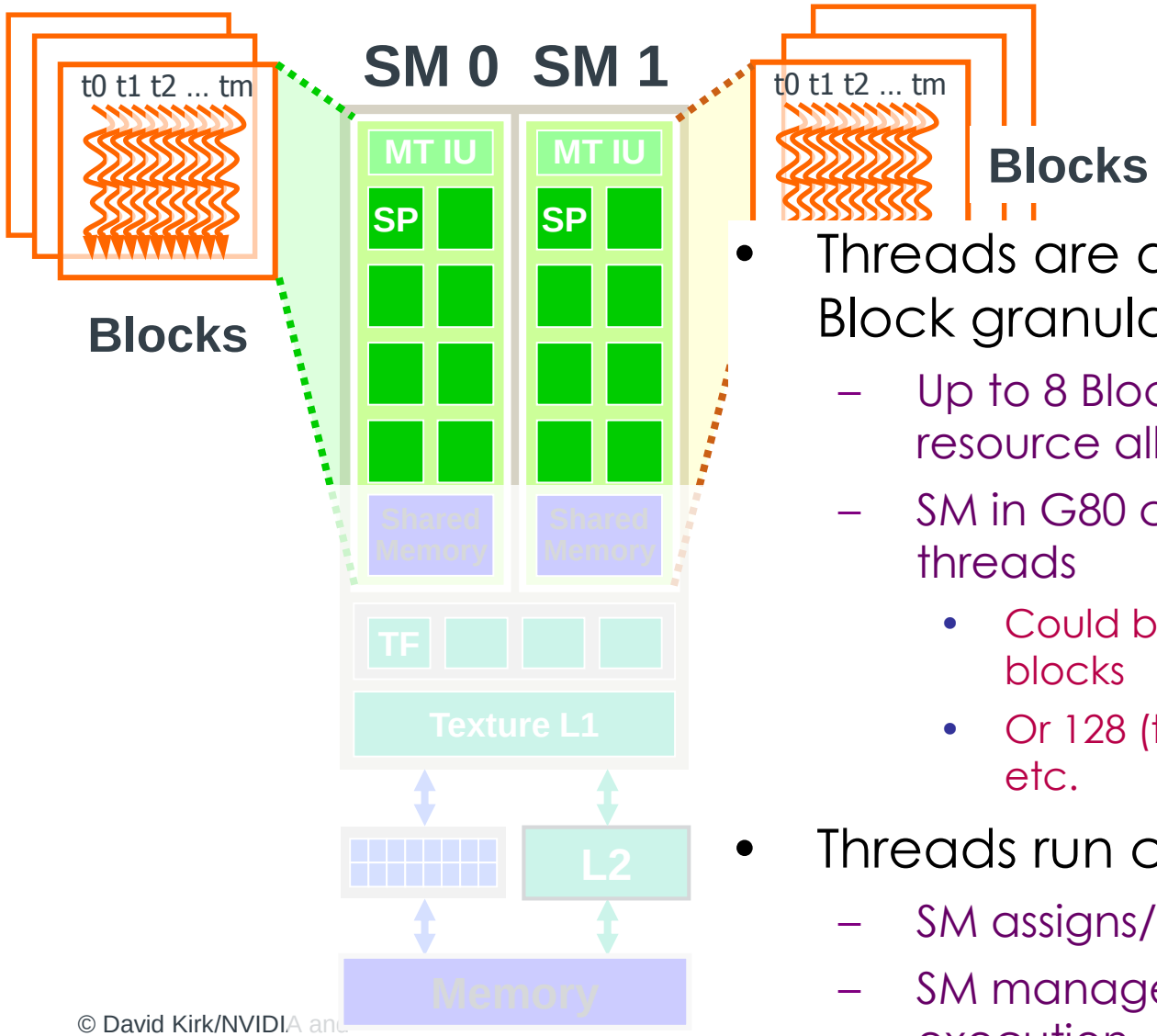


Thread Life Cycle in HW

- Kernel is launched on the SPA
  - Kernels known as ***grids*** of thread blocks
- Thread Blocks are serially distributed to all the SM's
  - Potentially >1 Thread Block per SM
  - At least 96 threads per block
- Each SM launches Warps of Threads
  - 2 levels of parallelism
- SM schedules and executes Warps that are ready to run
- As Warps and Thread Blocks complete, resources are freed
  - SPA can distribute more Thread Blocks

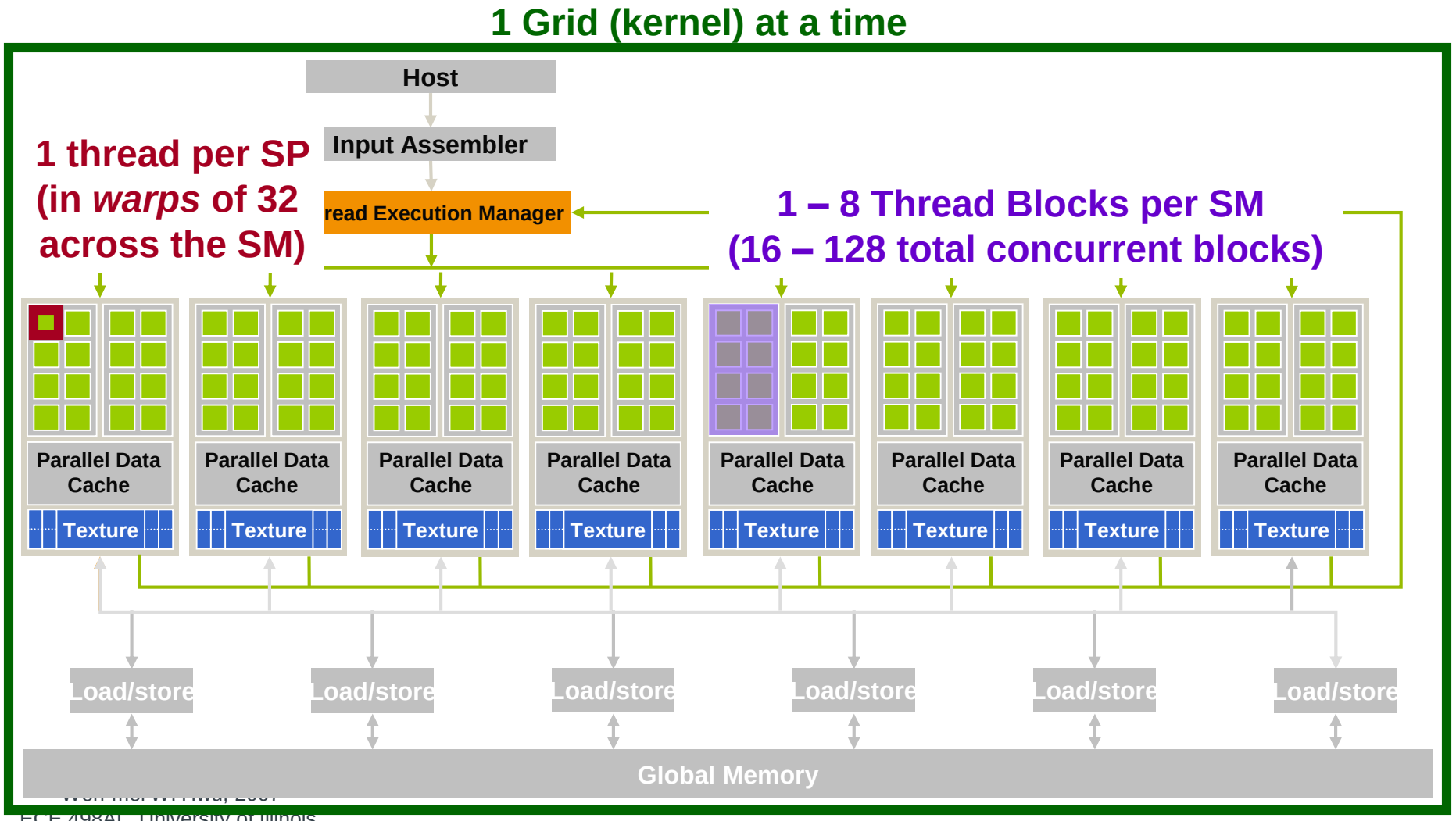


SM Executes Blocks



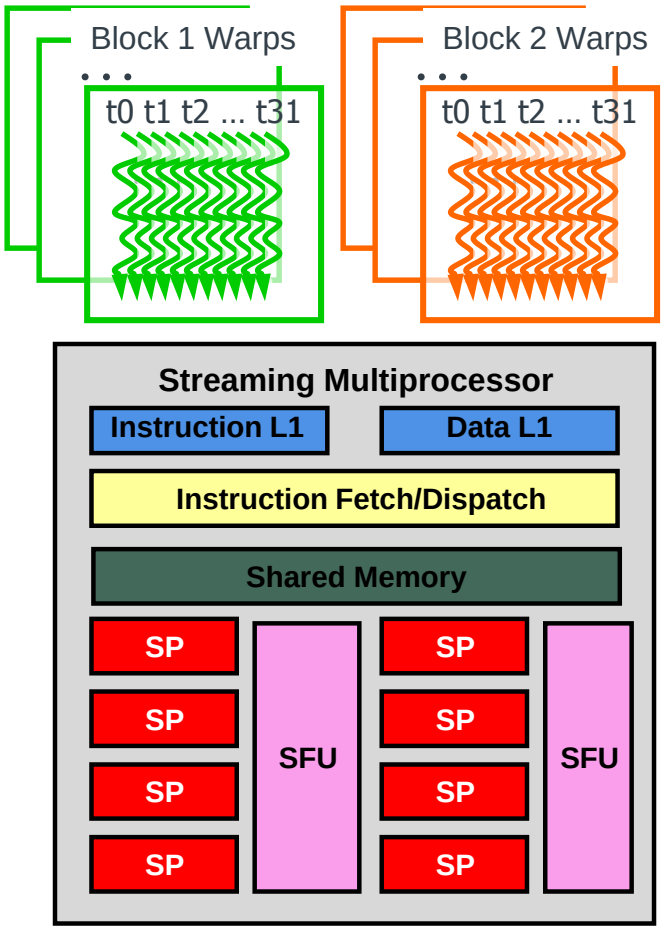
- Threads are assigned to SMs in Block granularity
  - Up to 8 Blocks to each SM as resource allows
  - SM in G80 can take up to 768 threads
    - Could be 256 (threads/block) * 3 blocks
    - Or 128 (threads/block) * 6 blocks, etc.
- Threads run concurrently
  - SM assigns/maintains thread IDs
  - SM manages/schedules thread execution

Make the Compute Core The Focus of the Architecture



Thread Scheduling/Execution

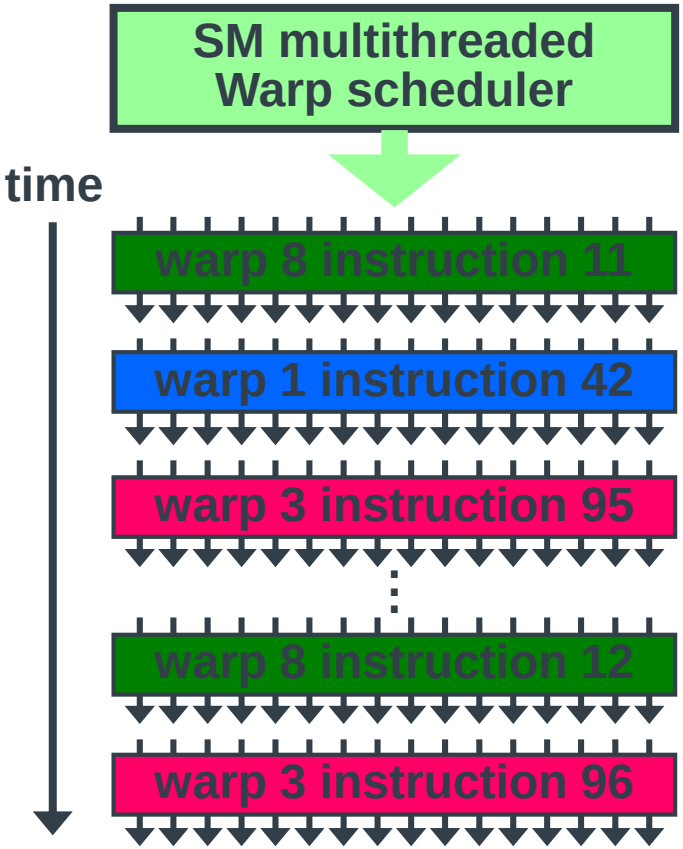
- Each Thread Block is divided into 32-thread Warps
  - This is an implementation decision
- Warps are scheduling units in SM
- If 3 blocks are assigned to an SM and each Block has 256 threads, how many Warps are there in an SM?
  - Each Block is divided into 256/32 = 8 Warps
  - There are 8 * 3 = 24 Warps
  - At any point in time, only one of the 24 Warps will be selected for instruction fetch and execution.



SM Warp Scheduling

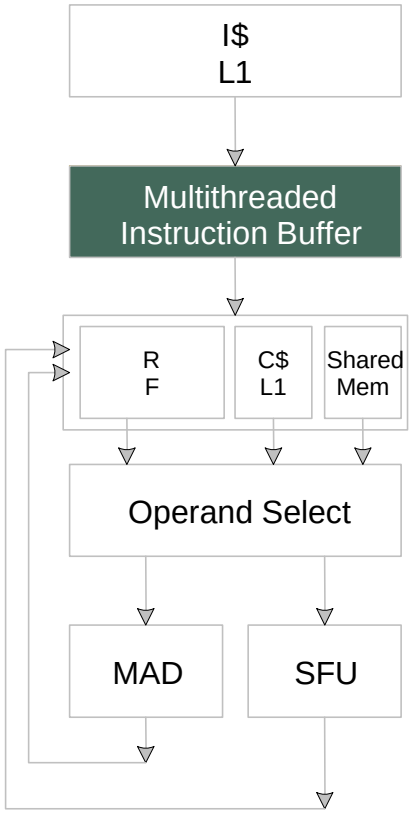
- SM hardware implements zero-overhead Warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - All threads in a Warp execute the same instruction when selected
  - Scoreboard scheduler

- 4 clock cycles needed to dispatch the same instruction for all threads in a Warp in G80
  - If one global memory access is needed for every 4 instructions
  - A minimal of 13 Warps are needed to fully tolerate 200-cycle memory latency



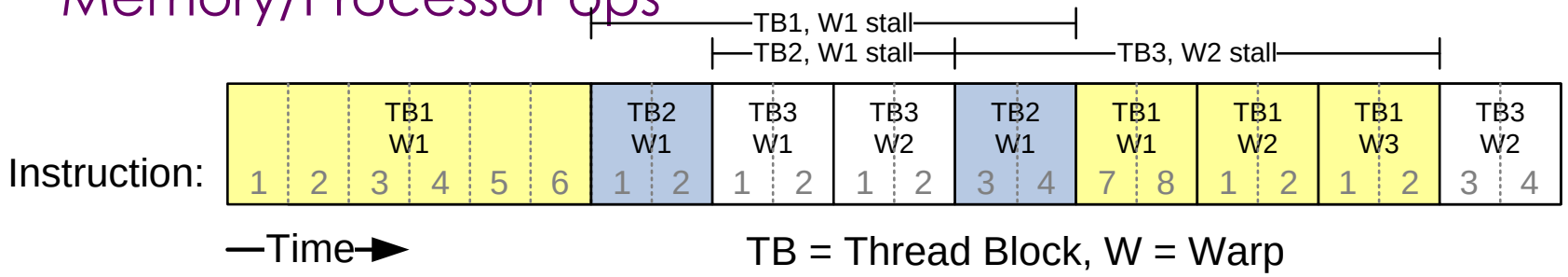
SM Instruction Buffer – Warp Scheduling

- Fetch one warp instruction/cycle
  - from instruction L1 cache
  - into any instruction buffer slot
- Issue one "ready-to-go" warp instruction/cycle
  - from any warp - instruction buffer slot
  - operand scoreboarding used to prevent hazards
- Issue selection based on round-robin/age of warp
- SM broadcasts the same instruction to 32 Threads of a Warp



Scoreboarding

- All register operands of all instructions in the Instruction Buffer are scoreboarded
  - Status becomes ready after the needed values are deposited
  - prevents hazards
  - cleared instructions are eligible for issue
- Decoupled Memory/Processor pipelines
  - any thread can continue to issue instructions until scoreboarding prevents issue
  - allows Memory/Processor ops to proceed in shadow of Memory/Processor ops



Granularity and Resource Considerations

- For Matrix Multiplication, should I use 8X8, 16X16 or 32X32 tiles (1 thread per tile element)?
  - For 8X8, we have 64 threads per Block. Since each SM can take up to 768 threads, it can take up to 12 Blocks. However, each SM can only take up to 8 Blocks, only 512 threads will go into each SM!
  - For 16X16, we have 256 threads per Block. Since each SM can take up to 768 threads, it can take up to 3 Blocks and achieve full capacity unless other resource considerations overrule.
  - For 32X32, we have 1024 threads per Block. Not even one ıto an SM!