# Advanced Parallel Architecture

Annalisa Massini - 2016/2017

# GPU - Graphics Processing Units

**Computer Architecture - A Quantitative Approach, Fifth Edition**

**Hennessy Patterson**

Chapter 4 - **Section 4.4 – Graphics Processing Units**

# Graphics Processing Units

- GPUs and CPUs do **not** go back in computer architecture genealogy to a **common ancestor**
- The primary ancestors of GPUs are graphics accelerators

- Given the hardware invested to do graphics well architects ask

***how can be the design of GPUs used to improve the performance of a wider range of applications?***

# Graphics Processing Units

- The **challenge** for the GPU programmer
  - is **not** simply **getting good performance** on the GPU
  - but also in **coordinating the scheduling** of **computation** on the system processor and the GPU and the **transfer of data** between system memory and GPU memory

- GPUs have virtually **every type of parallelism** that can be captured by the programming environment:
  - multithreading
  - MIMD
  - SIMD
  - instruction-level

# Programming the GPU

- NVIDIA developed a ***C-like language and programming environment*: CUDA - Compute Unified Device Architecture**
- CUDA produces C/C++ for the system processor - ***host*** - and a C and C++ dialect for the GPU - ***device*** (D in CUDA)
- A similar programming language is **OpenCL**, which several companies are developing as vendor-independent language for multiple platforms

# Programming the GPU

- NVIDIA unified all forms of parallelism in the ***CUDA Thread and*** classifies the CUDA programming model as Single Instruction, Multiple Thread (***SIMT***)

- The compiler and the hardware can gang thousands of CUDA Threads together to utilize the various styles of parallelism within a GPU (multithreading, MIMD, SIMD, ILP)
- Threads are blocked together - ***Thread Block*** - and executed in **groups of 32** threads
- In Hennessy-Patterson the hardware that executes a whole **block of threads** is called a ***multithreaded SIMD Processor***

# Programming the GPU

- To distinguish between functions for the GPU (device) and functions for the system processor (host), CUDA uses
  - `__device__` or `__global__` for the device
  - `__host__` for the processor

- CUDA variables declared as in the
  - `__device__` or `__global__` functions

  are allocated to the GPU Memory which is accessible by all multithreaded SIMD processors

# Programming the GPU

- The call syntax for the function *name* that runs on the GPU

  ```
  name<<<dimGrid, dimBlock>>>(... parameter list ...)
  ```

  where `dimGrid` and `dimBlock` specify the dimensions of the code (in blocks) and the dimensions of a block (in threads)

- CUDA provides keywords for:
  - the identifier for blocks per grid - **`blockIdx`** - and
  - the identifier for threads per block - **`threadIdx`** -
  - the number of threads per block - **`blockDim`** - which comes from the **`dimBlock`** parameter

# Example

- Consider the DAXPY example

```
// Invoke DAXPY
daxpy(n, 2.0, x, y);
// DAXPY in C
void daxpy(int n, double a, double *x, double *y)
{
      for (int i = 0; i < n; ++i)
            y[i] = a*x[i] + y[i];
}
```

# Example

- In the CUDA version, we launch:
  - n threads, one per vector element
  - with 256 CUDA Threads per thread block
  - in a multithreaded SIMD Processor

```
// Invoke DAXPY with 256 threads per Thread Block
__host__
int nblocks = (n+ 255) / 256;
      daxpy<<<nblocks, 256>>>(n, 2.0, x, y);
// DAXPY in CUDA
__device__
void daxpy(int n, double a, double *x, double *y)
{
      int i = blockIdx.x*blockDim.x + threadIdx.x;
      if (i < n) y[i] = a*x[i] + y[i];
}
```

# Example

- The GPU function calculates the corresponding element **index i based on** the **block ID**, the **number of threads per block**, and the **thread ID**
- If this index is within the array (i < n), it performs the multiply and add

```
// Invoke DAXPY with 256 threads per Thread Block
__host__
int nblocks = (n+ 255) / 256;
      daxpy<<<nblocks, 256>>>(n, 2.0, x, y);
// DAXPY in CUDA
__device__
void daxpy(int n, double a, double *x, double *y)
{
      int i = blockIdx.x*blockDim.x + threadIdx.x;
      if (i < n) y[i] = a*x[i] + y[i];
}
```

# Threads and Blocks

- The C version has:
  - a loop where each iteration is independent of the others
  - This allows the loop to be transformed straightforwardly into a parallel code
  - **each loop iteration** becomes an **independent thread**
- The programmer determines the parallelism in CUDA explicitly by specifying
  - the grid dimensions and
  - the number of threads per SIMD Processor
- By assigning a single thread to each element, there is **no need to synchronize** among threads when writing results to memory

# Threads and Blocks

- A **thread** is associated with **each data element**
  - *CUDA threads*, with thousands of which for various styles of parallelism (multithreading, SIMD, MIMD, ILP)
- Threads are organized into **blocks**
  - ***Thread Blocks***: groups of up to 512 elements
  - ***Multithreaded SIMD Processor***: hardware that executes a whole thread block (32 elements executed per thread at a time)
- Blocks are organized into a **grid**
  - **Blocks are executed independently** and in any order
  - Different **blocks cannot communicate directly** but can *coordinate* using atomic memory operations in GPU Main Memory

# Threads and Blocks

- The **programmer** determines the parallelism in CUDA explicitly by specifying the **grid dimensions** and the **number of threads per SIMD Processor**
- By assigning a single thread to each element, there is **no need to synchronize** among threads when writing results to memory

- **GPU hardware** handles **parallel execution** and **thread management** (not done by applications or OS)
  - A multiprocessor composed of multithreaded SIMD processors
  - A Thread Block Scheduler

# Threads and Blocks

- Performance programmers must keep the GPU hardware in mind when writing in CUDA
- They need:
  - **To keep groups of 32 threads together** in control flow to get the best performance from multithreaded SIMD Processors
  - **To create many more threads** per multithreaded SIMD Processor to hide latency to DRAM.
  - **To keep the data addresses localized** in one or a few blocks of memory to get the expected memory performance
- Like many parallel systems, a compromise between productivity and performance is for CUDA to include intrinsics **to give programmers explicit control of the hardware**

# NVIDIA GPU Computational Structures

- Similarities between vector architectures and GPUs:
  - Work well with data-level parallel problems
  - Scatter-gather transfers
  - Mask registers
  - Many registers (GPUs have more registers)

- Differences between vector architectures and GPUs:
  - No scalar processor (GPU in hw, Vector arch in sw)
  - GPUs use multithreading to hide memory latency
  - GPUs have many functional units, as opposed to a few deeply pipelined units like a vector processor

# NVIDIA GPU Computational Structures

- **Efficient code for both** vector architectures and GPUs requires programmers to think in **groups of SIMD operations**

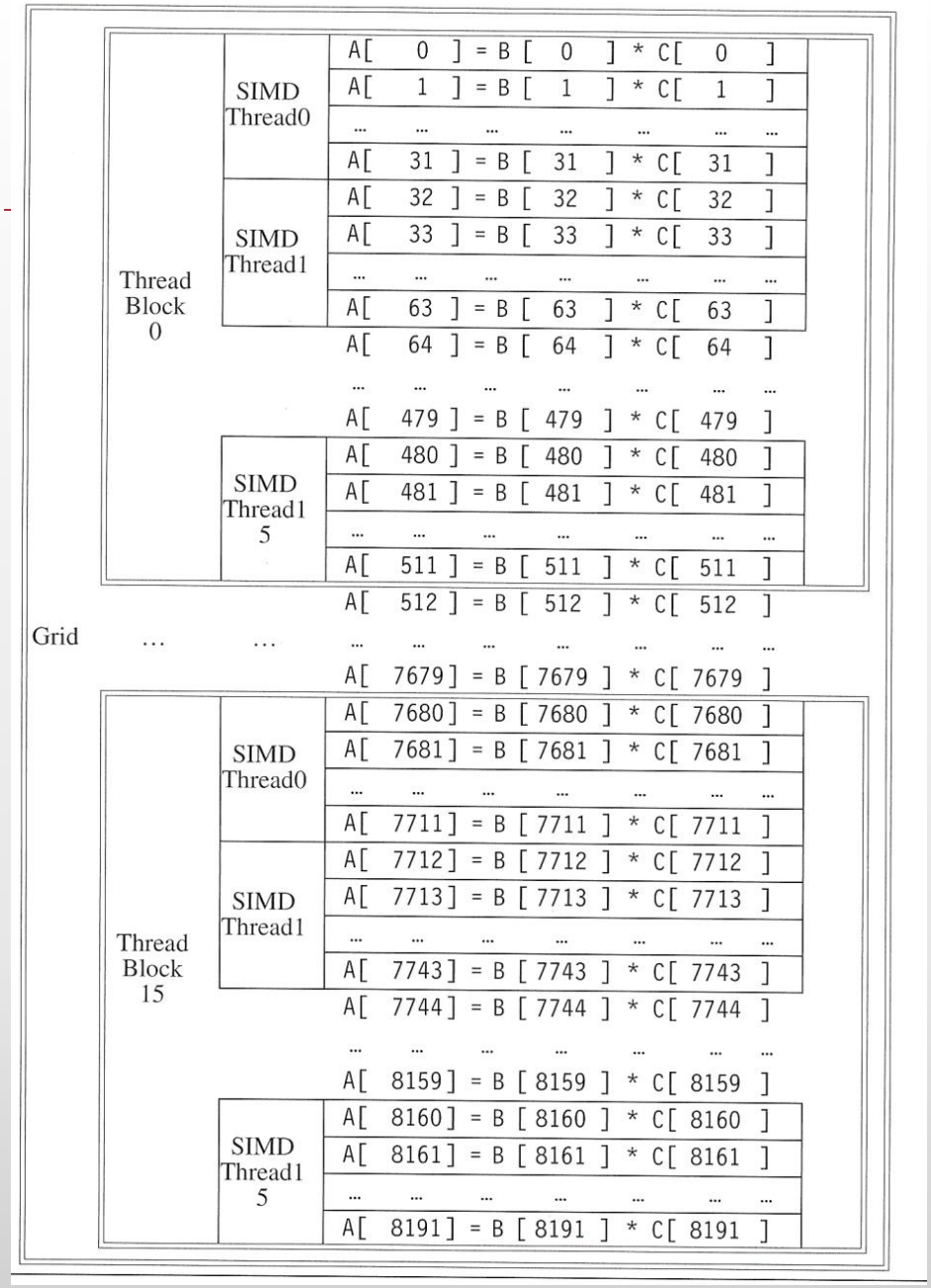
- A *Grid* is the code that runs on a GPU that consists of a set of *Thread Blocks*
- The analogy is :
  - between a **grid** and a **vectorizabled loop** and
  - between a **Thread Block** and the **body of that loop** (after it has been strip-mined, for a full computation loop - *Strip mining is the generation of code such that each vector operation is done for a size less than or equal to the MVL*)

# Example

- Multiply two vectors of length 8192

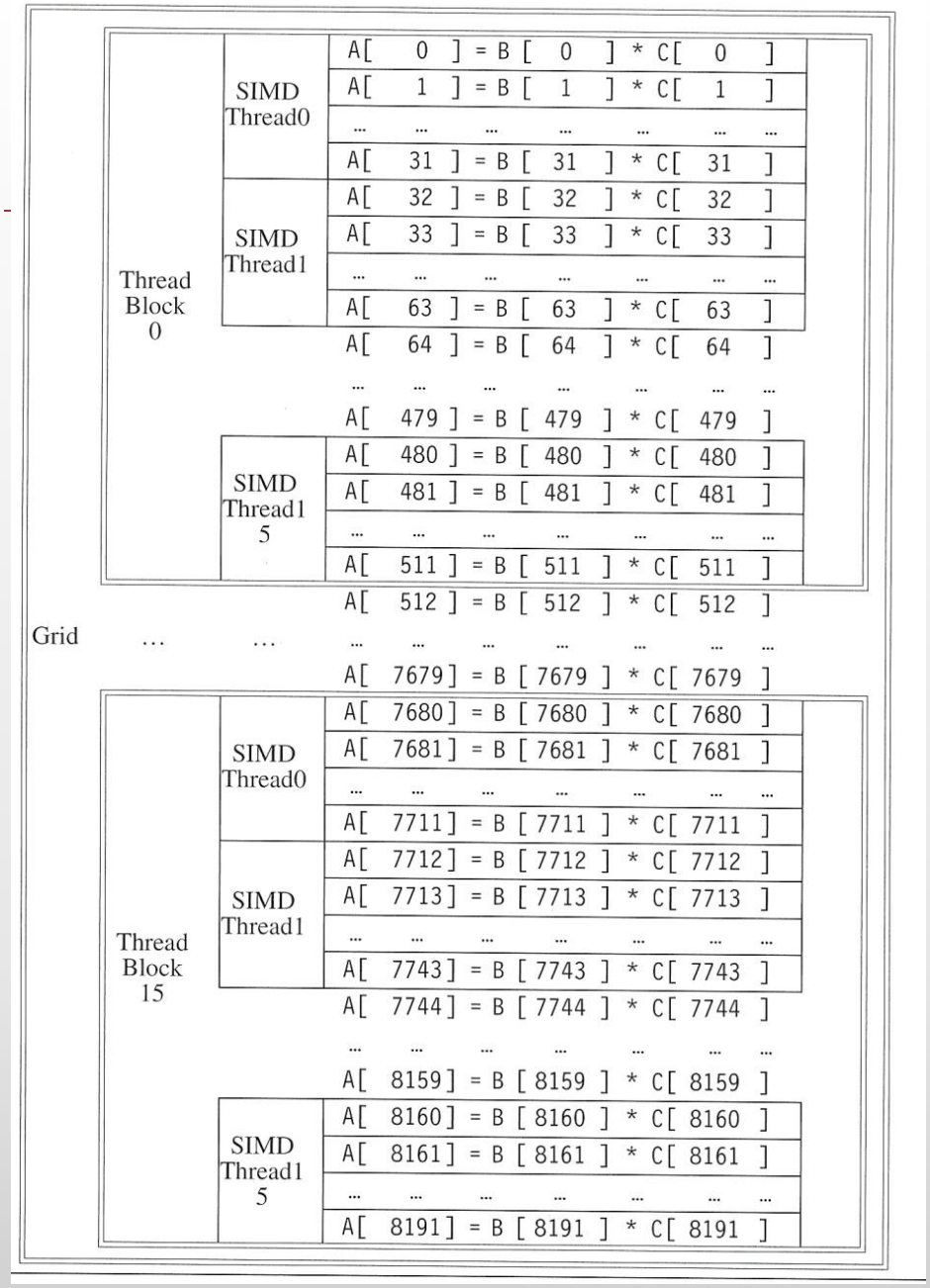
The mapping is:

- **Grid** → vectorizable loop
- **Thread Blocks** → SIMD basic blocks
- **threads of SIMD instructions** → vector-vector multiply



# Example

- Multiply two vectors of length 8192
- Code that works over all elements is the **grid**
- Thread blocks break the grid manageable sizes
  - Grid is composed of blocks with up to 512 elements
  - SIMD instruction executes 32 elements at a time
  - The number of Threads Blocks is 8192/512=16



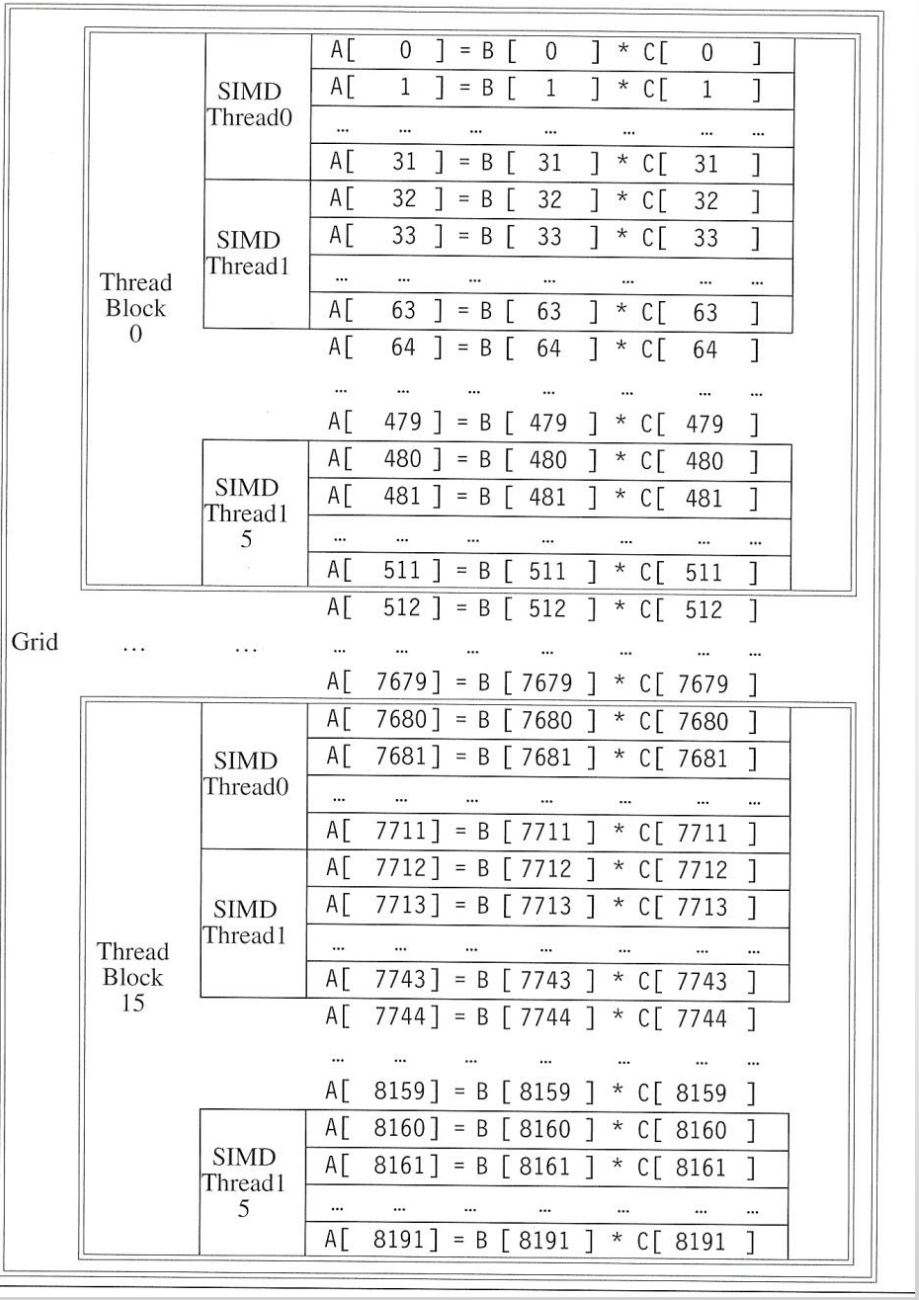
# Example

## Thread Block:

- Analogous to a strip-mined vector loop with vector length of 32
- Is assigned to a ***multithreaded SIMD Processor*** by the ***Thread Block Scheduler***

## The Thread Block Scheduler:

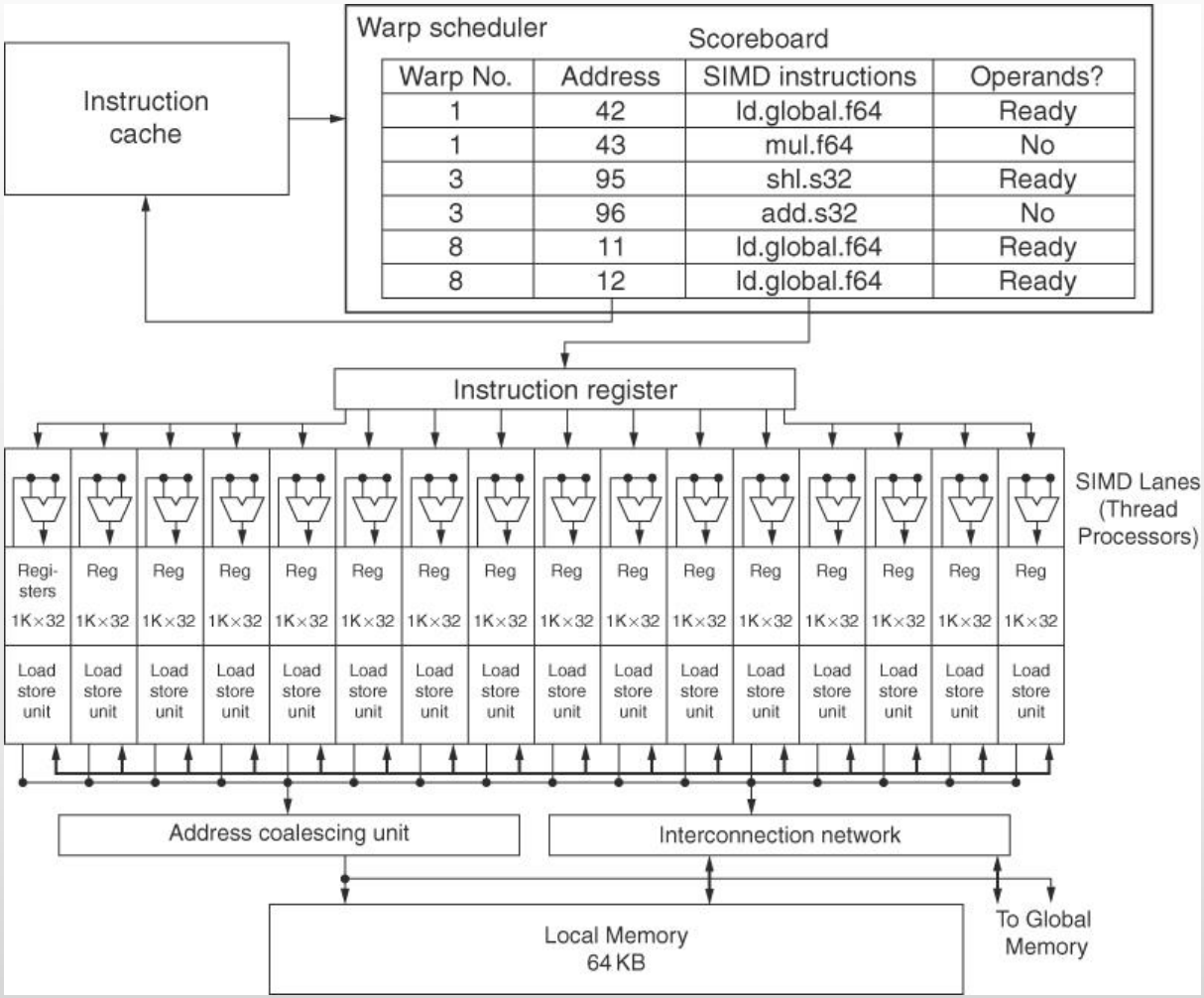
- Determines the number of thread blocks needed for the loop and
- Keeps allocating them to different multithreaded SIMD Processors until the loop is completed



# NVIDIA GPU Computational Structures

A multithreaded SIMD Processor has many **parallel functional units: *SIMD Lanes***

***SIMD Lanes*** are similar to the Vector Lanes (but GPUs have many instead of a few, deeply pipelined, as a Vector Processor)



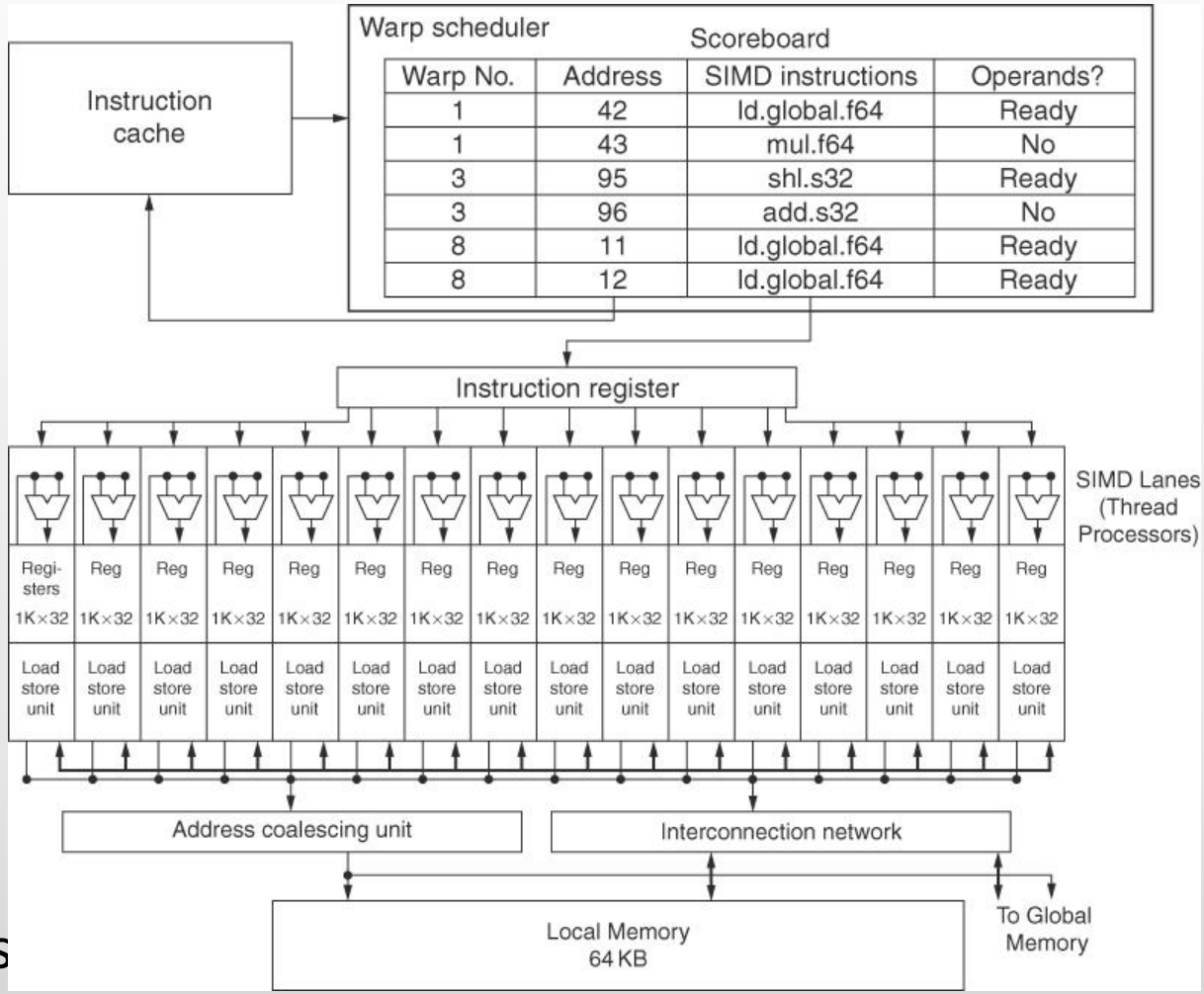

Simplified block diagram of a multithreated SIMD Processor

# NVIDIA GPU Computational Structures

A GPU is a **multiprocessor** composed of **multithreaded SIMD Processors**

The GPU hardware

- contains a collection of multithreaded SIMD Processors that execute a Grid of Thread Blocks (bodies of vectorized loop)

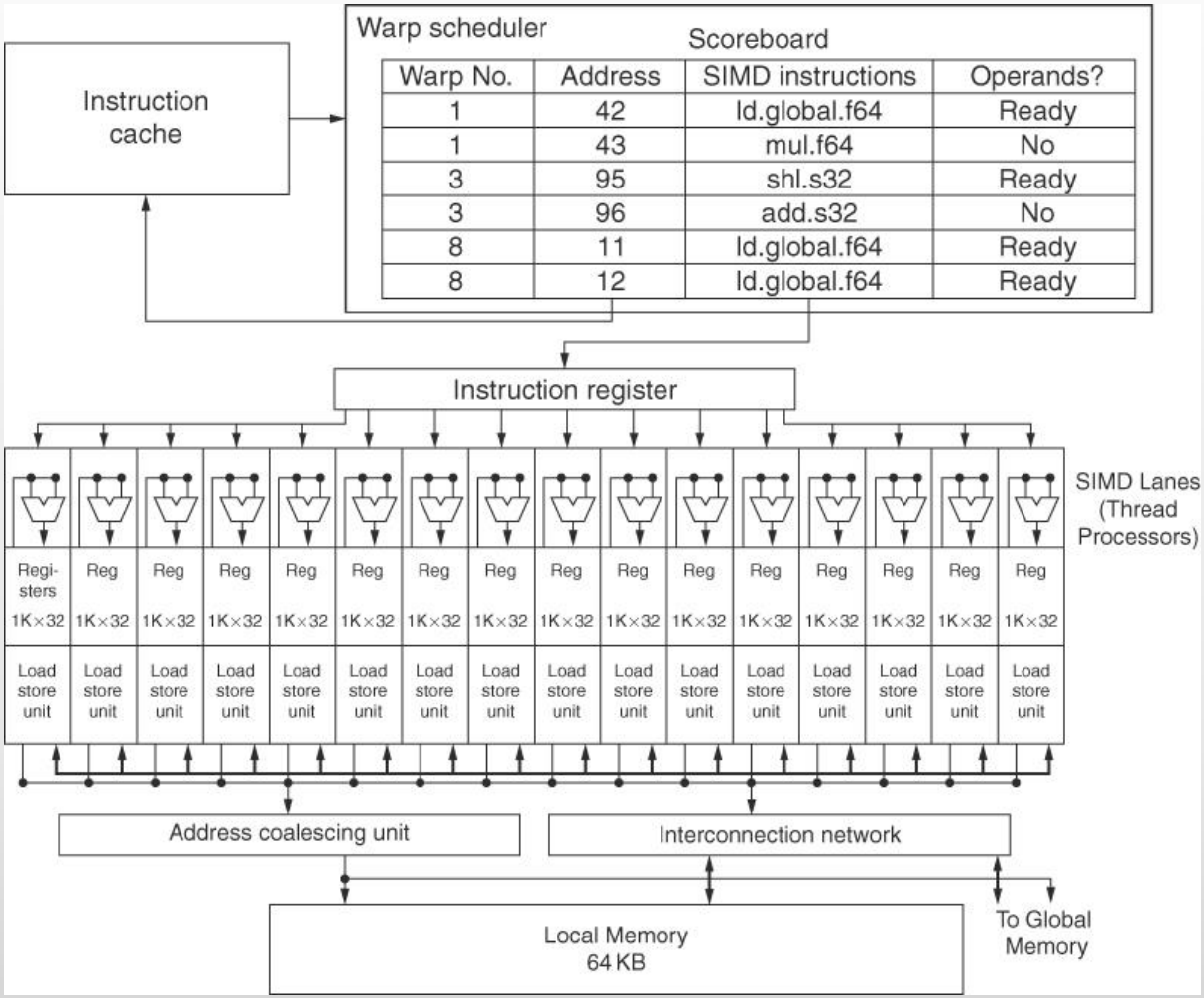



Simplified block diagram of a multithreated SIMD Processor

# NVIDIA GPU Computational Structures

Multithreaded SIMD Processors

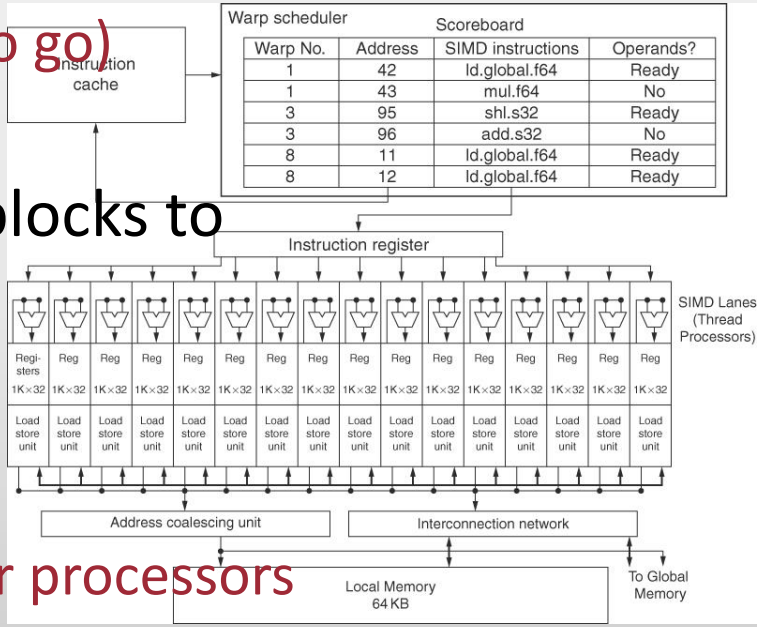
- **full processors** with **separate PCs**
- programmed using threads
- the machine object that the hw creates, manages, schedules, and executes is a *thread of SIMD instructions*




Simplified block diagram of a multithreated SIMD Processor

# NVIDIA GPU Computational Structures

- *Threads of SIMD instructions*
  - Each has its own PC
  - Thread scheduler uses scoreboard to dispatch
  - No data dependences between threads!
  - Keeps track of threads of SIMD instructions (to see which SIMD instruction is ready to go)
    - Hides memory latency
- Thread block scheduler schedules blocks to SIMD processors
- Within each SIMD processor:
  - SIMD lanes
  - Wide and shallow compared to vector processors



# NVIDIA GPU Computational Structures

- NVIDIA GPU has registers (32,768 in Fermi)
  - Divided into lanes (Like a vector processor)
  - Each SIMD thread is limited to 64 registers
  - SIMD thread is limited to no more than 64 registers, that can be:
    - 64 vector registers of 32 32-bit elements
    - 32 vector registers of 32 64-bit elements

- In the **vector multiply example**, each multithreaded SIMD Proc:
  - must load 32 elements of two vectors from memory into registers
  - perform the multiply by reading and writing registers
  - store the product from registers into memory

# NVIDIA Instruction Set Architecture

- Instruction Set target of the NVIDIA compiler is an abstraction of the hardware instruction set
  - *Parallel Thread Execution (PTX)*
  - The hardware instruction set is hidden from the programmer
  - PTX uses virtual registers → compiler figures out how many physical vector registers a SIMD thread needs
  - an optimizer divides the available register storage between the SIMD threads
  - Translation to machine code is performed in software

# NVIDIA Instruction Set Architecture

The format of a PTX instruction is

```
opcode.type d, a, b, c;
```

- **d** is the destination operand; **a**, **b**, and **c** are source operands
- The operation type is one of the following:

| Type | .type Specifier |
|---|---|
| Untyped bits 8, 16, 32, and 64 bits | .b8, .b16, .b32, .b64 |
| Unsigned integer 8, 16, 32, and 64 bits | .u8, .u16, .u32, .u64 |
| Signed integer 8, 16, 32, and 64 bits | .s8, .s16, .s32, .s64 |
| Floating Point 16, 32, and 64 bits | .f16, .f32, .f64 |

# NVIDIA Instruction Set Architecture

The groups of instructions are:

- **`Arithmetic`**
- **`Special Functions (mathematical)`**
- **`Logical`**
- **`Memory access`**
- **`Control Flow`**
- The control flow instructions are functions **`call`** and **`return`**, thread **`exit`**, **`branch`**, and barrier synchronization for threads within a thread block (**`bar.sync`**)

# Arithmetic group

```
arithmetic .type = .s32, .u32, .f32, .s64, .u64, .f64
add.type       add.f32 d, a, b       d = a + b;
sub.type       sub.f32 d, a, b       d = a - b;
mul.type       mul.f32 d, a, b       d = a * b;
mad.type       mad.f32 d, a, b, c    d = a * b + c;
div.type       div.f32 d, a, b       d = a / b;
rem.type       rem.u32 d, a, b       d = a % b;
abs.type       abs.f32 d, a          d = |a|;
neg.type       neg.f32 d, a          d = 0 - a;
min.type       min.f32 d, a, b       d = (a < b)? a:b;
max.type       max.f32 d, a, b       d = (a > b)? a:b;
setp.cmp.type setp.lt.f32 p, a, b    p = (a < b);
numeric .cmp = eq, ne, lt, le, gt, ge;
unordered cmp = equ, neu, ltu, leu, gtu, geu, num, nan
```

**ecc.**

# Special and logic groups

```
special .type = .f32 (some .f64)
rcp.type rcp.f32 d, a d = 1/a; reciprocal
sqrt.type sqrt.f32 d, a d = sqrt(a); square root
rsqrt.type rsqrt.f32 d, a d = 1/sqrt(a); reciprocal square root
sin.type sin.f32 d, a d = sin(a); sine
cos.type cos.f32 d, a d = cos(a); cosine
lg2.type lg2.f32 d, a d = log(a)/log(2) binary logarithm
ex2.type ex2.f32 d, a d = 2 ** a; binary exponential
logic.type = .pred,.b32, .b64
and.type and.b32 d, a, b d = a & b;
or.type or.b32 d, a, b d = a | b;
xor.type xor.b32 d, a, b d = a ^ b;
not.type not.b32 d, a, b d = ~a; one's complement
cnot.type cnot.b32 d, a, b d = (a==0)? 1:0; C logical not
shl.type shl.b32 d, a, b d = a << b; shift left
shr.type shr.s32 d, a, b d = a >> b; shift right
```

# Memory access and control flow groups

**Memory Access**

```
memory.space = .global, .shared, .local, .const
.type = .b8,, .u8, .s8, .b16, .b32, .b64
ld.space.type          ld.global.b32 d, [a+off]      d = *(a+off);
st.space.type          st.shared.b32 [d+off],        a *(d+off) = a;
tex.nd.dtyp.btype      tex.2d.v4.f32.f32 d, a, b     d = tex2d(a, b);
atom.spc.op.type       atom.global.add.u32 d,[a], b
                       atom.global.cas.b32 d,[a], b, c
atom.op = and, or, xor, add, min, max, exch, cas;
.spc = .global; .type = .b32
```

**Control Flow**

```
branch          @p bra target                  if (p) goto target;
call            call (ret), func, (params)     ret = func(params);
ret             ret                            return;
bar.sync        bar.sync d                     wait for threads
exit            exit                           exit;
```

# NVIDIA Instruction Set Arch.

- Sequence of PTX instructions for one iteration of DAXPY loop

```
shl.s32 R8, blockIdx, 9      ; Thread Block ID * Block size (512=2^9)
add.s32 R8, R8, threadIdx    ; R8 = i = my CUDA thread ID
ld.global.f64 RD0, [X+R8]    ; RD0 = X[i]
ld.global.f64 RD2, [Y+R8]    ; RD2 = Y[i]
mul.f64 R0D, RD0, RD4        ; Product in RD0 = RD0 * RD4 (scalar a)
add.f64 R0D, RD0, RD2        ; Sum in RD0 = RD0 + RD2 (Y[i])
st.global.f64 [Y+R8], RD0    ; Y[i] = sum (X[i]*a + Y[i])
```

- CUDA
  - assigns one CUDA Thread to each loop iteration
  - offers a unique identifier number to each thread block (`blockIdx`) and one to each CUDA Thread within a block (`threadIdx`)
  - uses the unique number to address each element in the array
  - 8192 CUDA threads are created!

# NVIDIA Instruction Set Arch.

- Sequence of PTX instructions for one iteration of DAXPY loop

```
shl.s32 R8, blockIdx, 9      ; Thread Block ID * Block size (512=2^9)
add.s32 R8, R8, threadIdx    ; R8 = i = my CUDA thread ID
ld.global.f64 RD0, [X+R8]    ; RD0 = X[i]
ld.global.f64 RD2, [Y+R8]    ; RD2 = Y[i]
mul.f64 R0D, RD0, RD4        ; Product in RD0 = RD0 * RD4 (scalar a)
add.f64 R0D, RD0, RD2        ; Sum in RD0 = RD0 + RD2 (Y[i])
st.global.f64 [Y+R8], RD0    ; Y[i] = sum (X[i]*a + Y[i])
```

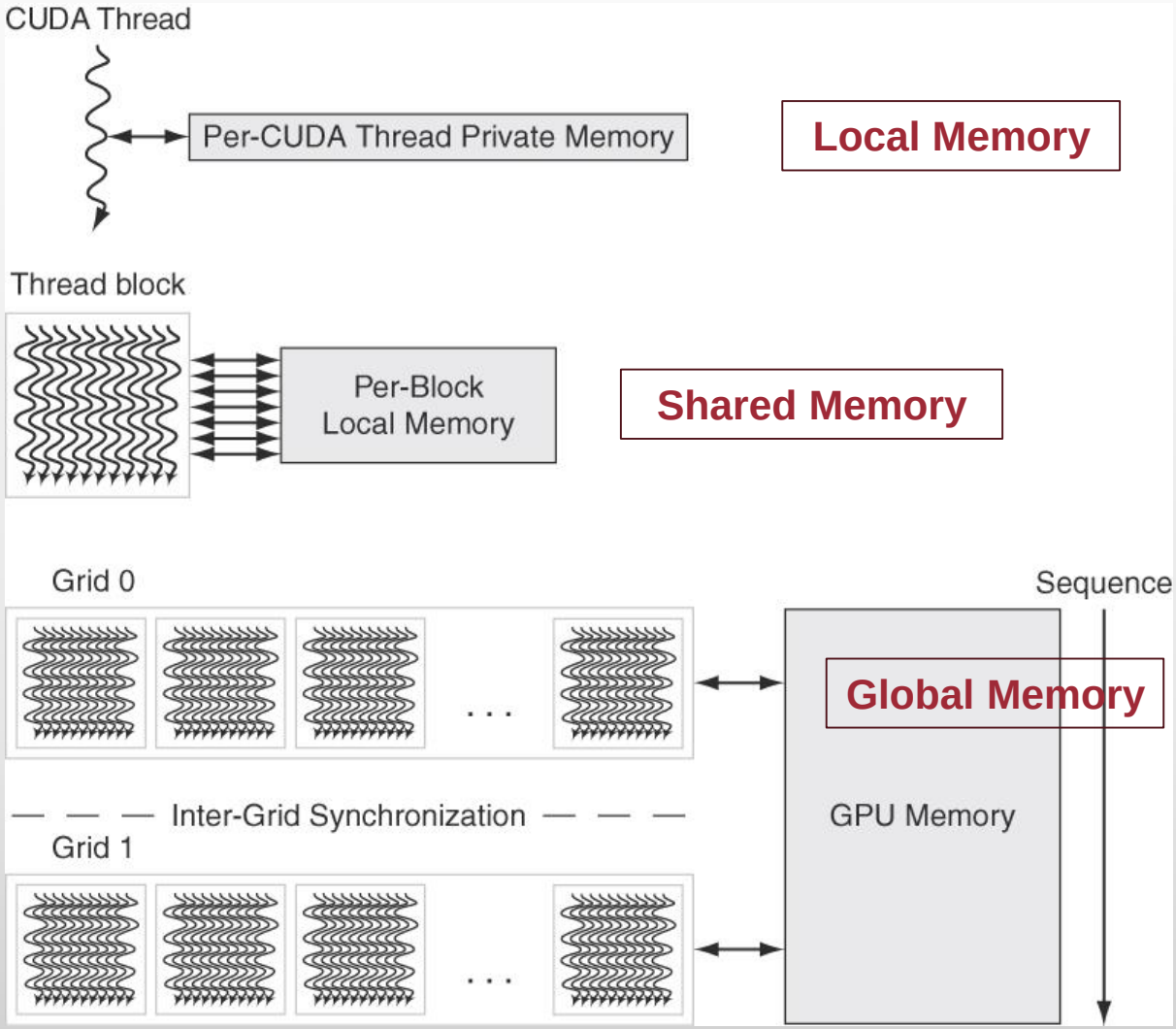
- GPUs
  - All data transfers are gather-scatter!
  - include special Address Coalescing hw to recognize when the SIMD Lanes within a thread are issuing sequential addresses
  - The GPU programmer must ensure that adjacent CUDA Threads access nearby addresses at the same time that can be coalesced, as in our example

# NVIDIA GPU Memory Structures

- Each **SIMD Lane** has private section of *off-chip* DRAM
  - "Private memory", not shared by any other lanes
  - Contains stack frame, spilling registers, and private variables
  - Recent GPUs cache this in L1 and L2 caches
- Each **multithreaded SIMD processor** also has local memory that is *on-chip*
  - Shared by SIMD lanes / threads *within a block only*
- The *off-chip* memory shared by SIMD processors is *GPU Memory*
  - Host can read and write GPU memory

# NVIDIA GPU Memory Structures

**GPU Memory** is shared by all Grids (vectorized loops) **Local Memory** is shared by all threads of SIMD instructions within a thread block (body of a vectorized loop), **Private Memory** is private to a single CUDA Thread.



# Summary

- GPUs are really just multithreaded SIMD processors, although they have:
  - more processors,
  - more lanes per processor, and
  - more multithreading hardware than traditional multicore computers
- The CUDA programmer can think of programming **thousands of threads**, although they are really executing each **block of 32 threads** on the **many lanes** of the many SIMD Processors
- The CUDA programmer who wants good performance keeps in mind that these **threads are blocked and executed 32 at a time** and that **addresses need to be to adjacent** addresses to get good performance from the memory system

# GPUs from CUDA point of view

## Programming Massively Parallel Processors

**D.B. Kirk W. W. Hwu**

- Chapter 3 - Introduction to Data Parallelism and CUDA C
  - **Sections 3.2 - 3.6**
- Chapter 4 - Data Parallel Execution Model
  - **Sections 4.5 - 4.7**
- Chapter 5 - CUDA Memories
  - **Sections 5.2 - 5.4**

## Multicore and GPU Programming

**G. Barlas**

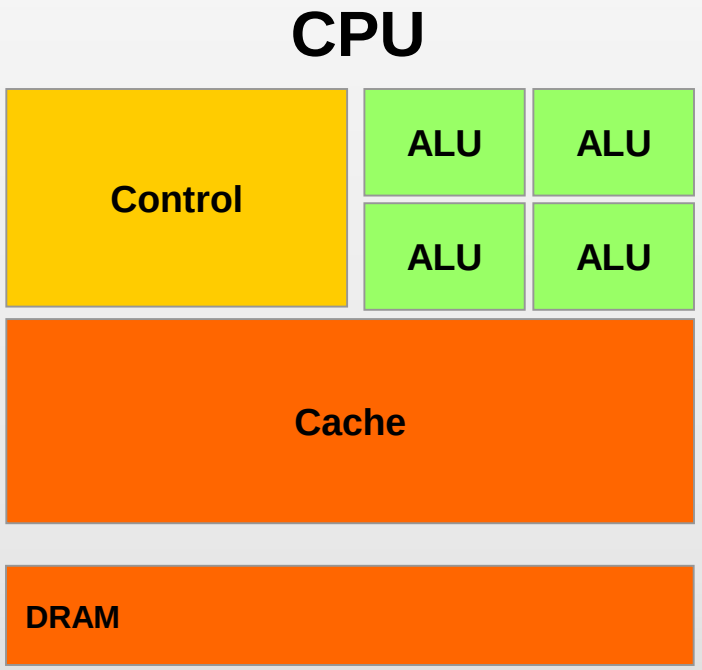
- Chapter 6 - GPU Programming
  - **Sections 6.2 - 6.7**

# CUDA Programming Model

- The GPU is viewed as a compute **device** that:
  - Is a coprocessor to the CPU (**host**)
  - Has its own DRAM (**device memory**)
  - Runs many **threads in parallel**
- Data-parallel portions of an application are executed on the device as **kernels** which run in parallel on many threads
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few

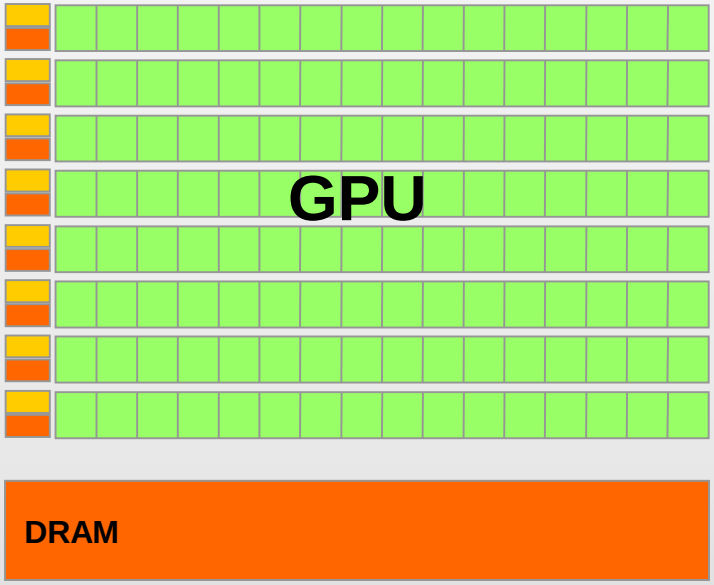
# CPUs: Latency Oriented Design

- Large caches
  - Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
  - Branch prediction for reduced branch latency
  - Data forwarding for reduced data latency
- Powerful ALU
  - Reduced operation latency



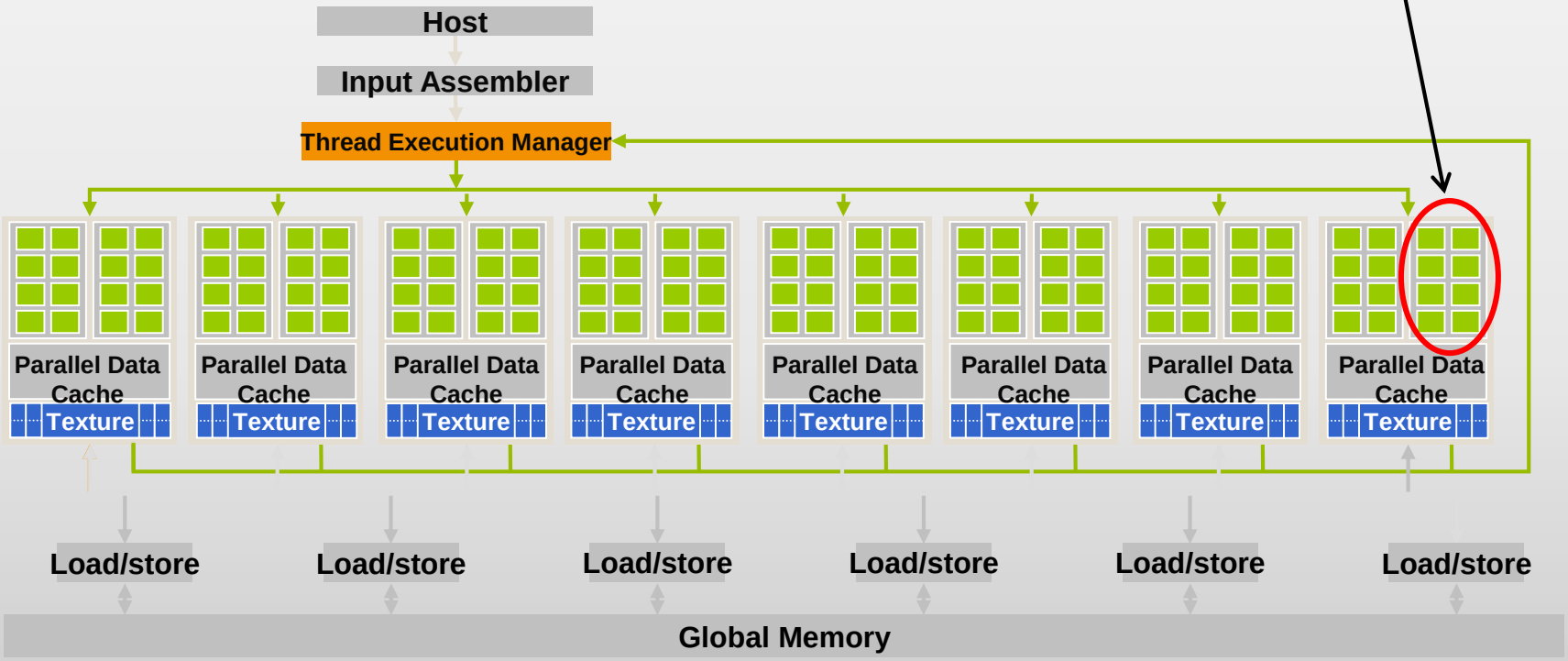
# GPUs: Throughput Oriented Design

- Small caches
  - To boost memory throughput
- Simple control
  - No branch prediction
  - No data forwarding
- Energy efficient ALUs
  - Many, long latency but heavily pipelined for high throughput
- Require massive number of threads to tolerate latencies



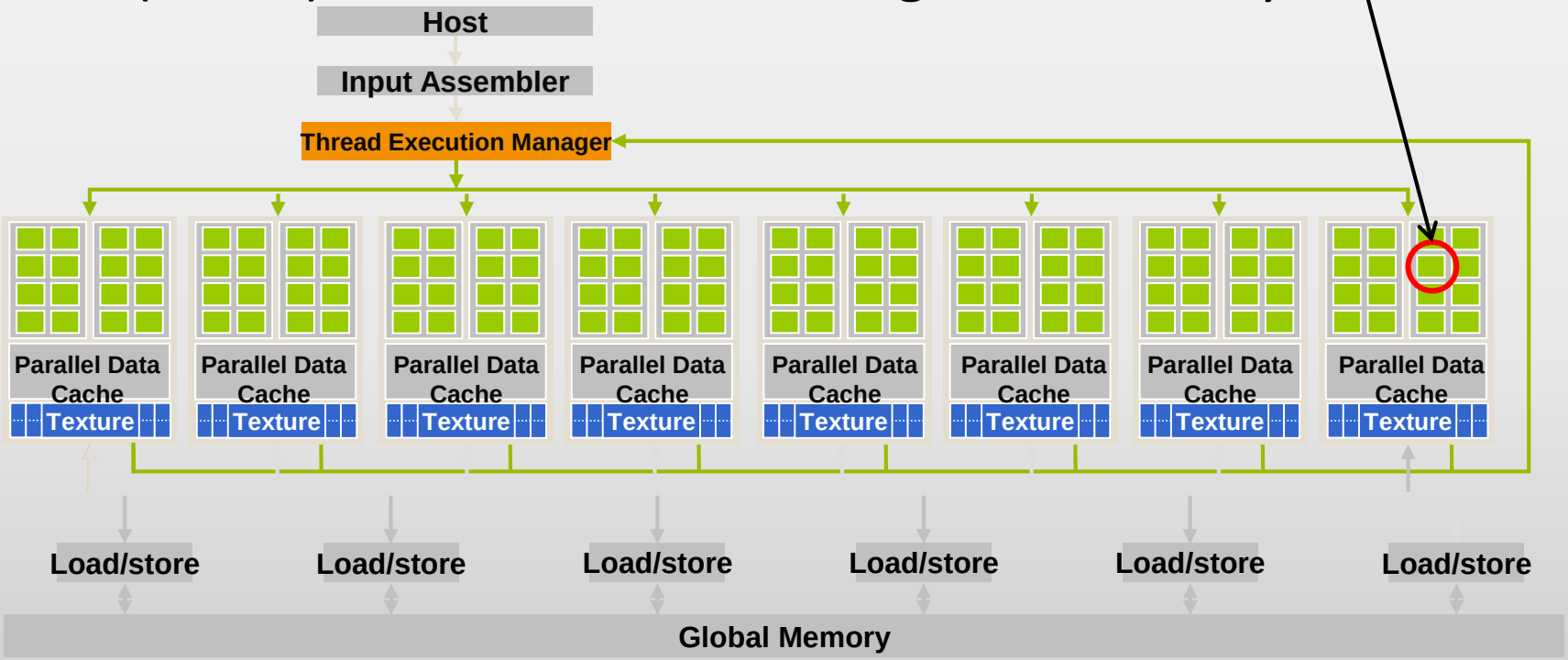
# GPU Architecture

- A typical CUDA-capable GPU can be organized into
  - An array of highly threaded Streaming Multiprocessors (SMs)
  - In Figure, two SMs form a building block; but, the number of SMs in a building block can vary



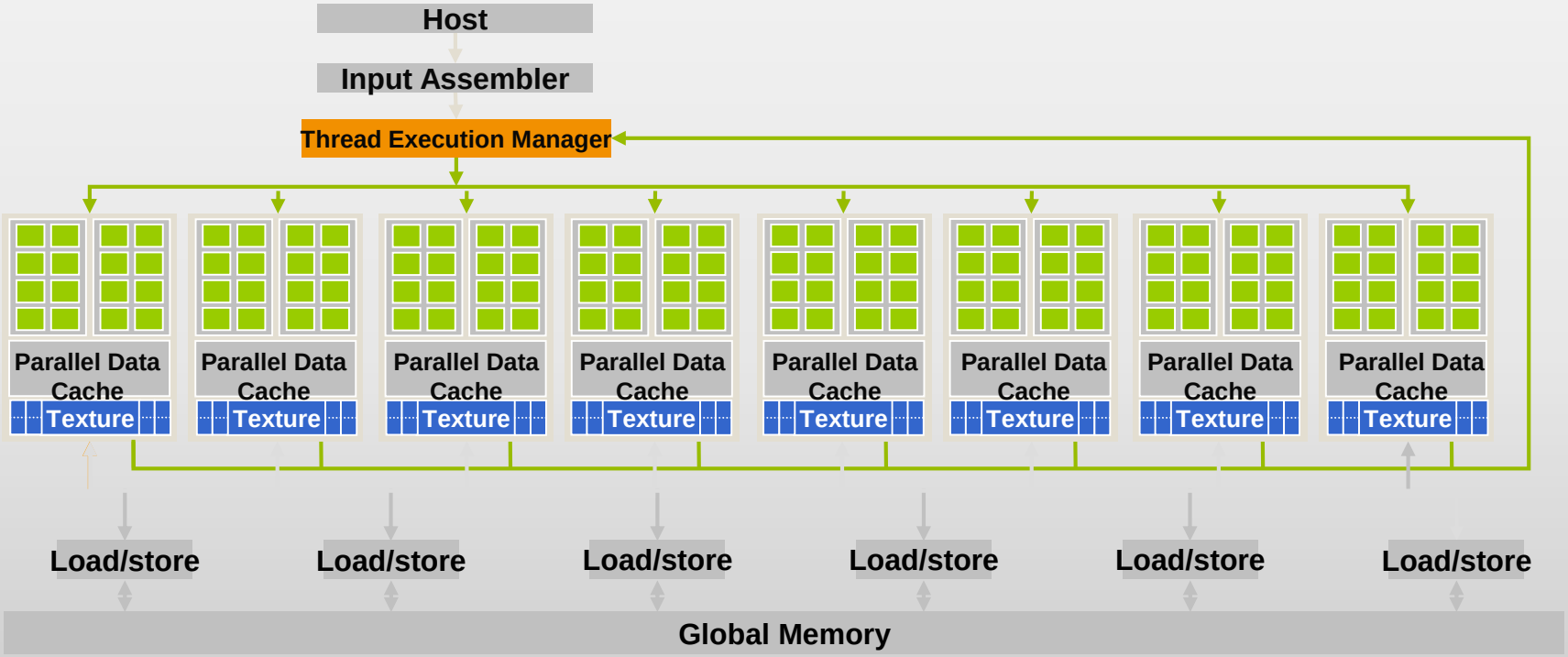
# GPU Architecture

- Each SM has a number of streaming processors (SPs) that share control logic and instruction cache
- Each GPU currently comes with a graphics double data rate (GDDR) DRAM, referred to as global memory



# GPU Architecture

- The parallel **G80** chip has 128 SPs (16 SMs, 8 SPs)
  - Each SP has a multiply–add (MAD) unit and a multiply unit
  - Produces a total of over 500 gigaflops
  - **GT200** (240 SPs) exceeds 1 teraflops - **GTX680** 1,5 teraflops



# CUDA Program Structure

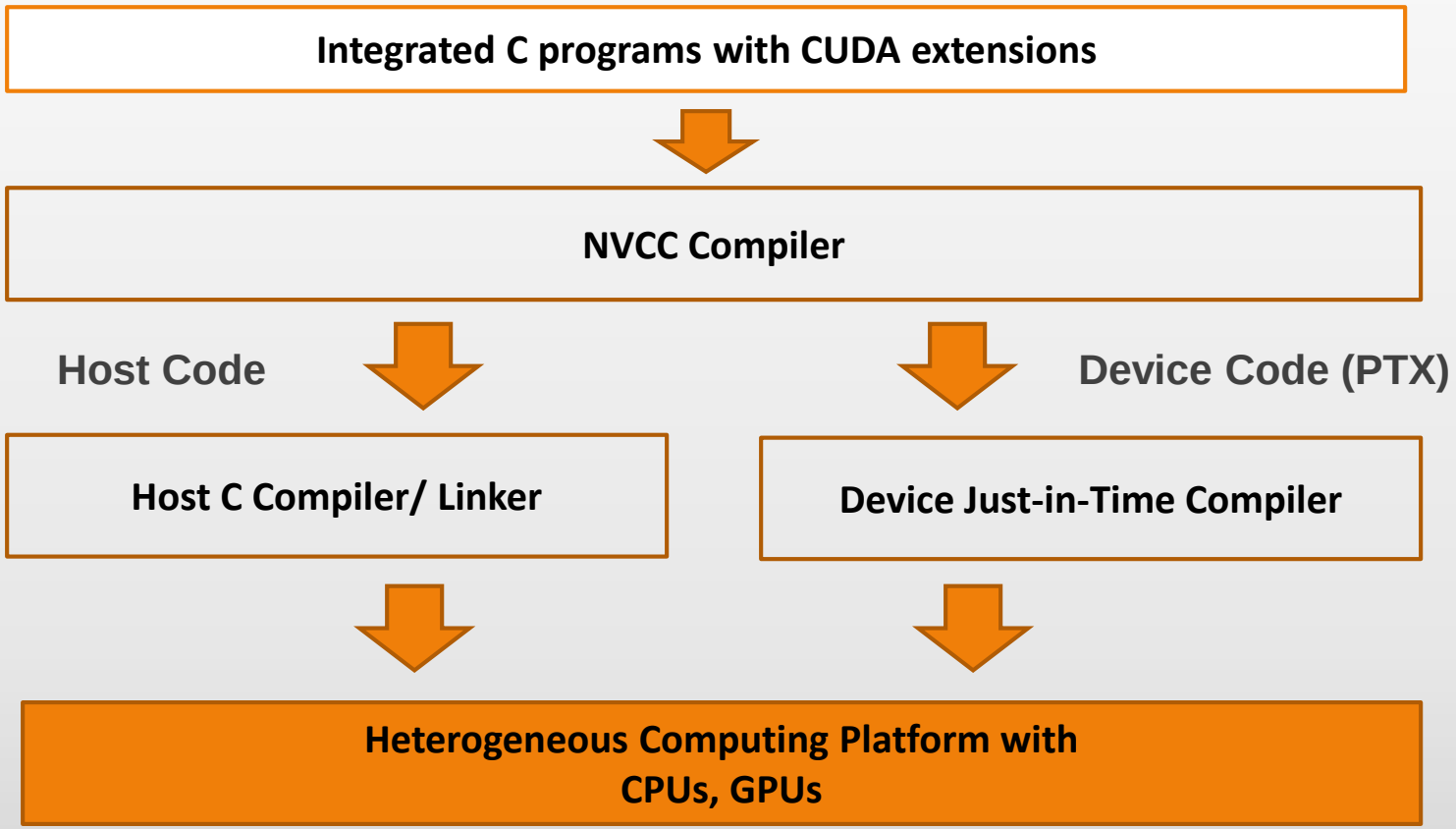
- The structure of a CUDA program reflects the computing system consisting of
  - a **host**, which is a traditional central processing unit (CPU)
  - one or more **devices** (GPUs)

- A CUDA program is a unified source code encompassing both host and device code
- The NVIDIA C compiler - **nvcc** - separates the two during the compilation process

# CUDA Program Structure

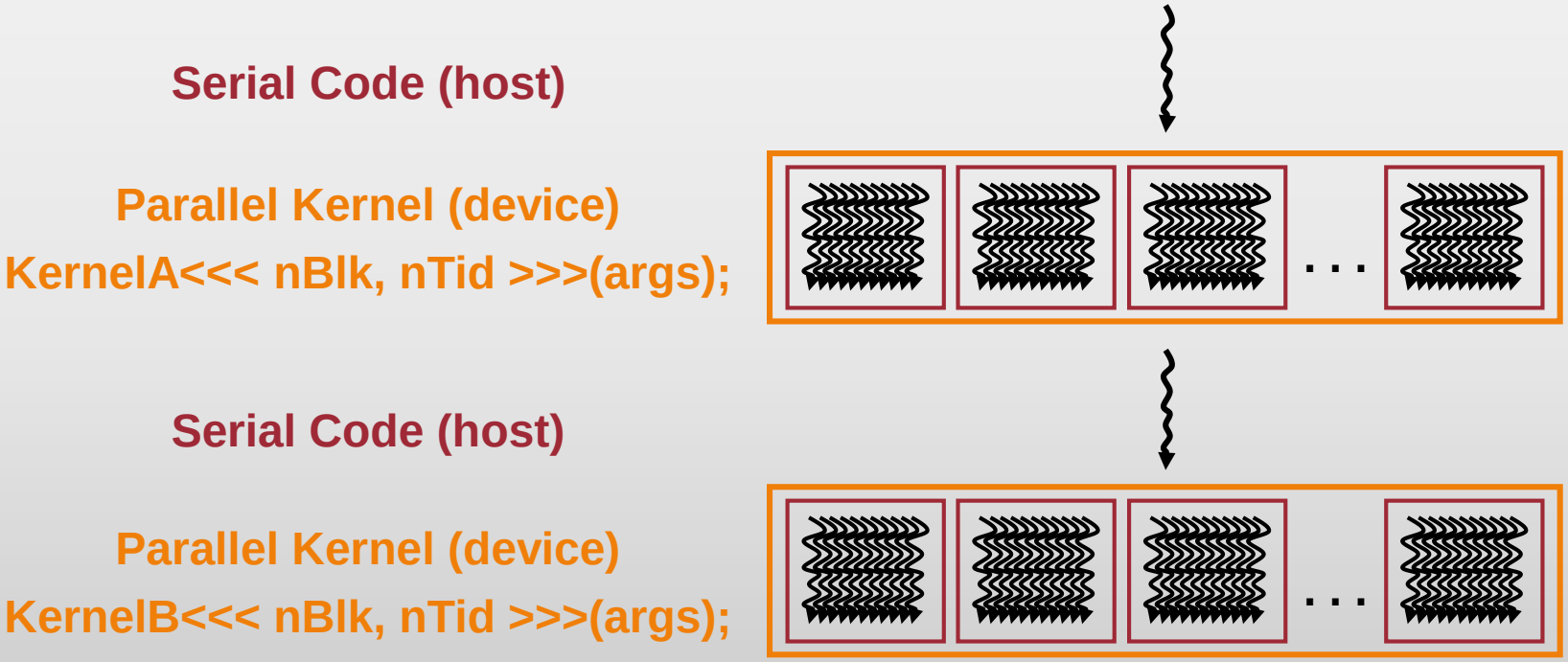
- The host code is:
  - straight ANSI C code
  - it is further compiled with the host's standard C compilers and runs as an ordinary CPU process
- The device code is:
  - written using ANSI C extended with **keywords** for labeling data-parallel functions, called **kernels**, and their associated **data structures**
  - The device code is typically further compiled by the **nvcc** and executed on a GPU device

# Compiling A CUDA Program



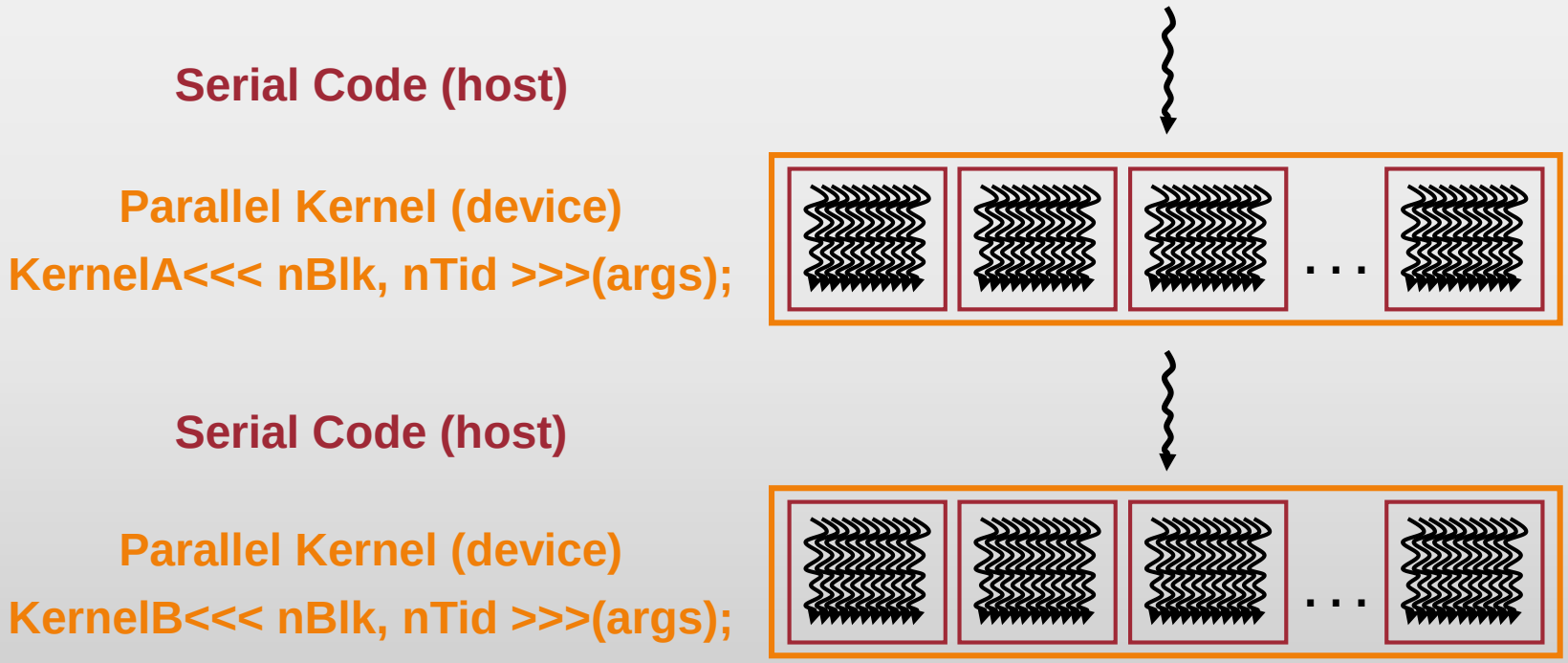
# CUDA Execution Model

- The execution starts with host (CPU) execution
- When a kernel function is launched, the execution is moved to a device (GPU), where a large number of threads are generated to take advantage of abundant data parallelism



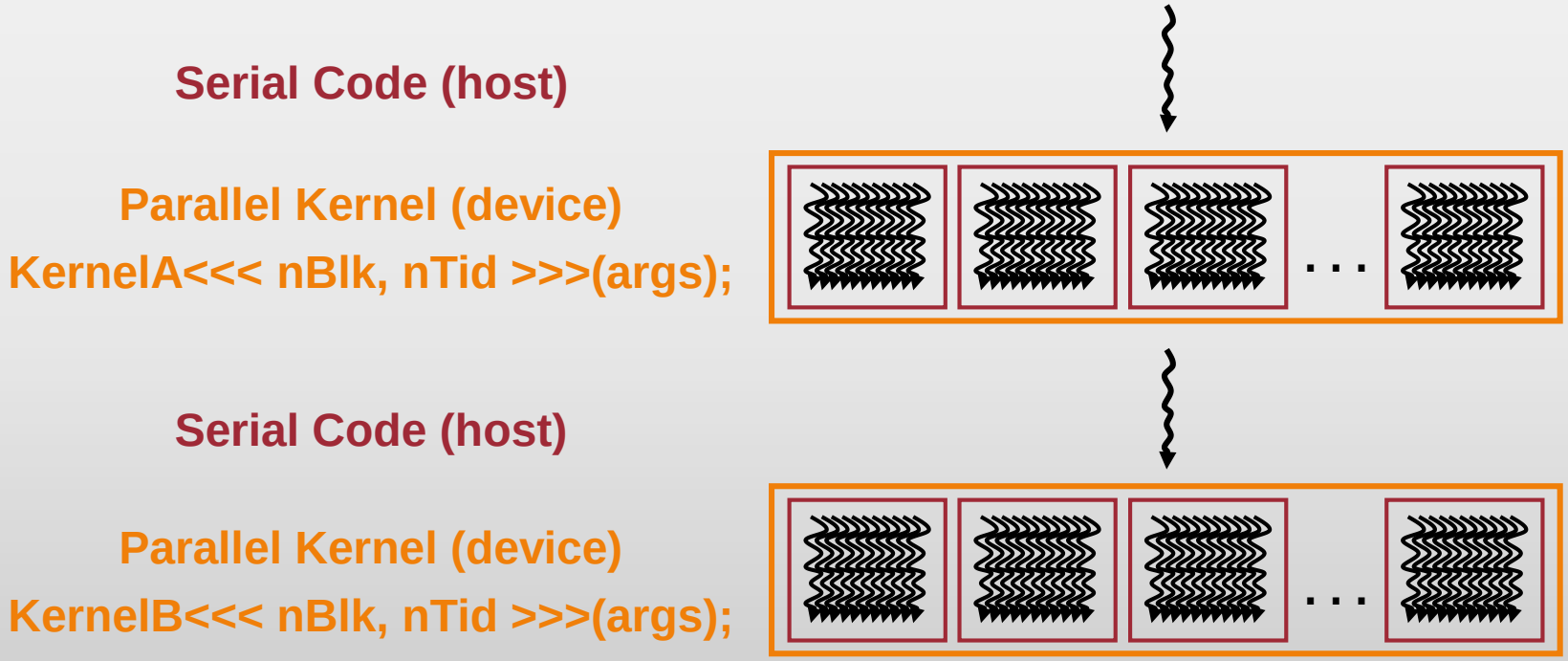
# CUDA Execution Model

- All the threads that are generated by a kernel during an invocation are collectively called a **grid**
- Figure shows the execution of **two grids of threads**



# CUDA Execution Model

- When all threads of a kernel complete their execution:
  - the corresponding grid terminates
  - the execution continues on the host until another kernel is invoked

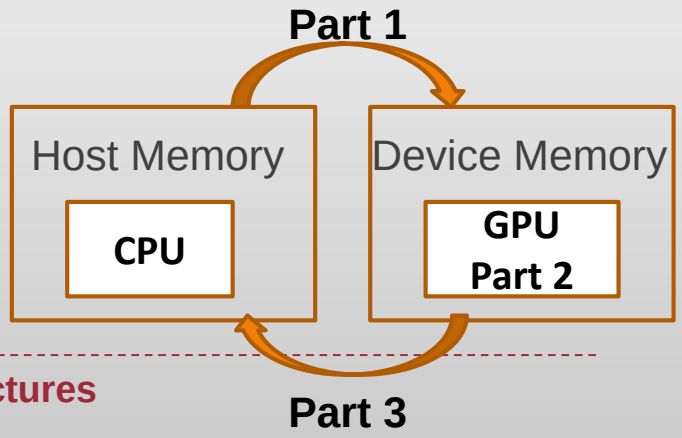


# Vector Addition – Traditional C Code

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

int main()
{
    // Memory allocation for A_h, B_h, and C_h
    // I/O to read A_h and B_h, N elements
    …
    vecAdd(A_h, B_h, C_h, N);
}
```

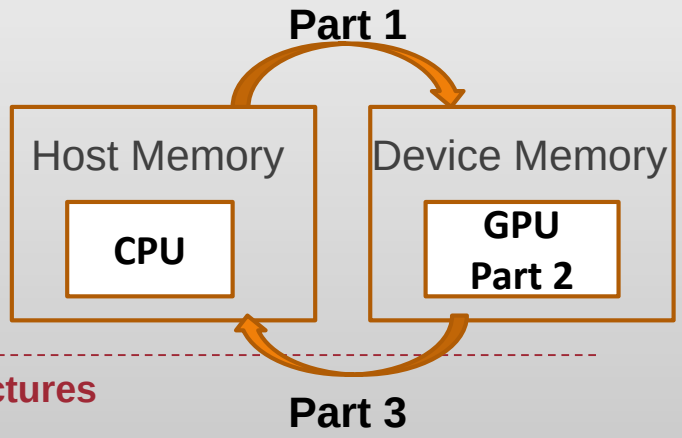
# Vector Addition – Kernel

```
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;
    …
1. // Allocate device memory for A, B, and C
     // copy A and B to device memory

2. // Kernel launch code – to have the device
     // to perform the actual vector addition

3. // copy C from the device memory
     // Free device vectors
}
```



# Device Memory and Data Transfer

- The host and devices have separate memory spaces
- To execute a kernel on a device
  - the programmer needs to **allocate memory on the device**
  - **transfer data** from the host memory to the allocated device memory
  - this corresponds to Part 1 of Figure
- After device execution
  - the programmer needs to **transfer result data** from the device memory back to the host memory
  - **free up** the device memory
  - this corresponds to Part 3 of Figure

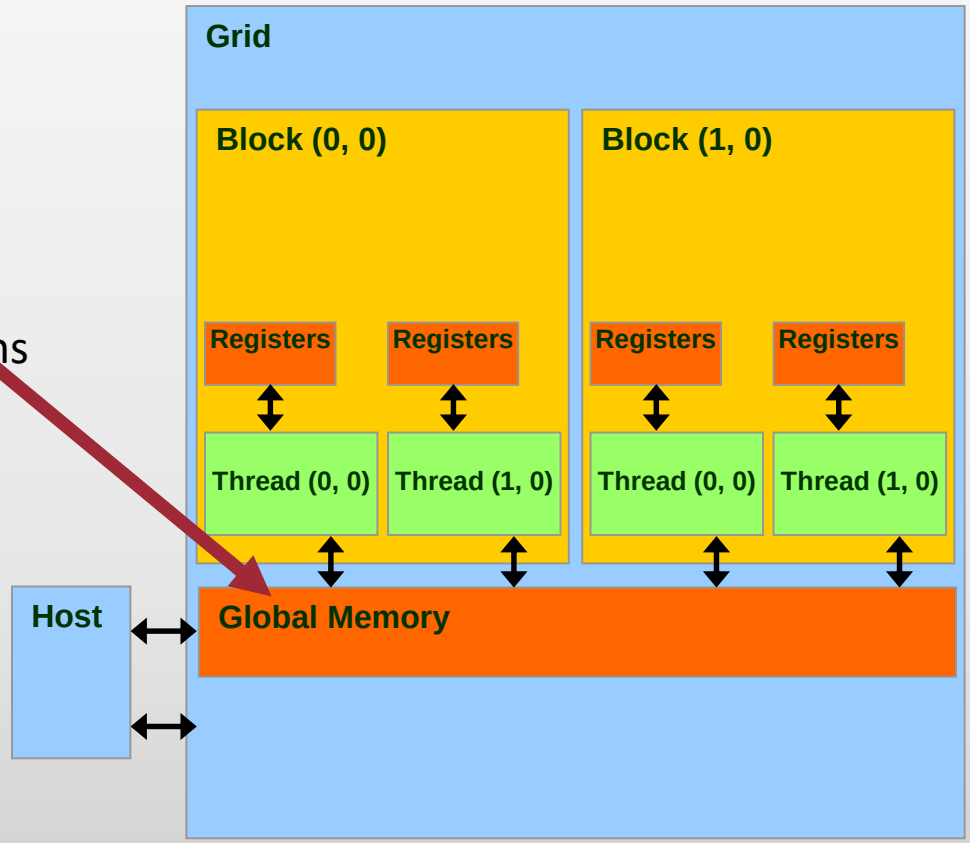


# Device Memory and Data Transfer

- The CUDA memory model is supported by API functions that help programmers to manage data in memories
- The function **`cudaMalloc()`**:
  - Called **from the host code** to allocate object in the device global memory
  - Two parameters:
    - **address of a pointer** variable to the allocated object after allocation
    - **size** of the allocated object in terms of bytes
- The function **`cudaFree()`**:
  - Frees object from device global memory
    - Pointer to freed object
- The function **`cudaMemcpy()`**: for memory data transfer

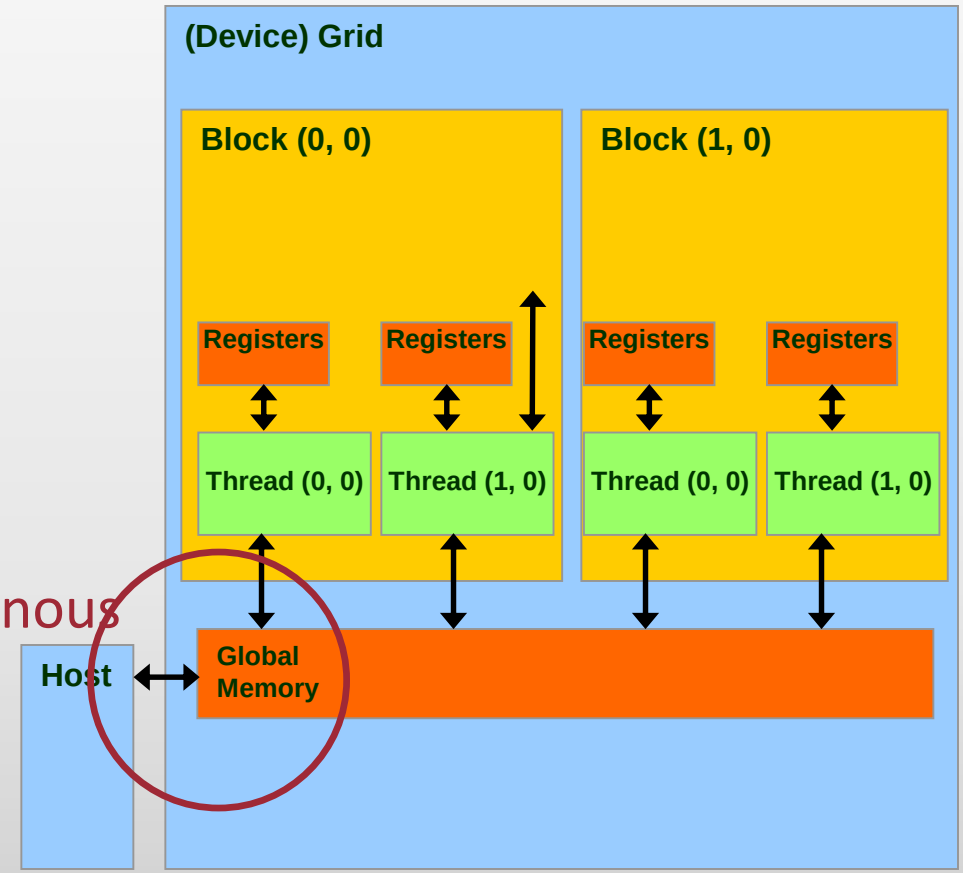
# CUDA Device Memory Management API

- cudaMalloc()
  - Allocates object in the device global memory
  - Two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** of allocated object in terms of bytes
- cudaFree()
  - Frees object from device global memory
    - **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

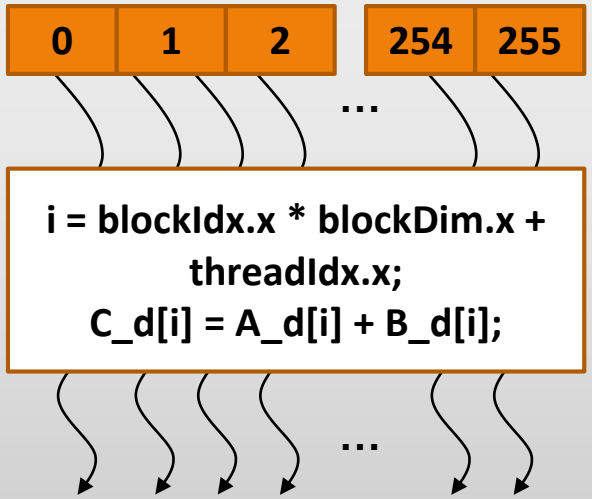


# Vector Addition – Traditional C Code

```
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;

1. // Transfer A and B to device memory
     cudaMalloc((void **) &d_A, size);
     cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
     cudaMalloc((void **) &d_B, size);
     cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

   // Allocate device memory for
      cudaMalloc((void **) &d_C, size);

2. // Kernel invocation code – to be shown later
      …
3. // Transfer C from device to host
      cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    // Free device memory for A, B, C
      cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

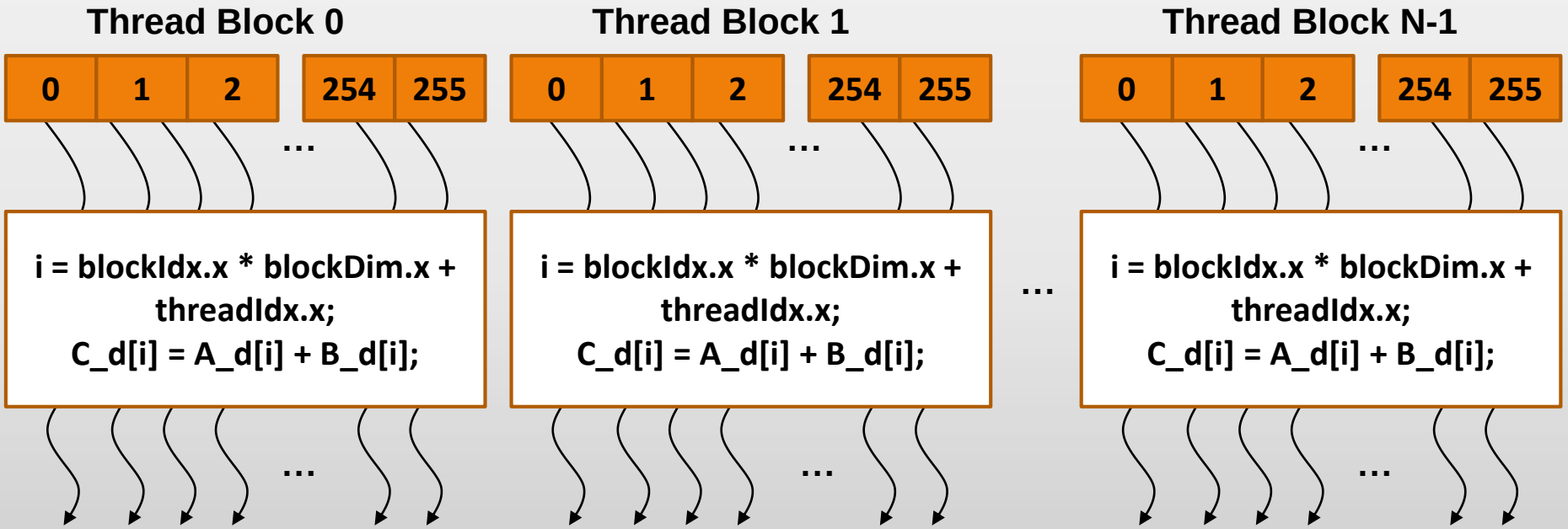
# Arrays of Parallel Threads

- A **kernel** function specifies the code to be executed by all threads during a parallel phase
  - All of these threads execute the same code
- A CUDA kernel is executed by a grid (array) of threads
  - All threads in a grid run the same kernel code (SPMD)
  - Each **thread** has an **index** that it uses to compute memory addresses and make control decisions



# Thread Blocks: Scalable Cooperation

- Thread array is divided into multiple blocks
  - Threads within a block cooperate via **shared memory, atomic operations** and **barrier synchronization**
  - Threads in different blocks cannot cooperate

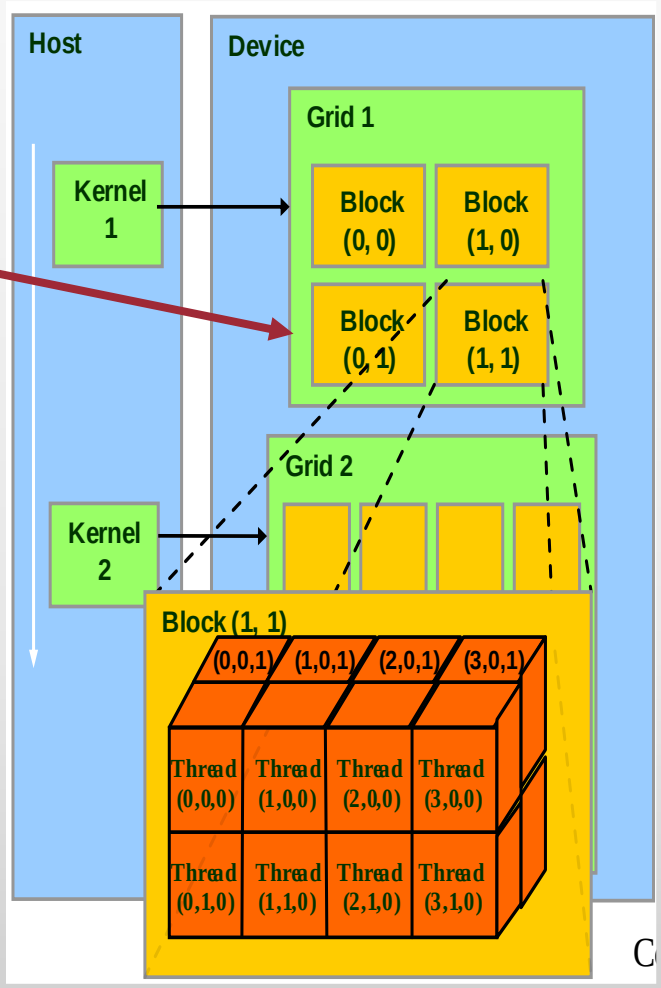


# Arrays of Parallel Threads

- When a **kernel** is invoked, it is executed as **grid** of parallel threads
- Each CUDA thread grid typically is comprised of **thousands to millions** of lightweight GPU threads per kernel invocation
- Creating enough threads to fully utilize the hardware often requires a large amount of data parallelism

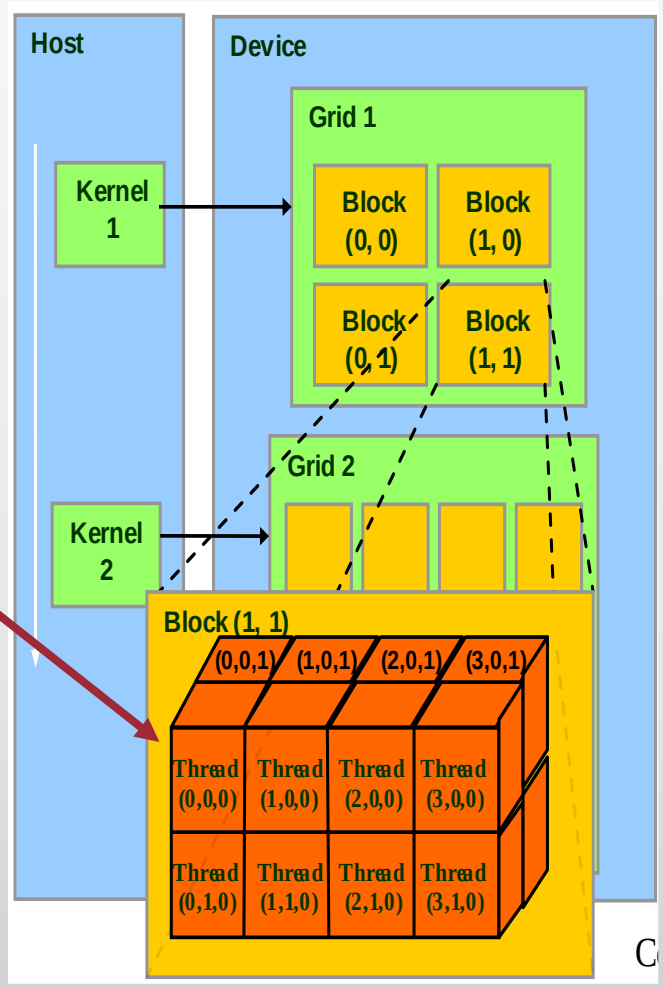
# blockIdx and threadIdx

- Threads in a grid are organized into **a two-level hierarchy**
  - ***top level***, each **grid** consists of one or more thread blocks
  - All **blocks** in a grid have the **same number of threads** organized in the same manner
  - Each **grid** is organized as a as a **three-dimensional array** of blocks
  - Each **block** has a unique **three dimensional coordinate** given by the CUDA specific keywords **`blockIdx.x`**, **`blockIdx.y`** and **`blockIdx.z`**



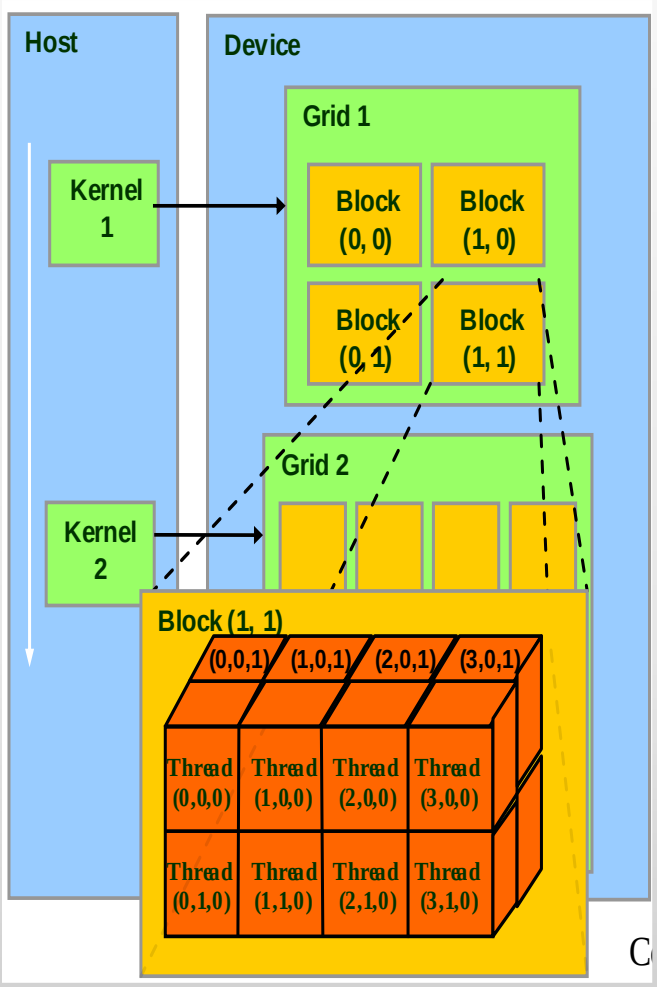
# blockIdx and threadIdx

- Threads in a grid are organized into **a two-level hierarchy**
  - Each thread **block** is organized as a **three-dimensional array** of threads with a total size of up to 512 threads
  - The coordinates of threads in a block are uniquely defined by three thread indices: `threadIdx.x`, `threadIdx.y`, and `threadIdx.z`
  - **Not all applications** will use all **three dimensions** of a thread block



# blockIdx and threadIdx

- Threads in a grid are organized into **a two-level hierarchy**
- In Figure
  - each thread block is organized into a 4x2x2 three-dimensional array of threads
  - this gives Grid 1 a total of 4x16 = 64 threads
- Each thread uses indices to decide what data to work on
  - **blockIdx**: 1D, 2D, or 3D (CUDA 4.0)
  - **threadIdx**: 1D, 2D, or 3D



# CUDA Thread Organization

- When a thread executes the kernel function, references to the `blockIdx` and `threadIdx` variables return the **coordinates of the thread**
- Additional built-in variables, `gridDim` and `blockDim`, provide the **dimension of the grid** and the **dimension of each block**

- `threadID = blockIdx.x * blockDim.x + threadIdx` identifies the part of the input data to read from and the part of the output data structure to write to
  - **Example** Thread 3 of Block 0 has a threadID value of 0*M + 3 = 3
  - **Example** Thread 3 of Block 5 has a threadID value of 5*M + 3

# CUDA threads, blocks and grids

- Nvidia use the **Compute Capability** specification to encode what each generation of GPU chips is capable of
- The Compute Capability (CC) of a GPU can be discovered by running the `deviceQuery` utility

| | Compute Capability | | | |
|---|---|---|---|---|
| **Item** | **1.x** | **2.x** | **3.x** | **5.x** |
| Max. number of grid dimensions | 2 | 3 | | |
| Grid maximum x-dimension | $2^{16}-1$ | | $2^{31}-1$ | |
| Grid maximum y/z-dimension | $2^{16}-1$ | | | |
| Max. number of block dimensions | 3 | | | |
| Block max. x/y-dimension | 512 | 1024 | | |
| Block max. z-dimension | 64 | | | |
| Max. threads per block | 512 | 1024 | | |
| GPU example (GTX family chips) | 8800 | 480 | 780 | 980 |

# CUDA Thread Organization

- The exact organization of a grid is determined by the **execution configuration** provided at kernel launch
  - The first parameter specifies the dimensions of the grid as # blocks
  - The second specifies the dimensions of each block as # threads
  - Each such parameter is a **`dim3`** type, a C struct with three unsigned integer fields: x, y, and z
  - Example

```
dim3 dimGrid(128, 1, 1);
dim3 dimBlock(32, 1, 1);
vecAddKernel<<<dimGrid, dimBlock>>>(. . .);
```

or

```
dim3 cat(128, 1, 1);
dim3 dog(32, 1, 1);
KernelFunction<<<cat, dog>>>(. . .);
```

# Execution Configuration Examples

**Examples** Assuming we have

```
dim3 b(3,3,3);
dim3 g(20,100);
```

Different grid-block combinations are possible:

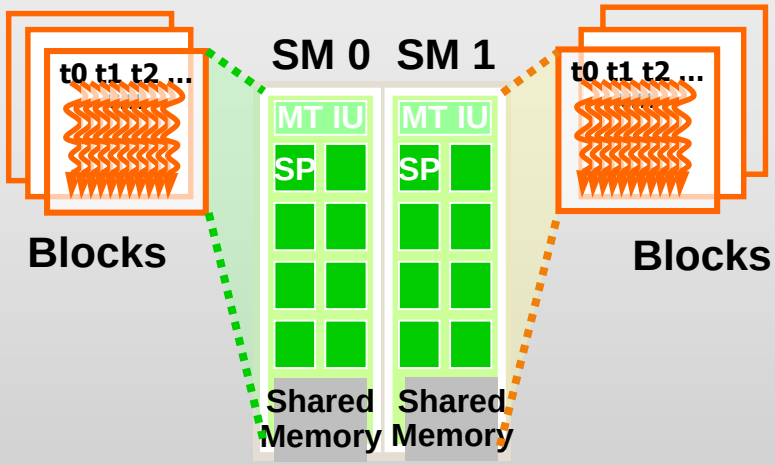
- `foo<<<g,b>>>();` // Run a 20x100 grid made of 3x3x3 blocks
- `foo<<<10,b>>>();` // Run a 10-block grid, each block made by 3x3x3 threads
- `foo<<<g,256>>>();` // Run a 20x100 grid, made of 256 threads
- `foo<<<g,2048>>>();` // An ***invalid*** example: maximum block size is 1024 threads even for compute capability 5.x
- `foo<<<5,g>>>();` // Another ***invalid*** example, that specifies a block size of 20x100=2000 threads
- `foo<<<10,256>>>;` // simplified configuration for a 1D grid of 1D blocks

# Synchronization

- CUDA allows threads in the same block to coordinate their activities using a barrier synchronization function, `__syncthreads()`
  - the thread that executes the function call will be held at the calling location until every thread in the block reaches the location
- A `__syncthreads()` statement must be executed by all threads in a block of the kernel before any moves on to the next phase

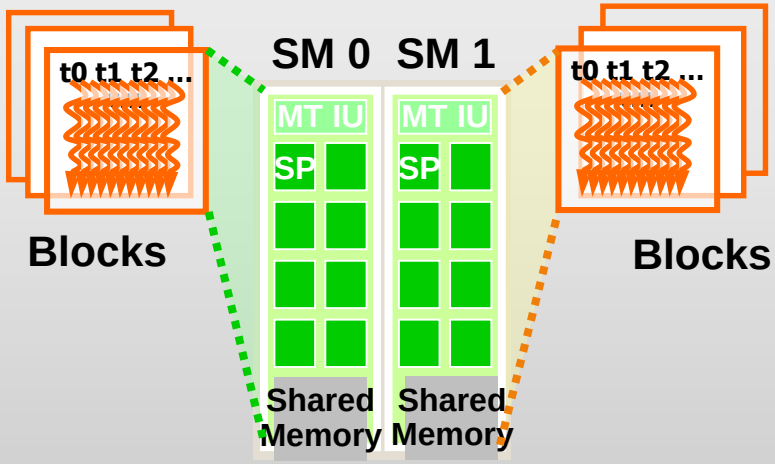
# Thread and Block Assignment

- Once a **kernel is launched**, the CUDA runtime system generates the corresponding **grid of threads**
  - threads are assigned to execution resources on a block-by-block basis
- The execution resources are organized into streaming multiprocessors (SMs)
- Each device has a limit on the number of block that can be assigned to each SM



# Thread and Block Assignment

- When an insufficient amount of any one or more types of resources needed for the simultaneous execution of blocks, the CUDA runtime automatically reduces the number of blocks assigned to each SM
- The runtime system maintains a list of blocks that need to execute and assigns new blocks to SMs as they complete the execution of blocks previously assigned to them

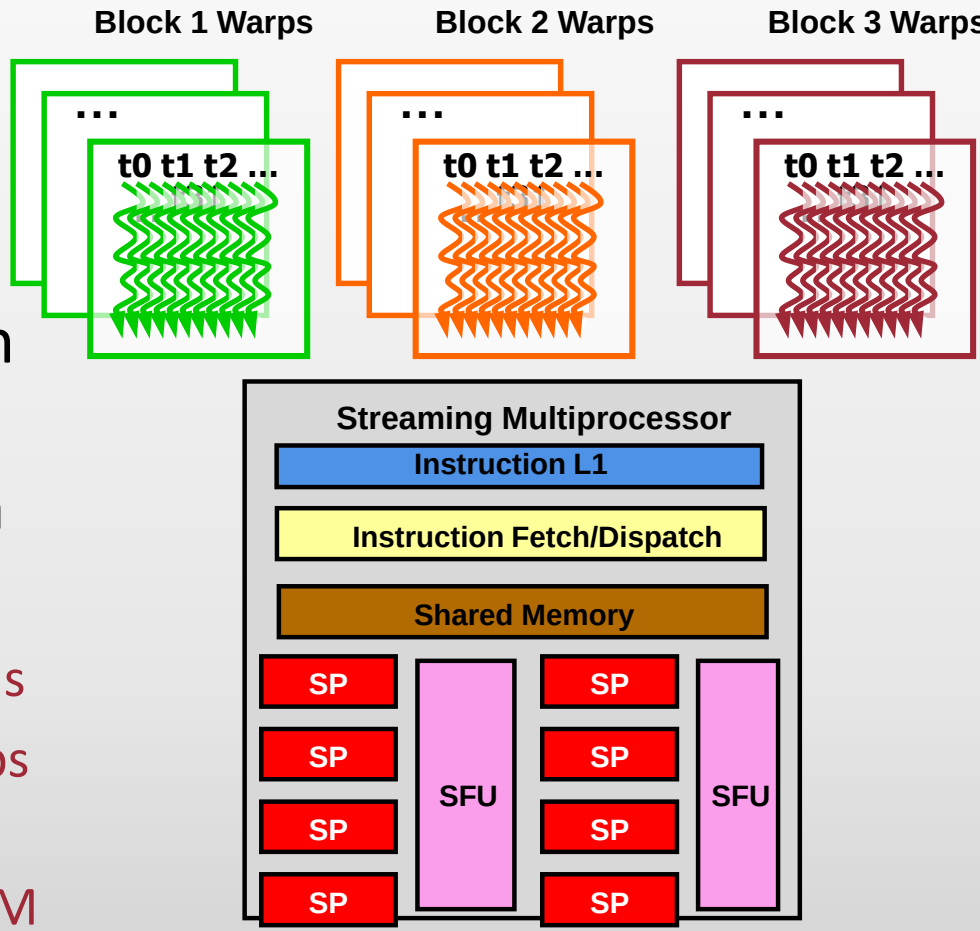


# Thread Scheduling

- Once a block is assigned to a streaming multiprocessor, it is further divided into 32-thread units called **warps**
- The warp is the **unit of thread scheduling** in SMs
- Each warp consists of 32 threads of consecutive threadIdx values:
  - Threads 0 through 31 form the first warp
  - Threads 32 through 63 the second warp, and so on
- We can calculate the number of warps that reside in an SM for a given block size and a given number of blocks assigned to each SM

# Thread Scheduling

- Each Block is executed as 32-thread Warps
  - Warps are scheduling units in SM
- **Example** If 3 blocks are assigned to an SM and each block has 256 threads, how many **warps** are there in an SM?
  - 3 blocks, each block 256 threads
  - each block has 256/32 = 8 warps
  - having 3 blocks in each SM, we have 8 x 3 = 24 warps in each SM



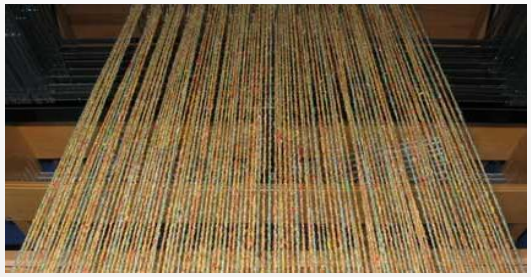
# Thread Scheduling

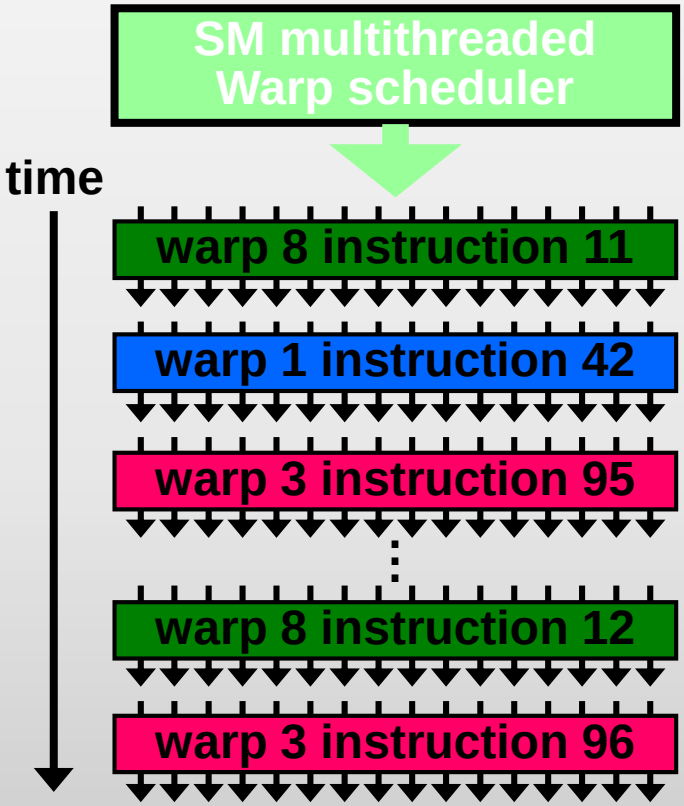
- **Why** do we need to have so many warps in an SM if there are only 8 SPs in an SM?
  - The answer is for **efficiently executing long-latency operations** such as global memory accesses
  - When an instruction executed by the threads in a warp needs to wait for the result of a previously initiated long-latency operation, the **warp is not selected for execution**
  - Another resident warp (that is no waiting for results) is selected for execution
  - If more than one warp is ready for execution, a **priority** mechanism is used to select one for execution
  - This mechanism of filling the latency of expensive operations with work from other threads is often referred to as ***latency hiding***

# Thread Scheduling

- Note that **warp scheduling** is also used for tolerating other types of long latency operations such as ***pipelined floating-point arithmetic*** and ***branch instructions***
- With enough warps around
  - The hardware will likely find a warp to execute at any point in time
  - Full use of the execution hardware in spite of long-latency operations
- The selection of ready warps for execution
  - Does not introduce any idle time into the execution timeline
  - ***zero-overhead thread scheduling***
- With warp scheduling, the long waiting time of warp instructions is hidden by executing instructions from other warps

# SM Warp Scheduling






- SM hardware implements zero-overhead Warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - **Eligible Warps** are selected for execution on a prioritized scheduling policy
  - **All threads in a Warp execute the same instruction** when selected
- 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 minimum of 13 Warps are needed to fully tolerate 200-cycle memory latency

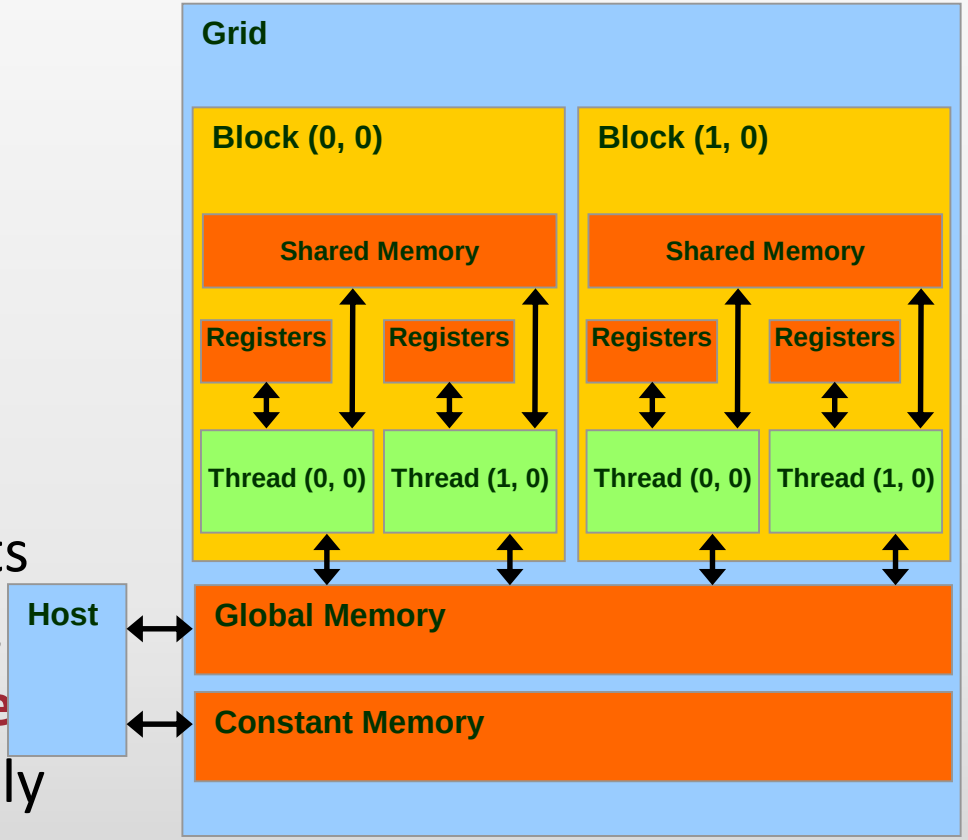
# Thread Scheduling

- List of GPU chips and their SM capabilty

| | compute capability | | | | | |
|---|---|---|---|---|---|---|
| **Item** | **1.0, 1.1** | **1.2, 1.3** | **2.x** | **3.0** | **3.5** | **5.0** |
| **Concurrent kernels/device** | 1 | | 16 | | 32 | |
| **Max. resident blocks/SM** | 8 | | | 16 | | 32 |
| **Max. resident warps/SM** | 24 | 32 | 48 | 64 | | |
| **Max. resident threads/SM** | 768 | 1024 | 1536 | 2048 | | |
| **32-bit registers/SM** | 8k | 16k | 32k | 64k | | |
| **Max. registers/thread** | 128 | | 63 | | 255 | |

# Programmer View of CUDA Memories

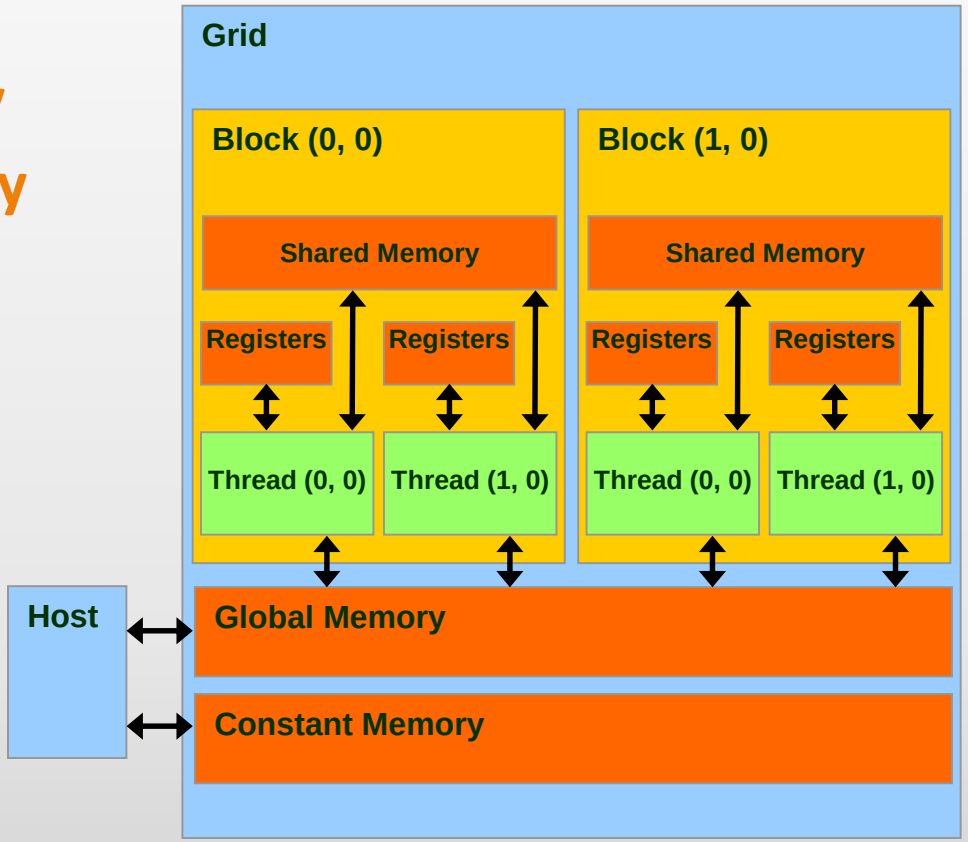
- At the bottom of the figure, we see global memory and constant memory
- These types of memory can be written (W) and read (R) by the host by calling API functions
- The **constant memory** supports short-latency, high-bandwidth, **read-only access by the device** when all threads simultaneously access the same location



# Programmer View of CUDA Memories

**Device** code can:

- R/W per-thread **registers**
- R/W per-thread **local memory**
- R/W per-block **shared memory**
- R/W per-grid **global memory**
- Read-only per-grid **constant memory**

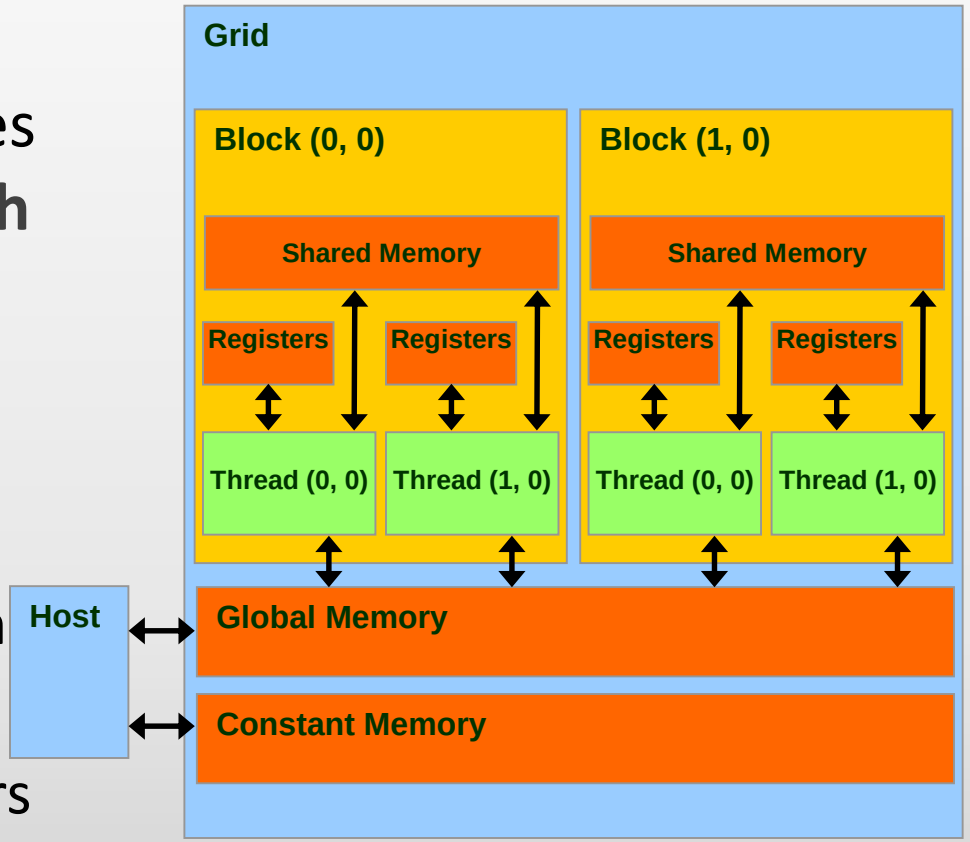



**Host** code can:

- Transfer data to/from per-grid global and constant memories

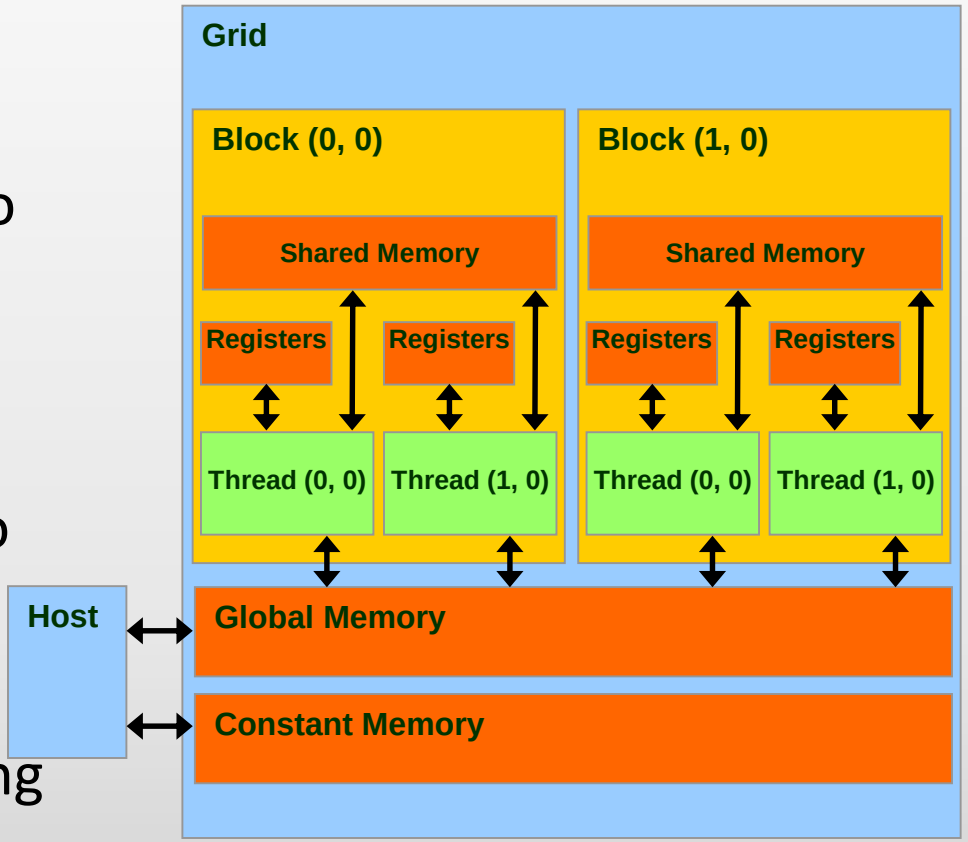
# Programmer View of CUDA Memories

- **Registers** and **shared memory** are on-chip memories
- Variables on these memories can be **accessed at very high speed** in a highly parallel manner
- **Registers** are allocated to individual threads and each thread can only access its own registers
- A kernel function uses registers to hold frequently accessed variables private to each thread



# Programmer View of CUDA Memories

- **Registers** and **shared memory** are on-chip memories
- **Shared memory** is allocated to thread blocks;
- **all threads in a block** can access variables in the **shared memory** locations allocated to the block
- Shared memory is used by threads to cooperate by sharing their input data and the intermediate results



# Variables

- Table presents the CUDA syntax for declaring program variables into the various types of device memory
- Each declaration gives to CUDA variable:
  - A **scope** identifies the range of threads that can access the variable: single thread only, all threads of a block, or all threads of all grids
  - A **lifetime** specifies the portion of the program's execution duration when the variable is available for use: either within a kernel's invocation or throughout the entire application

| Variable declaration | Memory | Scope | Lifetime |
|---|---|---|---|
| Automatic Variables | register | thread | kernel |
| `__device__ __shared__ int SharedVar;` | shared | block | kernel |
| `__device__ int GlobalVar;` | global | grid | application |
| `__device__ __constant__ int ConstantVar;` | constant | grid | application |