

**GPU**

**Computation**

**CSCI 4239/5239**

**Advanced Computer Graphics**

**Spring 2018**

# Solutions to Parallel Processing

- Message Passing (distributed)
  - MPI (library)
- Threads (shared memory)
  - pthreads (library)
  - OpenMP (compiler)
- GPU Programming (shared bus)
  - CUDA (compiler)
  - OpenCL (library)
  - OpenACC (compiler)
  - GLSL Compute Shader

# Using the GPU for Computation

- The GPU is very good at floating point. How can we use that to do computations?
  - Write a shader and be the result be a pseudo-color
  - Use CUDA with nVidia hardware
  - Use OpenCL with general hardware
  - Use an OpenGL 4.3 Compute Shader
- Issues
  - Getting instructions and data to the GPU
  - Precision of computations

# Text/Notes

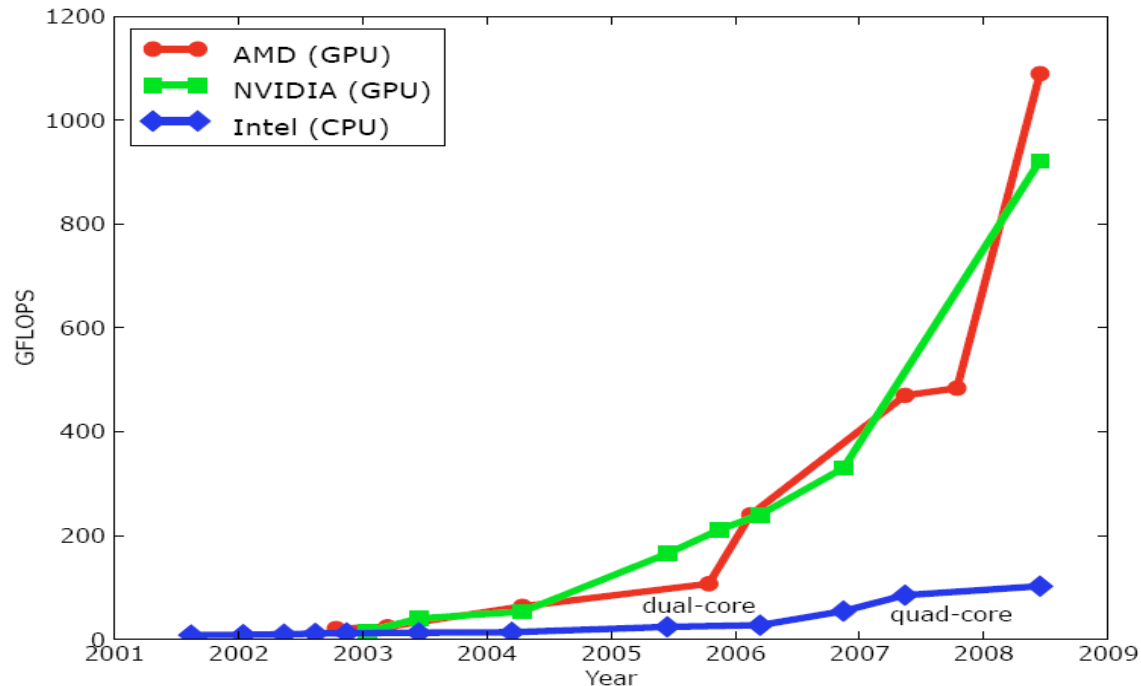
- Programming massively Parallel Processors
  - *Kirk and Hwu*
  - Good introduction to CUDA and OpenCL
  - Examples, tips and Tricks
  - Most slides taken from their lecture notes
- CUDA by Example
  - *Sanders and Kandrot*
  - CUDA only
  - Examples

# History of Coprocessors

- Floating point option
  - 8087, 80287, Weitek
- Floating Point Systems Array Processors
  - Attaches to VAX
- DSP chips
- Analog and special purpose CPUs
- Graphics Processors

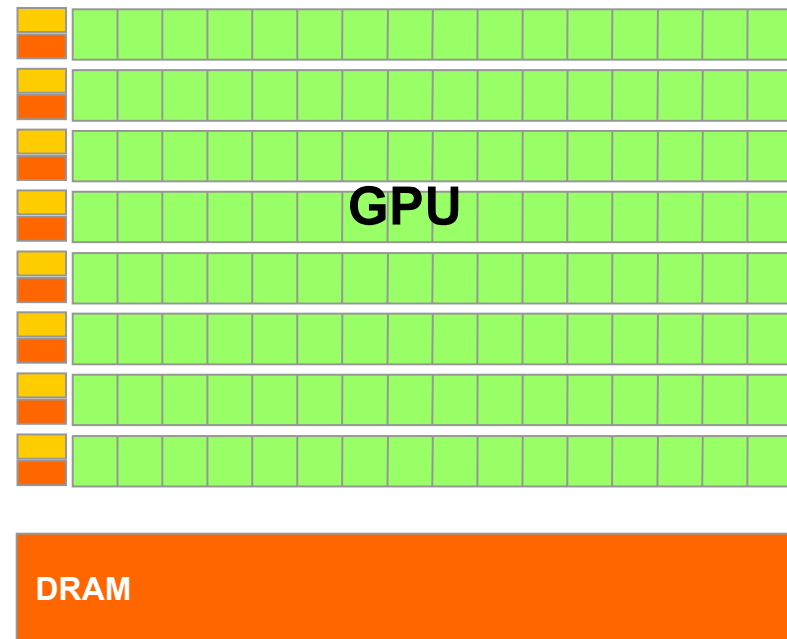
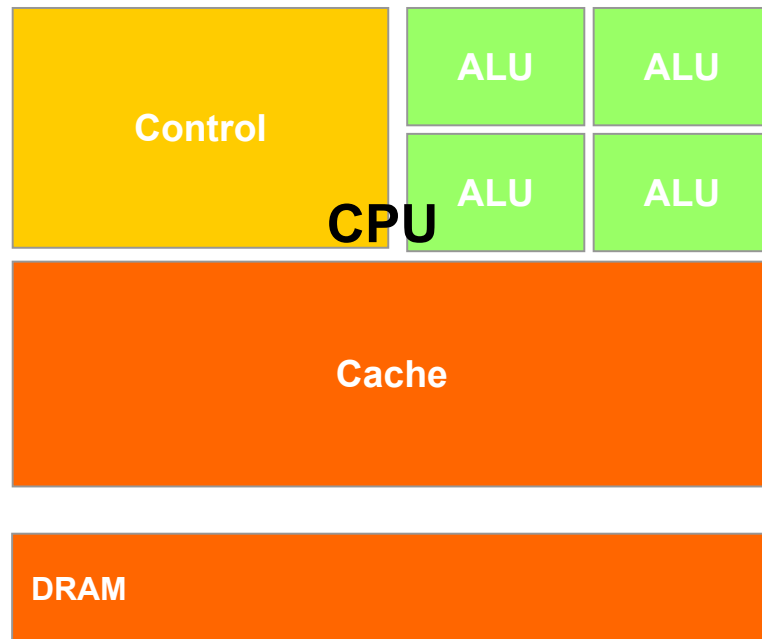
# Why Massively Parallel Processor

- A quiet revolution and potential build-up
  - Calculation: 367 GFLOPS vs. 32 GFLOPS
  - Memory Bandwidth: 86.4 GB/s vs. 8.4 GB/s
  - Until 2006, programmed through graphics API

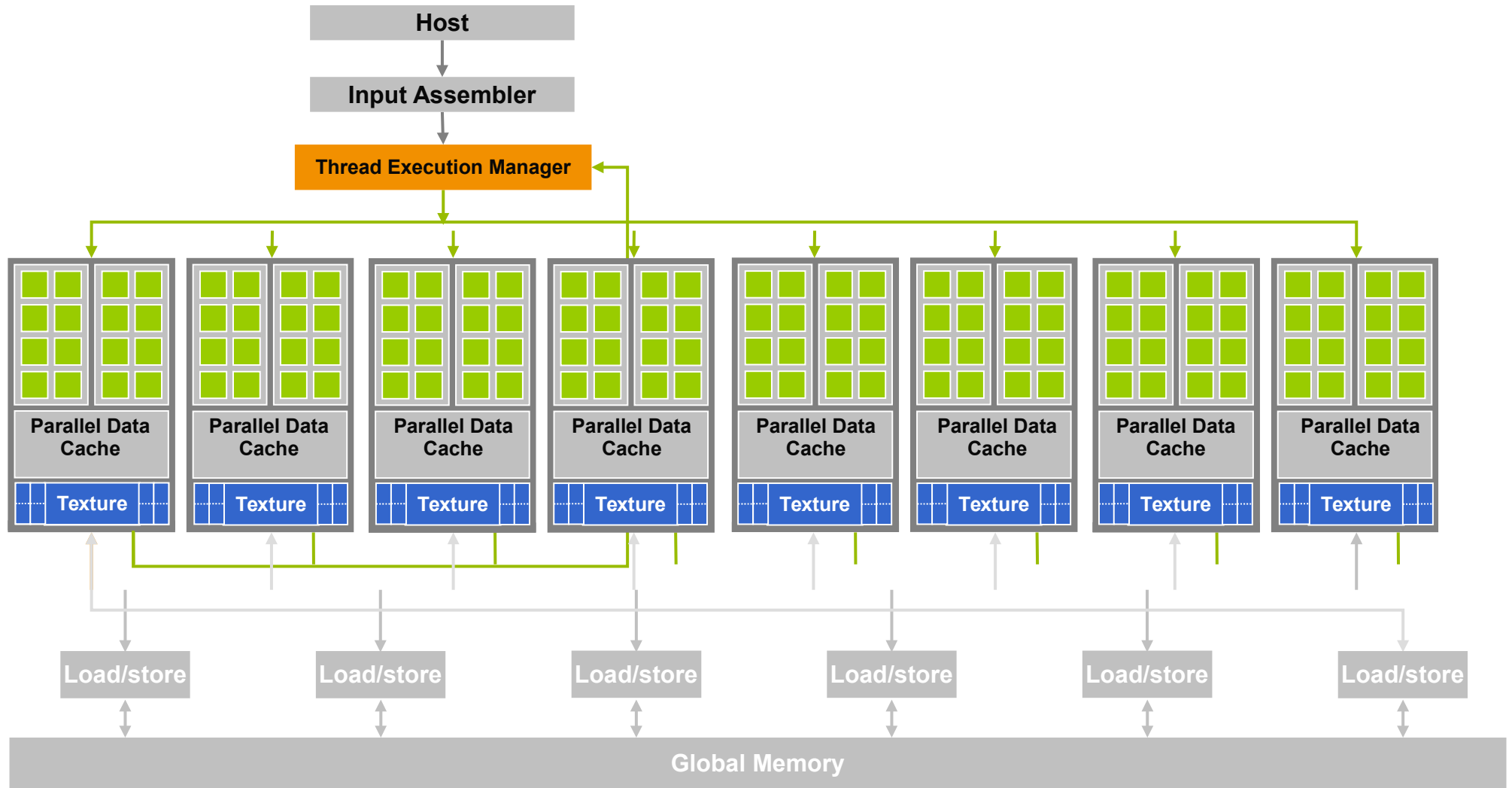


- GPU in every PC and workstation – massive volume and potential impact

# CPUs and GPUs have fundamentally different design philosophies



# Architecture of a CUDA-capable GPU



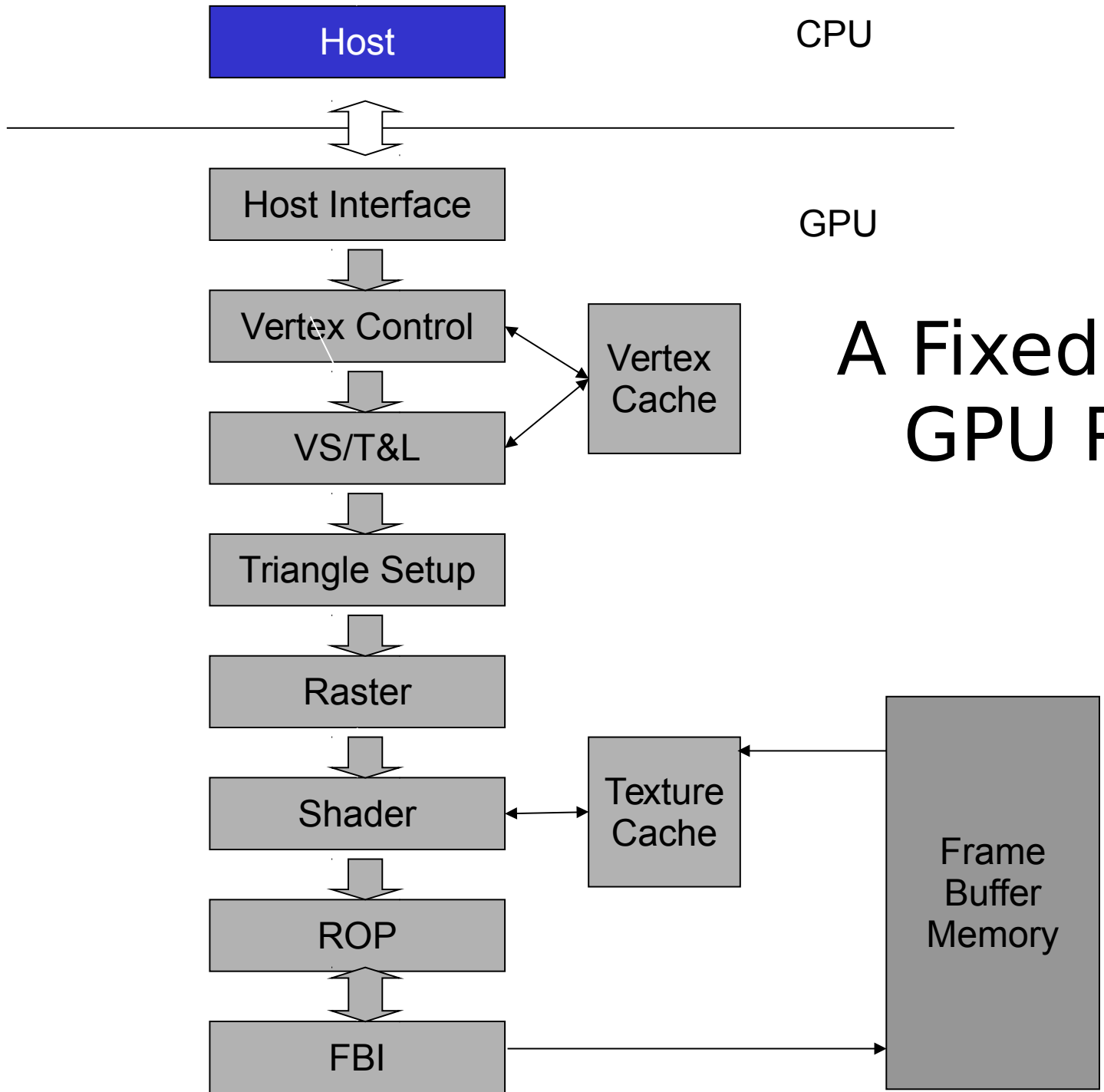


# GT200 Characteristics

- 1 TFLOPS peak performance (25-50 times of current high-end microprocessors)
- 265 GFLOPS sustained for apps such as VMD
- Massively parallel, 128 cores, 90W
- Massively threaded, sustains 1000s of threads per app
- 30-100 times speedup over high-end microprocessors on scientific and media applications: medical imaging, molecular dynamics

“I think they're right on the money, but the huge performance differential (currently 3 GPUs  $\approx$  300 SGI Altix Itanium2s) will invite close scrutiny so I have to be careful what I say publically until I triple check those numbers.”

-John Stone, VMD group, Physics UIUC

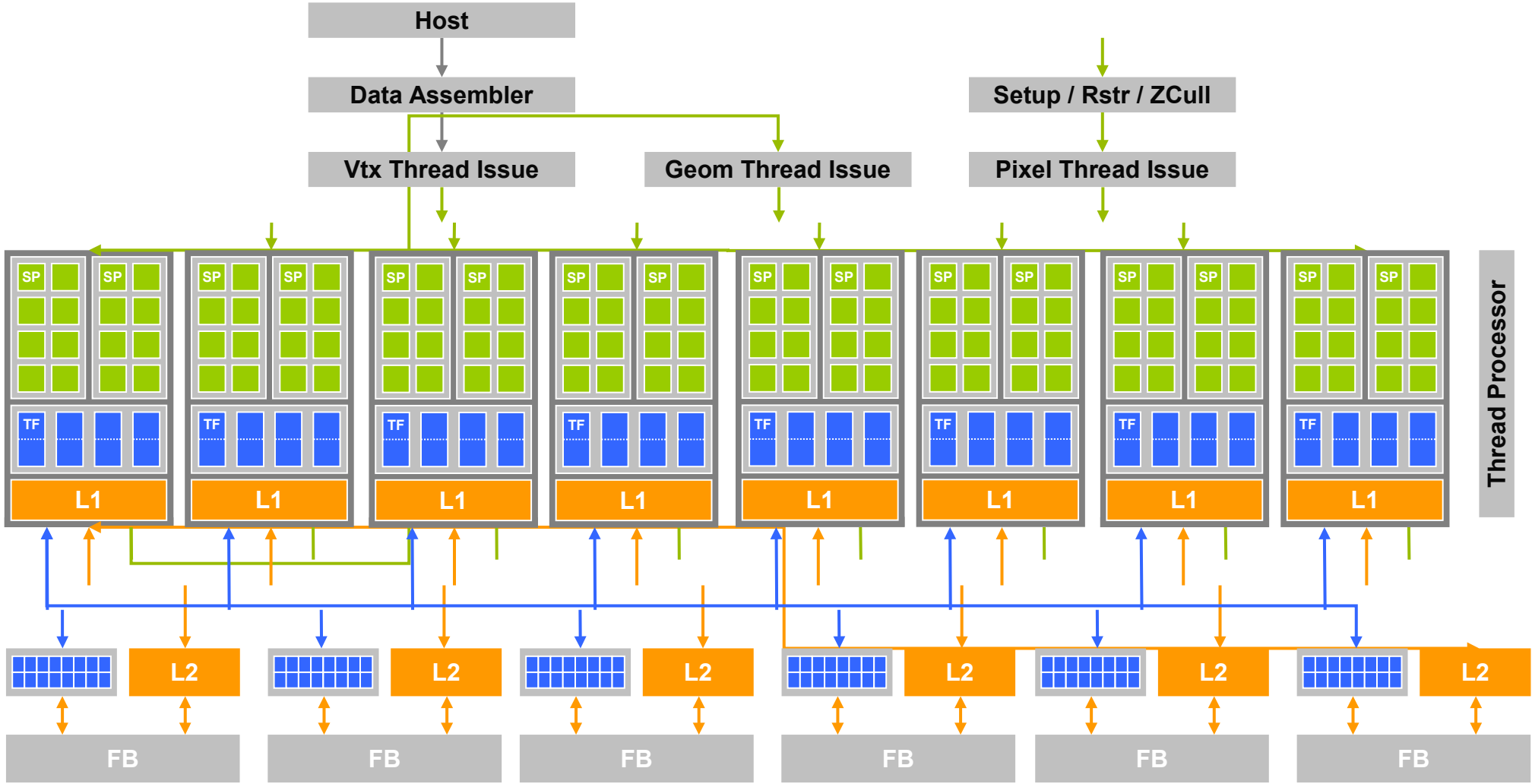


CPU

GPU

# A Fixed Function GPU Pipeline

# Unified Graphics Pipeline



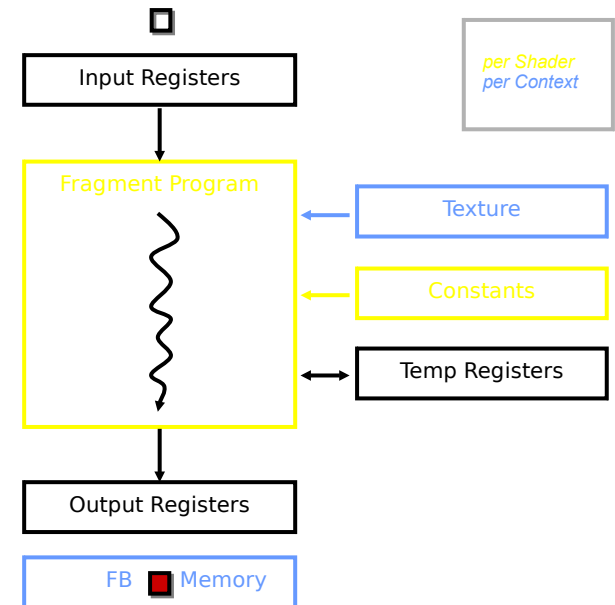
# What is (Historical) GPGPU ?

- General Purpose computation using GPU and graphics API in applications other than 3D graphics
  - GPU accelerates critical path of application
- Data parallel algorithms leverage GPU attributes
  - Large data arrays, streaming throughput
  - Fine-grain SIMD parallelism
  - Low-latency floating point (FP) computation
- Applications – see [GPGPU.org](http://GPGPU.org)
  - Game effects (FX) physics, image processing
  - Physical modeling, computational engineering, matrix algebra, convolution, correlation, sorting



# Previous GPGPU Constraints

- Dealing with graphics API
  - Working with the corner cases of the graphics API
- Addressing modes
  - Limited texture size/dimension
- Shader capabilities
  - Limited outputs
- Instruction sets
  - Lack of Integer & bit ops
- Communication limited
  - Between pixels
  - Scatter  $a[i] = p$



# Compute Shaders

- Shader buffers for memory access
- Shader has access to entire array for both read and write
- Compute shader compiled using OpenGL
- Requires OpenGL 4.3

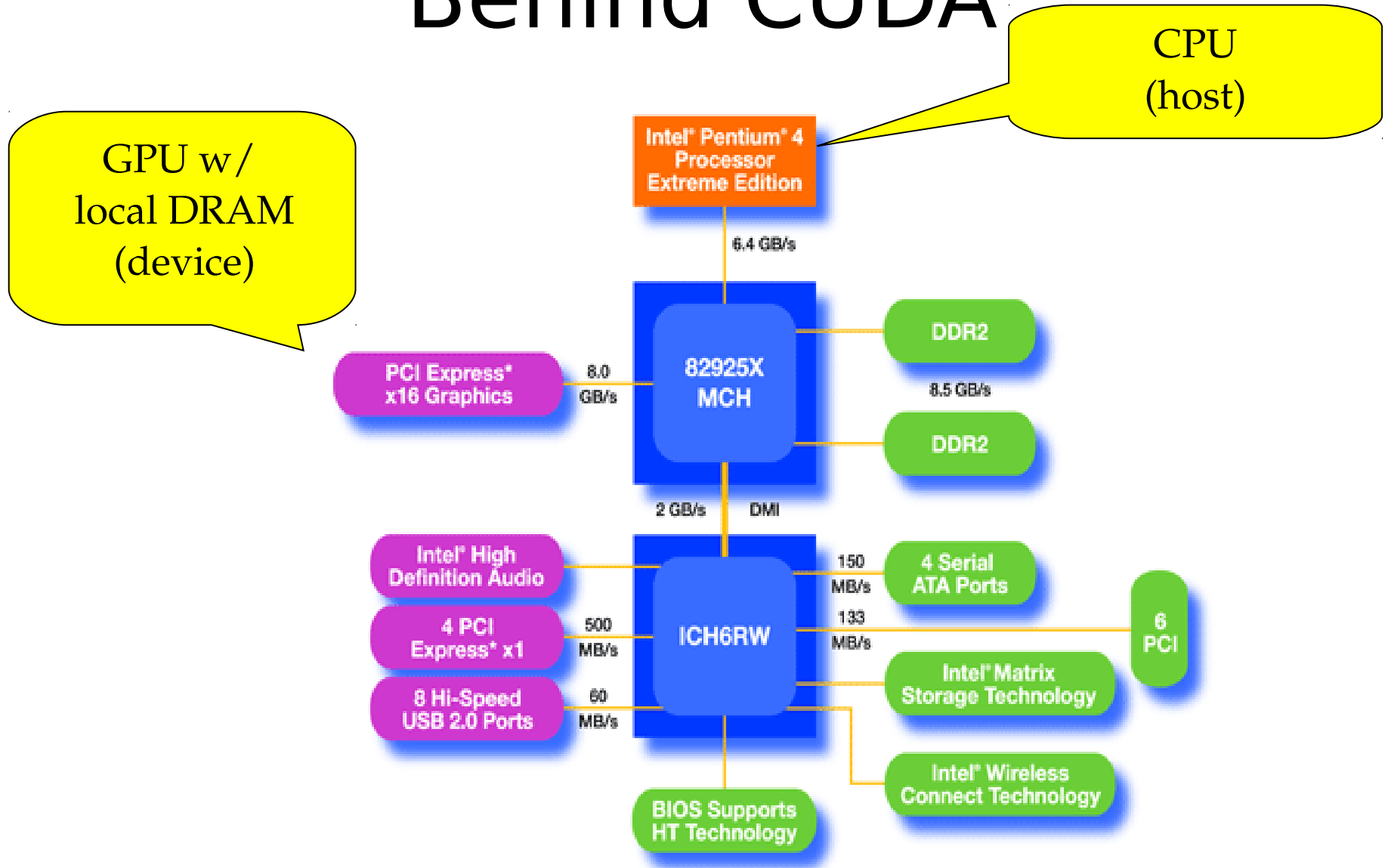


# CUDA

- “Compute Unified Device Architecture”
- General purpose programming model
  - User kicks off batches of threads on the GPU
  - GPU = dedicated super-threaded, massively data parallel co-processor
- Targeted software stack
  - Compute oriented drivers, language, and tools
- Driver for loading computation programs into GPU
  - Standalone Driver - Optimized for computation
  - Interface designed for compute – graphics-free API
  - Data sharing with OpenGL buffer objects
  - Guaranteed maximum download & readback speeds
  - Explicit GPU memory management



# An Example of Physical Reality Behind CUDA





# Parallel Computing on a GPU

- 8-series GPUs deliver 25 to 200+ GFLOPS on compiled parallel C applications
  - Available in laptops, desktops, and clusters
- GPU parallelism is doubling every year
- Programming model scales transparently
- Programmable in C with CUDA tools
- Multithreaded SPMD model uses application data parallelism and thread parallelism



**GeForce 8800**



**Tesla D870**



**Tesla S870**

# Overview

- CUDA programming model – basic concepts and data types
- CUDA application programming interface – basic
- Simple examples to illustrate basic concepts and functionalities
- Performance features will be covered later

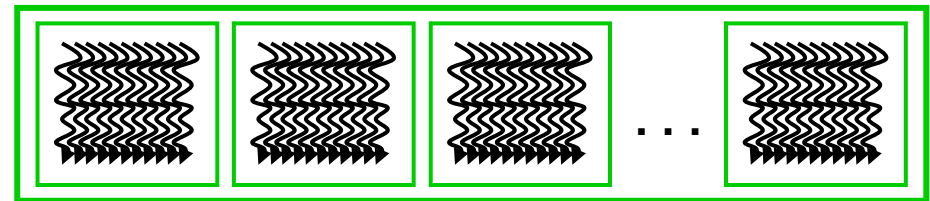
# CUDA - C with no shader limitations!

- Integrated host+device app C program
  - Serial or modestly parallel parts in **host** C code
  - Highly parallel parts in **device** SPMD kernel C code

Serial Code (host)

Parallel Kernel (device)

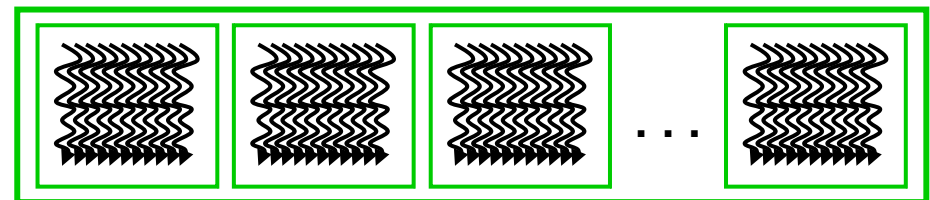
```
KernelA<<< nBlk, nTid >>>(args);
```



Serial Code (host)

Parallel Kernel (device)

```
KernelB<<< nBlk, nTid >>>(args);
```

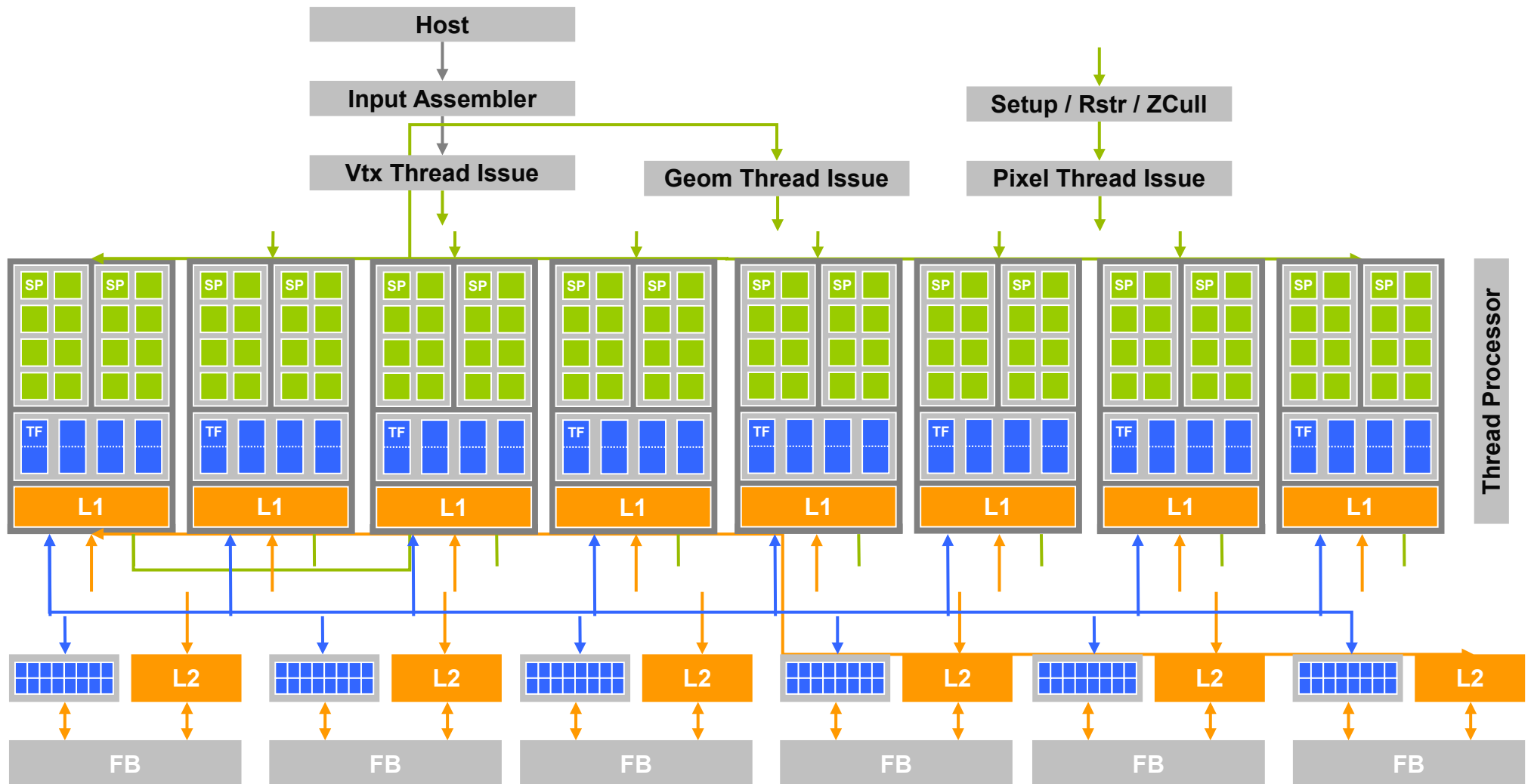


# CUDA Devices and Threads

- A compute **device**
  - Is a coprocessor to the CPU or **host**
  - Has its own DRAM (**device memory**)
  - Runs many **threads in parallel**
  - Is typically a **GPU** but can also be another type of parallel processing device
- Data-parallel portions of an application are expressed as device **kernels** which run 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

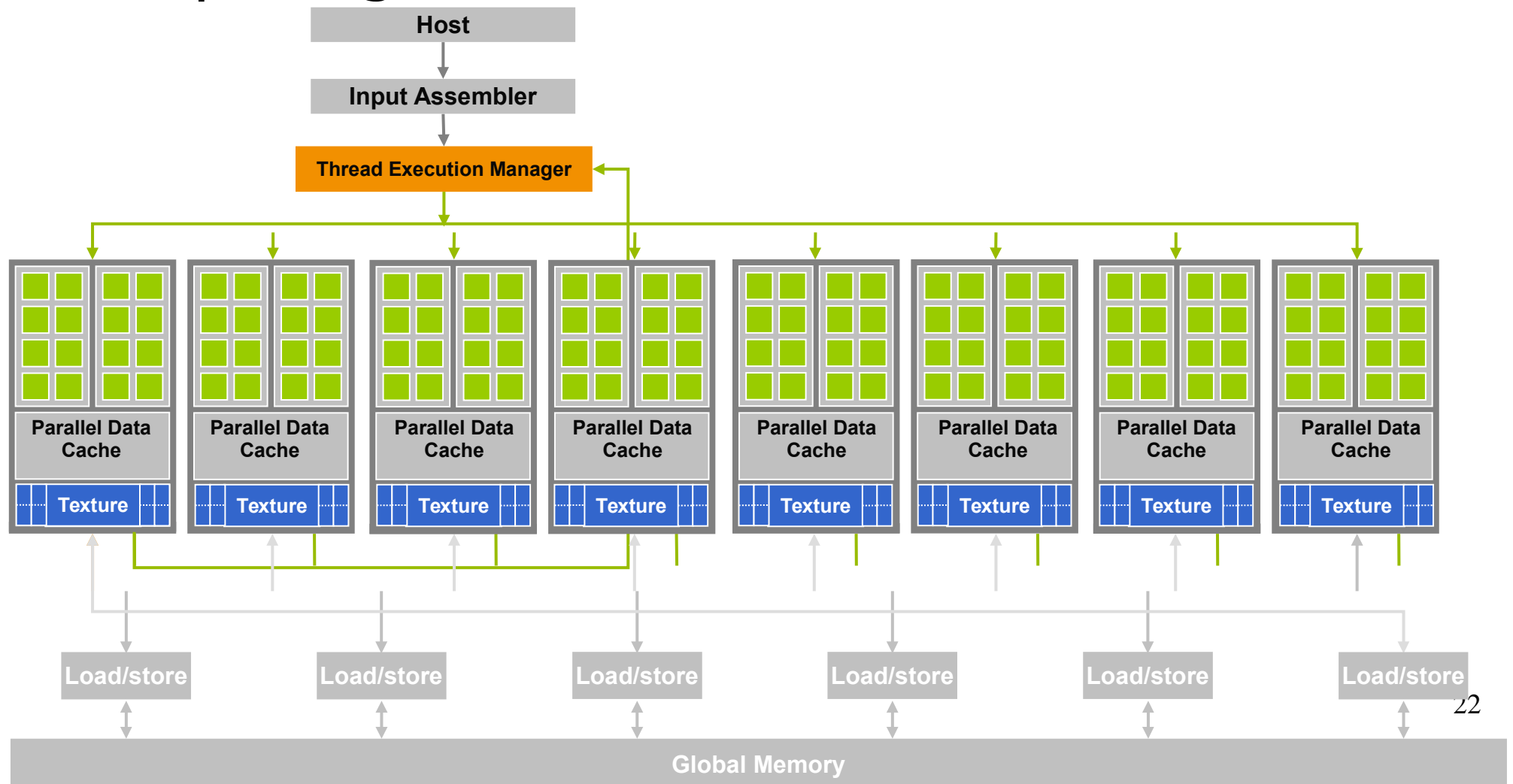
# G80 - Graphics Mode

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



# G80 CUDA mode - A **Device** Example

- Processors execute computing threads
- New operating mode/HW interface for computing



# Extended C

- **Declspecs**

- **global, device, shared, local, constant**

```
__device__ float filter[N];  
__global__ void convolve (float *image) {  
    __shared__ float region[M];  
    ...  
  
    region[threadIdx] = image[i];  
  
    __syncthreads()  
    ...  
  
    image[j] = result;  
}
```

- **Keywords**

- **threadIdx, blockIdx**

- **Intrinsics**

- **\_\_syncthreads**

- **Runtime API**

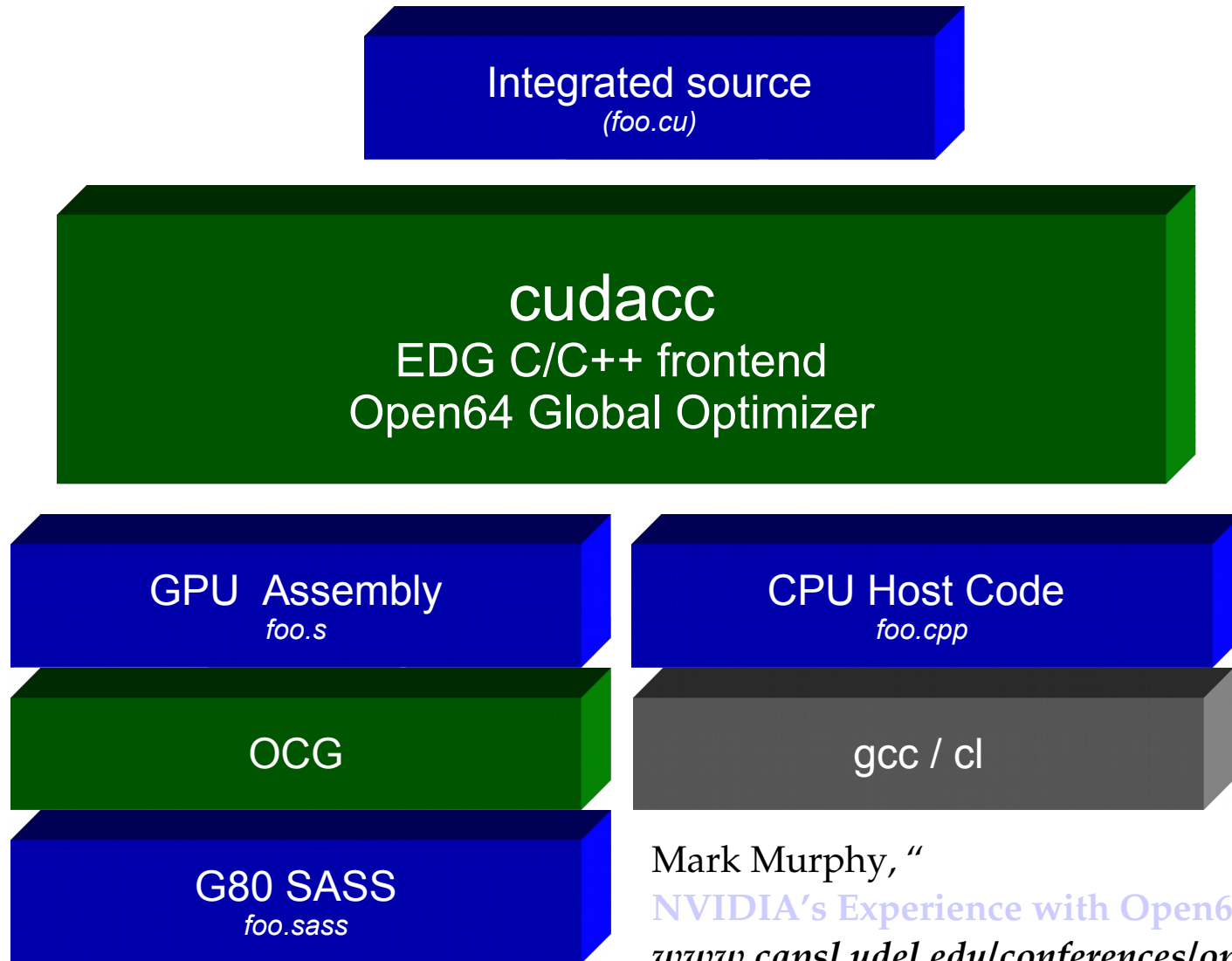
- **Memory, symbol, execution management**

```
// Allocate GPU memory  
void *myimage = cudaMalloc(bytes)
```

```
// 100 blocks, 10 threads per block  
convolve<<<100, 10>>> (myimage);
```

- **Function launch**

# Extended C

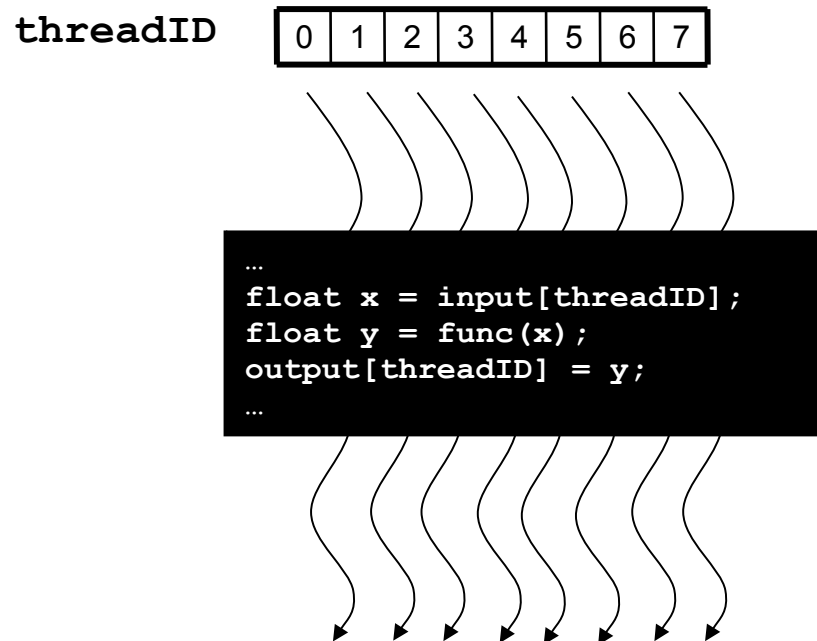


Mark Murphy, "  
*NVIDIA's Experience with Open64,*"  
*[www.capsl.udel.edu/conferences/open64/2008/  
Papers/101.doc](http://www.capsl.udel.edu/conferences/open64/2008/Papers/101.doc)*



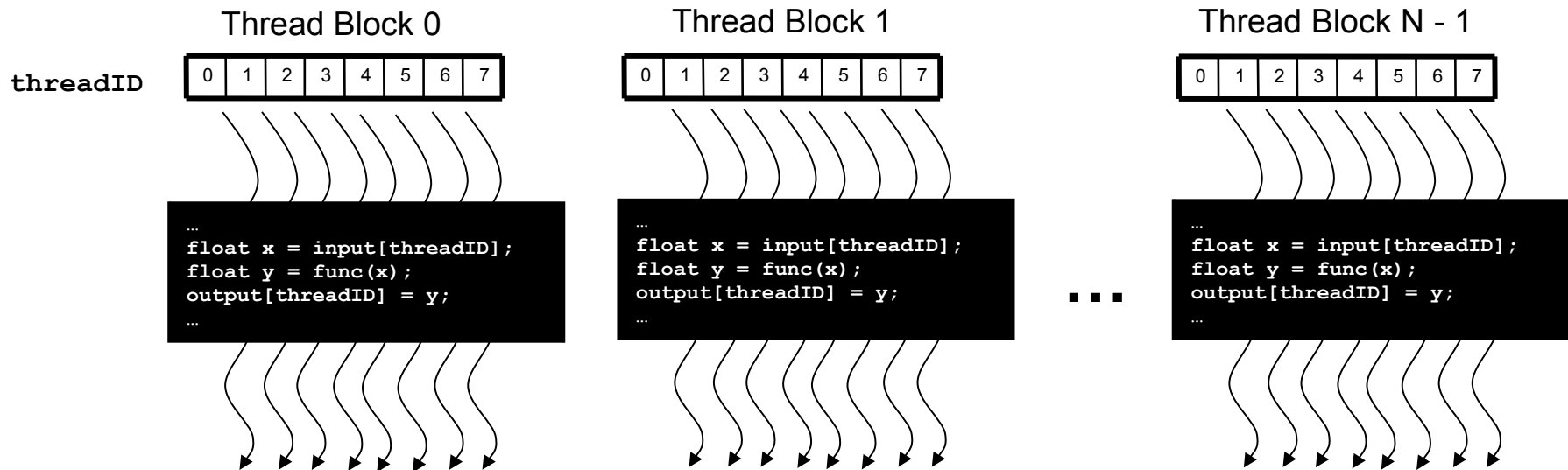
# Arrays of Parallel Threads

- A CUDA kernel is executed by an array of threads
  - All threads run the same code (SPMD)
  - Each thread has an ID that it uses to compute memory addresses and make control decisions



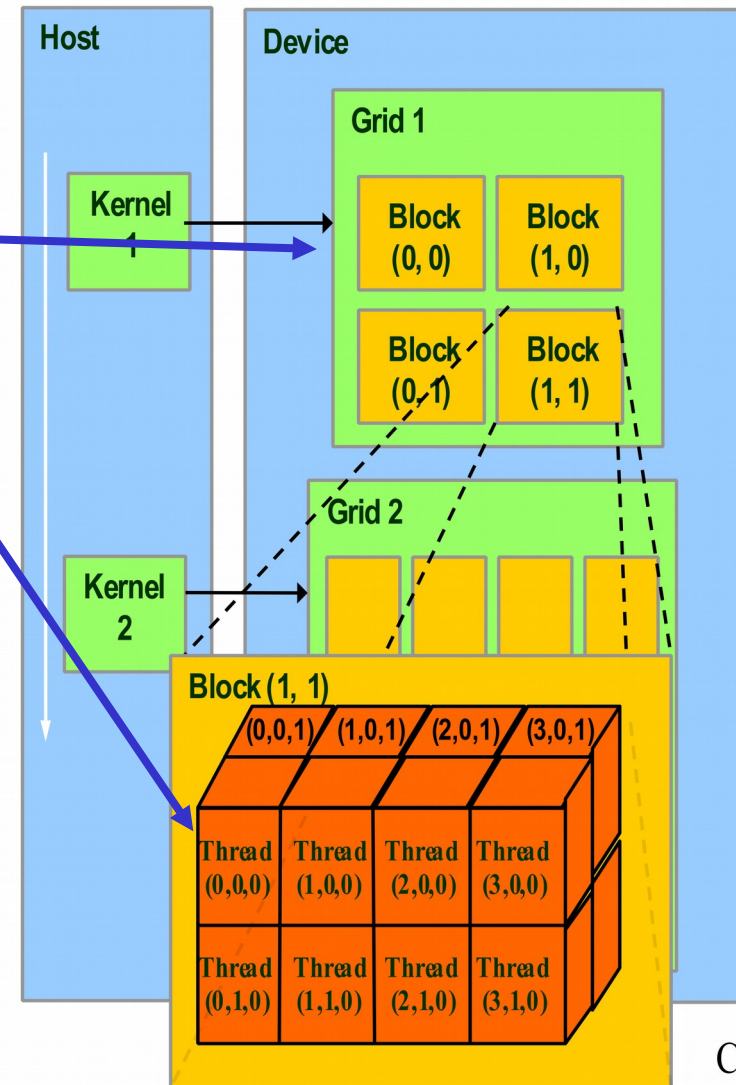
# Thread Blocks: Scalable Cooperation

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



# Block IDs and Thread IDs

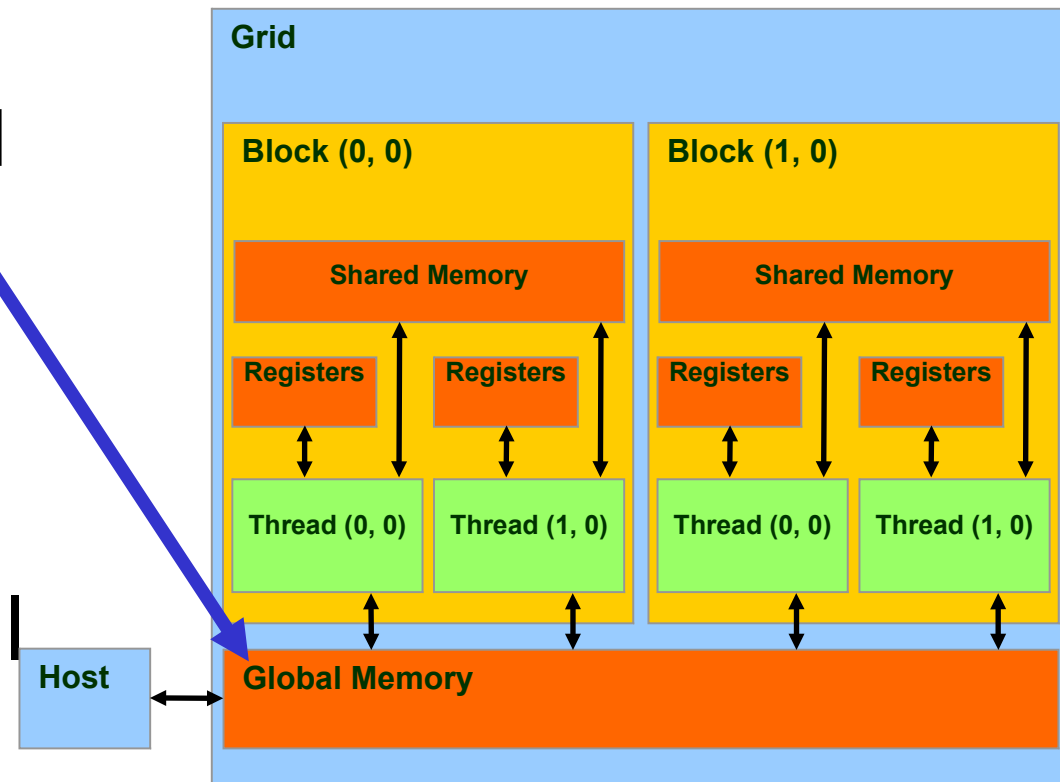
- Each thread uses IDs to decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - ...



Courtesy: NDVIA

# CUDA Memory Model Overview

- Global memory
  - Main means of communicating R/W Data between **host** and **device**
  - Contents visible to all threads
  - Long latency access
- We will focus on global memory for now
  - Constant and texture memory will come later

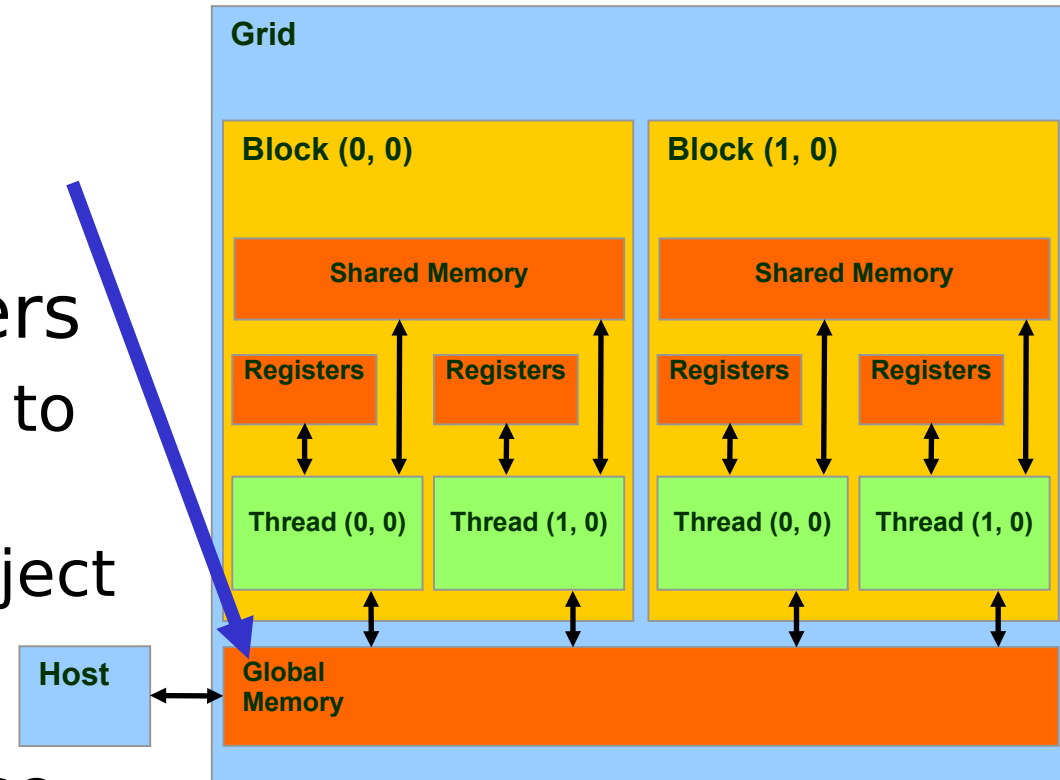


# CUDA API Highlights: Easy and Lightweight

- The API is an **extension to the ANSI C programming language**
  - ➔ Low learning curve
- The hardware is **designed to enable lightweight runtime and driver**
  - ➔ High performance

# CUDA Device Memory Allocation

- `cudaMalloc()`
  - Allocates object in the device Global Memory
  - Requires two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** allocated object
- `cudaFree()`
  - Frees object from device Global Memory
    - Pointer to freed object



# CUDA Device Memory Allocation (cont.)

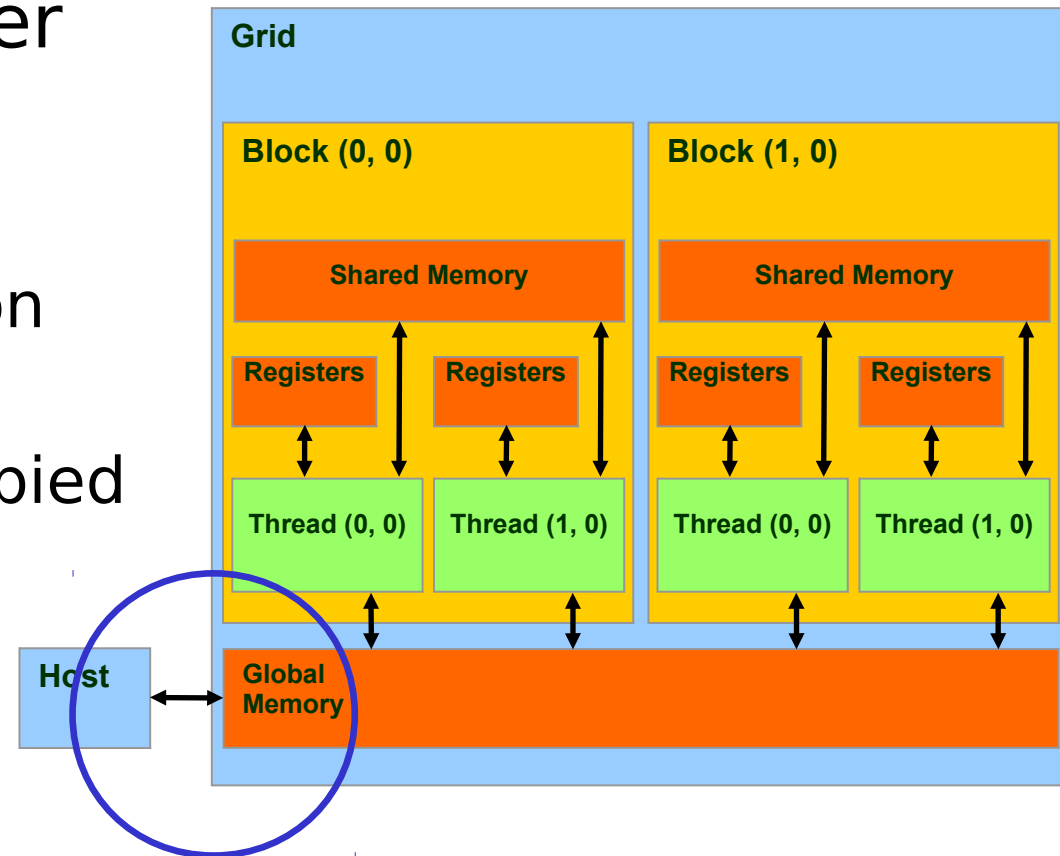
- Code example:
  - Allocate a  $64 * 64$  single precision float array
  - Attach the allocated storage to Md
  - “d” is often used to indicate a device data structure

```
TILE_WIDTH = 64;  
Float* Md  
int size = TILE_WIDTH * TILE_WIDTH * sizeof(float);
```

```
cudaMalloc((void**)&Md, size);  
cudaFree(Md);
```

# CUDA Host-Device Data Transfer

- `cudaMemcpy()`
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device
- Asynchronous transfer





# CUDA Host-Device Data Transfer (cont.)

- Code example:
  - Transfer a  $64 * 64$  single precision float array
  - M is in host memory and Md is in device memory
  - `cudaMemcpyHostToDevice` and `cudaMemcpyDeviceToHost` are symbolic constants

```
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
```

```
cudaMemcpy(M, Md, size, cudaMemcpyDeviceToHost);
```

# CUDA Function Declarations

	Executed on the:	Only callable from the:
<code>__device__ float DeviceFunc()</code>	device	device
<code>__global__ void KernelFunc()</code>	device	host
<code>__host__ float HostFunc()</code>	host	host

- `__global__` defines a kernel function
  - Must return `void`
- `__device__` and `__host__` can be used together

# CUDA Function Declarations (cont.)

- \_\_device\_\_ functions cannot have their address taken
- For functions executed on the device:
  - No recursion
  - No static variable declarations inside the function
  - No variable number of arguments

# Calling a Kernel Function - Thread Creation

- A kernel function must be called with an **execution configuration**:

```
__global__ void KernelFunc(...);  
dim3      DimGrid(100, 50);      // 5000 thread blocks  
dim3      DimBlock(4, 8, 8);     // 256 threads per block  
size_t    SharedMemBytes = 64;  // 64 bytes of shared memory  
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>(...);
```

- Any call to a kernel function is asynchronous from CUDA 1.0 on, explicit synch needed for blocking

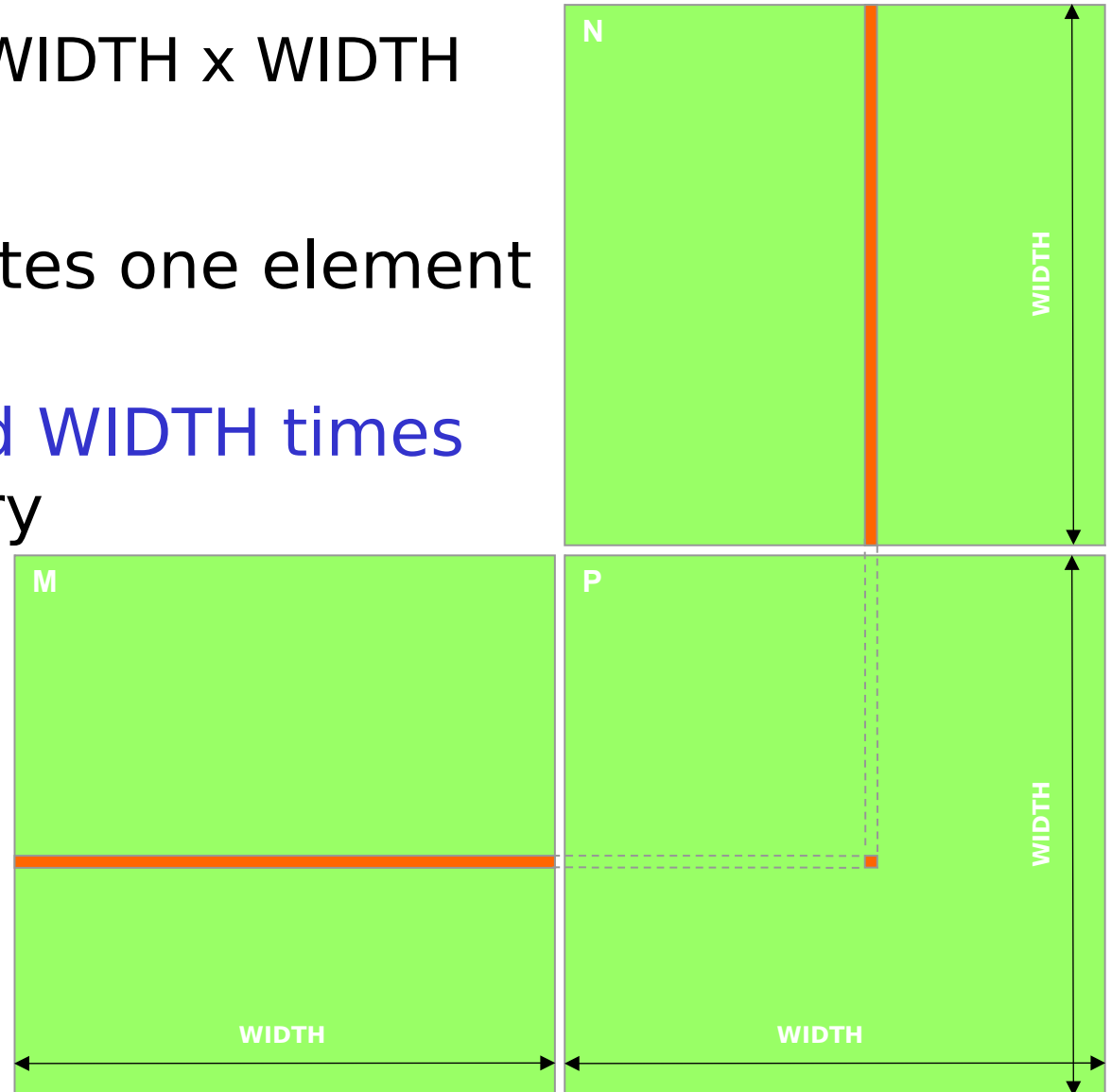
# A Simple Running Example

## Matrix Multiplication

- A simple matrix multiplication example that illustrates the basic features of memory and thread management in CUDA programs
  - Leave shared memory usage until later
  - Local, register usage
  - Thread ID usage
  - Memory data transfer API between host and device
  - Assume square matrix for simplicity

# Programming Model: Square Matrix Multiplication Example

- $P = M * N$  of size  $WIDTH \times WIDTH$
- Without tiling:
  - One **thread** calculates one element of  $P$
  - $M$  and  $N$  are loaded  $WIDTH$  times from global memory



# Memory Layout of a Matrix in C

$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	$M_{3,0}$
$M_{0,1}$	$M_{1,1}$	$M_{2,1}$	$M_{3,1}$
$M_{0,2}$	$M_{1,2}$	$M_{2,2}$	$M_{3,2}$
$M_{0,3}$	$M_{1,3}$	$M_{2,3}$	$M_{3,3}$

M



$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	$M_{3,0}$	$M_{0,1}$	$M_{1,1}$	$M_{2,1}$	$M_{3,1}$	$M_{0,2}$	$M_{1,2}$	$M_{2,2}$	$M_{3,2}$	$M_{0,3}$	$M_{1,3}$	$M_{2,3}$	$M_{3,3}$
-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------

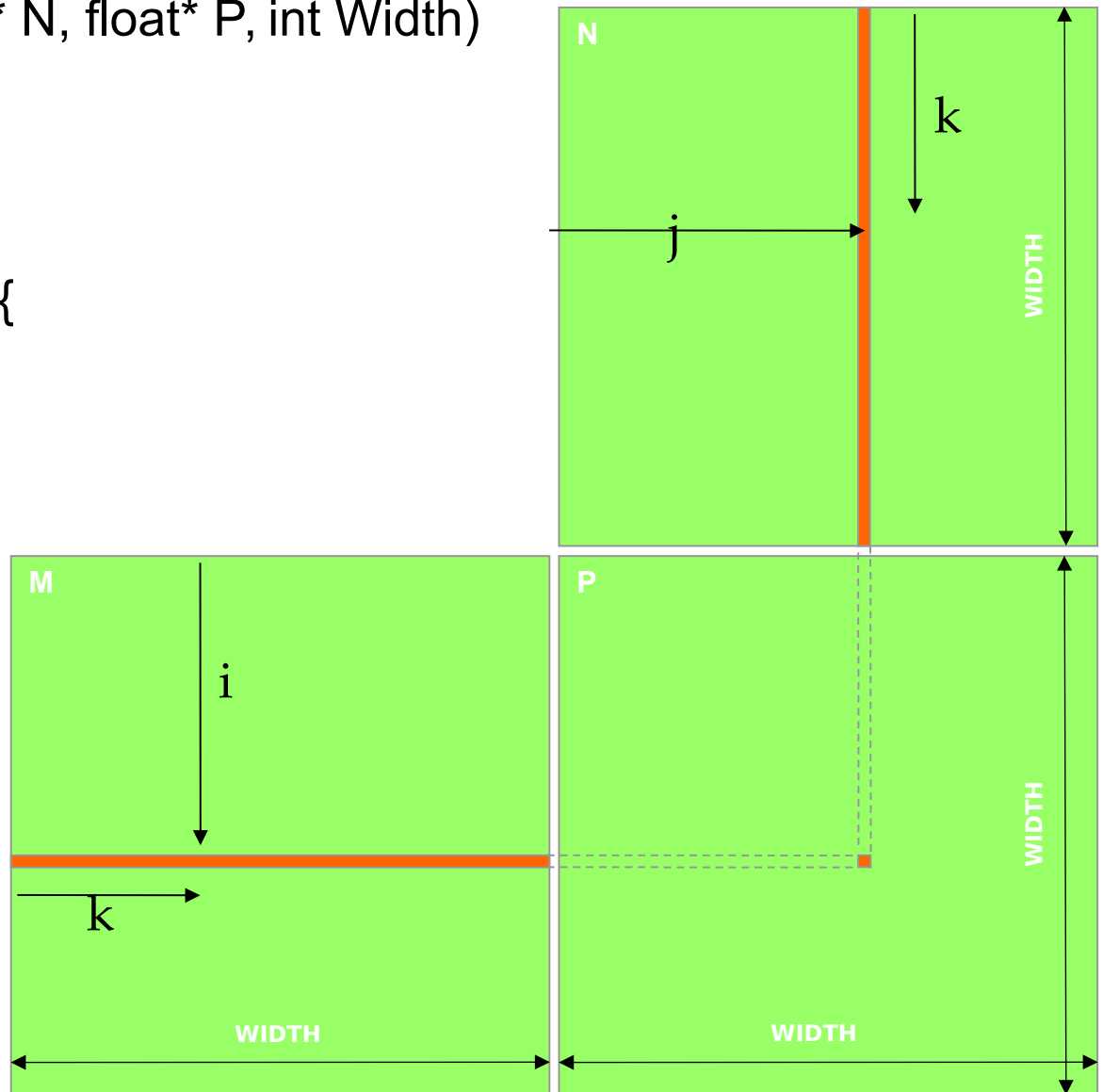
# Matrix Multiplication

## A Simple Host Version in C

```
// Matrix multiplication on the (CPU) host in double precision
```

```
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
```

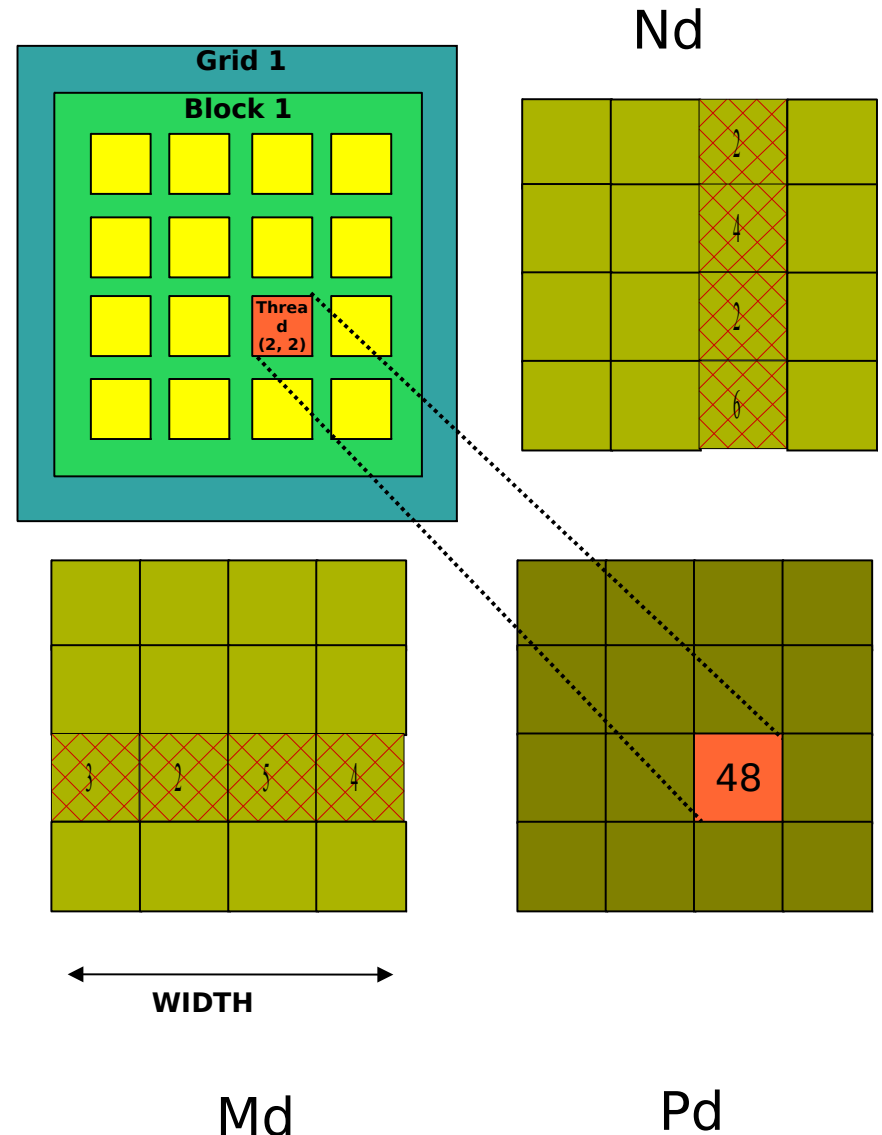
```
{  
  for (int i = 0; i < Width; ++i)  
    for (int j = 0; j < Width; ++j) {  
      double sum = 0;  
      for (int k = 0; k < Width; ++k) {  
        double a = M[i * width + k];  
        double b = N[k * width + j];  
        sum += a * b;  
      }  
      P[i * Width + j] = sum;  
    }  
}
```





# Threads and Blocks

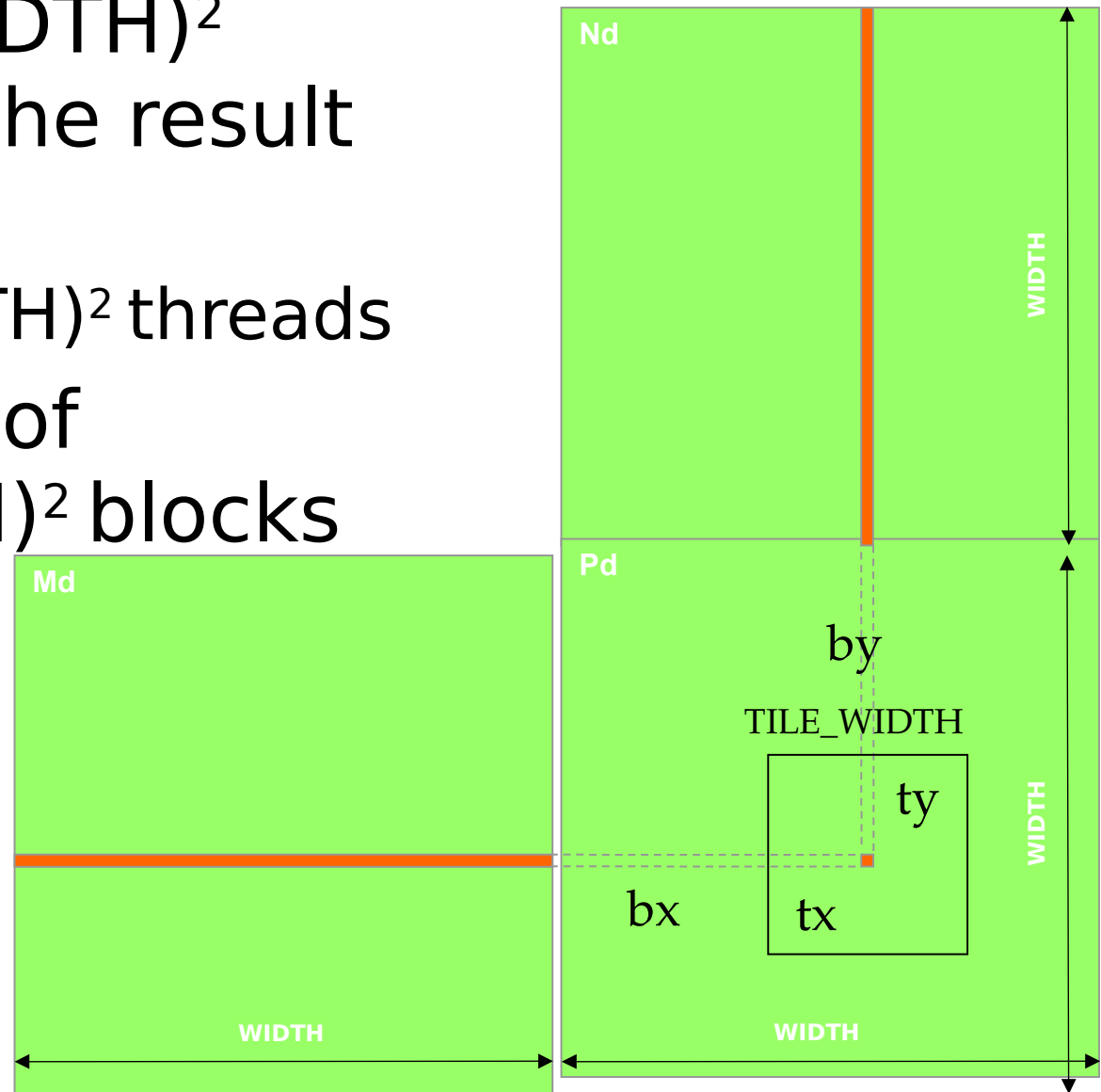
- One Block of threads compute matrix Pd
  - Each thread computes one element of Pd
- Each thread
  - Loads a row of matrix Md
  - Loads a column of matrix Nd
  - Perform one multiply and addition for each pair of Md and Nd elements
  - Compute to off-chip memory access ratio close to 1:1 (not very high)
- Size of matrix limited by the number of threads allowed in a thread block



# Tiled Kernel Function

- Have each 2D thread block to compute a  $(\text{TILE\_WIDTH})^2$  sub-matrix (tile) of the result matrix
  - Each has  $(\text{TILE\_WIDTH})^2$  threads
- Generate a 2D Grid of  $(\text{WIDTH}/\text{TILE\_WIDTH})^2$  blocks

You still need to put a loop around the kernel call for cases where  $\text{WIDTH}/\text{TILE\_WIDTH}$  is greater than max grid size (64K)!



# Kernel Function Code

```
// Matrix multiplication kernel
```

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // Index of thread
    unsigned int j = blockIdx.x*blockDim.x+threadIdx.x;
    unsigned int i = blockIdx.y*blockDim.y+threadIdx.y;

    // Calculate element value
    float sum = 0;
    for (int k=0;k<n;k++)
        sum += A[i*n+k] * B[k*n+j];

    // Store element value
    C[i*n+j] = sum;
}
```

# Step 1: Copy Input Data

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Bw, int Bn)
{
    int Width = Wb*Bn;
    int size = Width * Width * sizeof(float);
    float* Md, Nd, Pd;

    // Allocate and Load M, N to device memory
    cudaMalloc(&Md, size);
    cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
    cudaMalloc(&Nd, size);
    cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);
    // Allocate P on the device
    cudaMalloc(&Pd, size);
```

# Step 2: Kernel Invocation

```
// Setup the execution configuration
```

```
dim3 dimGrid(Bw, Bw);
```

```
dim3 dimBlock(Bn, Bn);
```

```
// Launch the device computation threads!
```

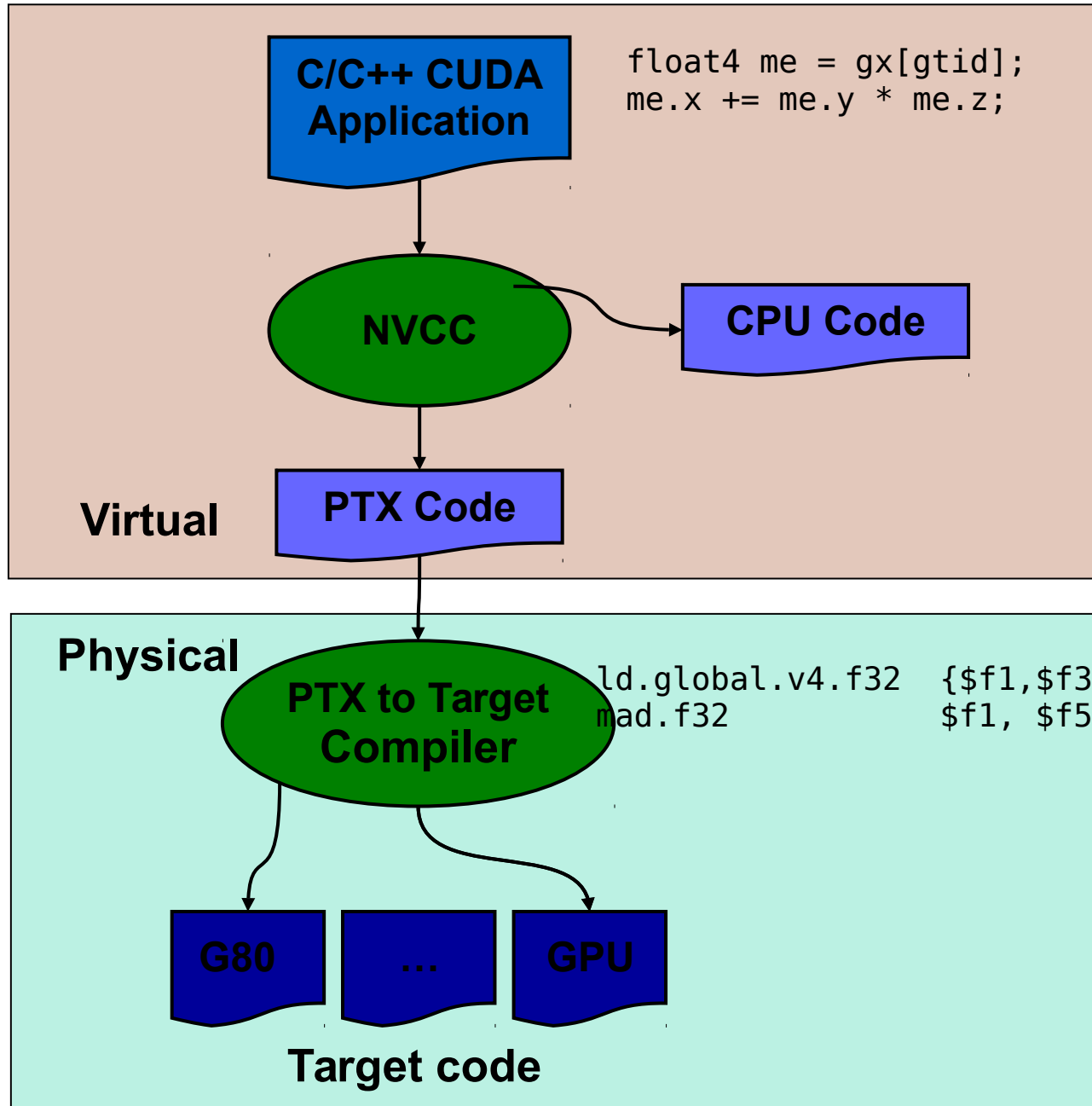
```
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Bw*Bn);
```

# Step 3: Copy Output Data

```
// Read P from the device
cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);

// Free device matrices
cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
}
```

# Compiling a CUDA Program



- Parallel Thread eXecution (PTX)
  - Virtual Machine and ISA
  - Programming model
  - Execution resources and state

# Compilation

- Any source file containing CUDA language extensions must be compiled with NVCC
- NVCC is a compiler driver
  - Works by invoking all the necessary tools and compilers like cudacc, g++, cl, ...
- NVCC outputs:
  - C code (host CPU Code)
    - Must then be compiled with the rest of the application using another tool
  - PTX
    - Object code directly
    - Or, PTX source, interpreted at runtime



# Linking

- Any executable with CUDA code requires two dynamic libraries:
  - The CUDA runtime library (`cudaart`)
  - The CUDA core library (`cuda`)

# Debugging Using the Device Emulation Mode

- An executable compiled in **device emulation mode** (`nvcc -deviceemu`) runs completely on the host using the CUDA runtime
  - No need of any device and CUDA driver
  - Each device thread is emulated with a host thread
- Running in device emulation mode, one can:
  - Use host native debug support (breakpoints, inspection, etc.)
  - Access any device-specific data from host code and vice-versa
  - Call any host function from device code (e.g. `printf`) and vice-versa
  - Detect deadlock situations caused by improper usage of `__syncthreads`

# Device Emulation Mode Pitfalls

- Emulated device threads execute sequentially, so **simultaneous accesses of the same memory location by multiple threads** could produce different results.
- **Dereferencing device pointers** on the host or host pointers on the device can produce correct results in device emulation mode, but will generate an error in device execution mode

# Floating Point

- Results of floating-point computations will slightly differ because of:
  - Different compiler outputs, instruction sets
  - Use of extended precision for intermediate results
    - There are various options to force strict single precision on the host