BITS Pilani

Pilani | Dubai | Goa | Hyderabad

CSISF301: Principles of Programming Languages

CUDA PROGRAMMING

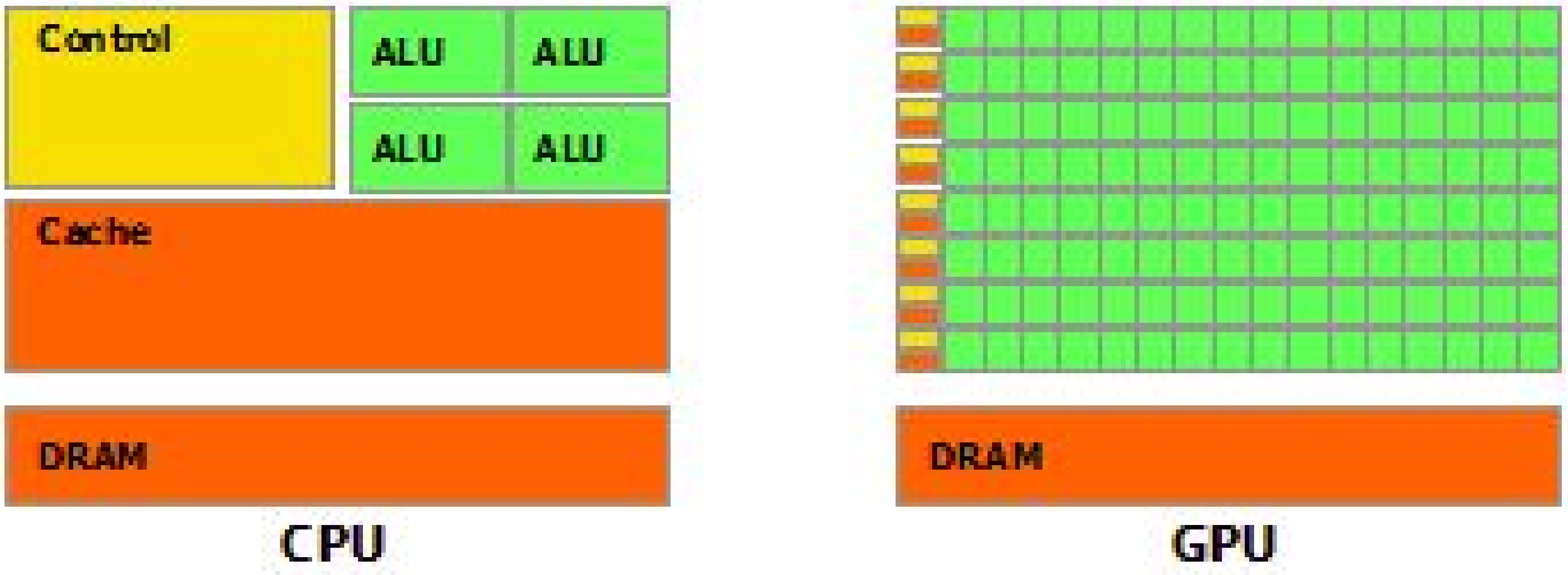
Tutorial

August 6, 2015

CUDA

- CUDA (Compute Unified Device Architecture)
- CUDA Architecture
  - Expose GPU parallelism for general-purpose computing
  - Achieves Performance
- CUDA C/C++
  - Based on industry-standard C/C++
  - Small set of extensions to enable heterogeneous programming
  - APIs to manage devices, memory etc.

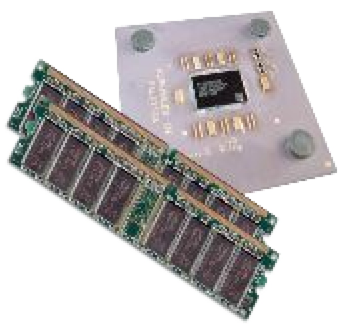
Why a GPU?



- GPU is specialized for compute-intensive, highly parallel computation - exactly what graphics rendering is about
- More transistors are devoted to data processing rather than data caching and flow control.

Heterogeneous Computing

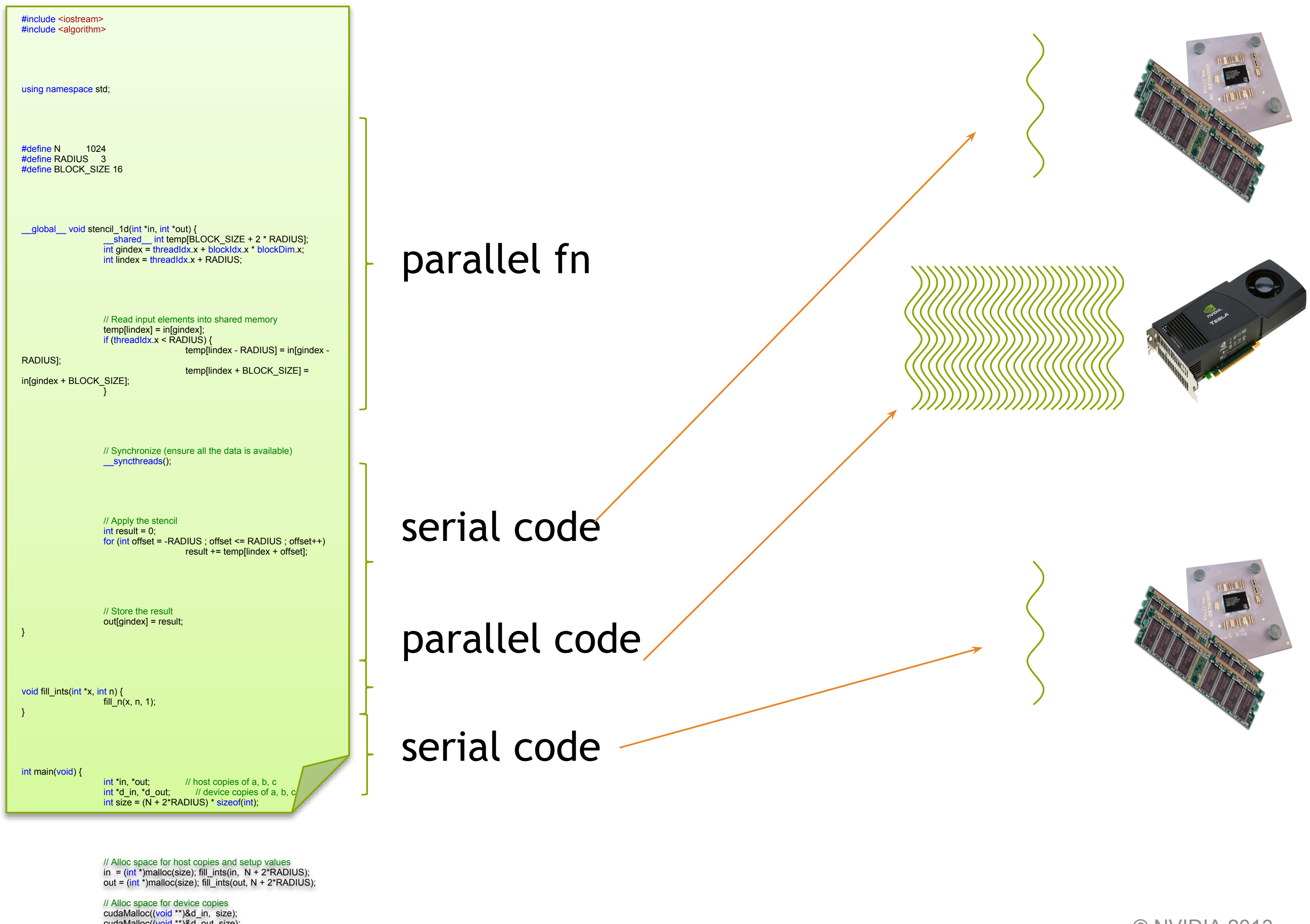
- Terminology:
  - *Host* The CPU and its memory (host memory)
  - *Device* The GPU and its memory (device memory)



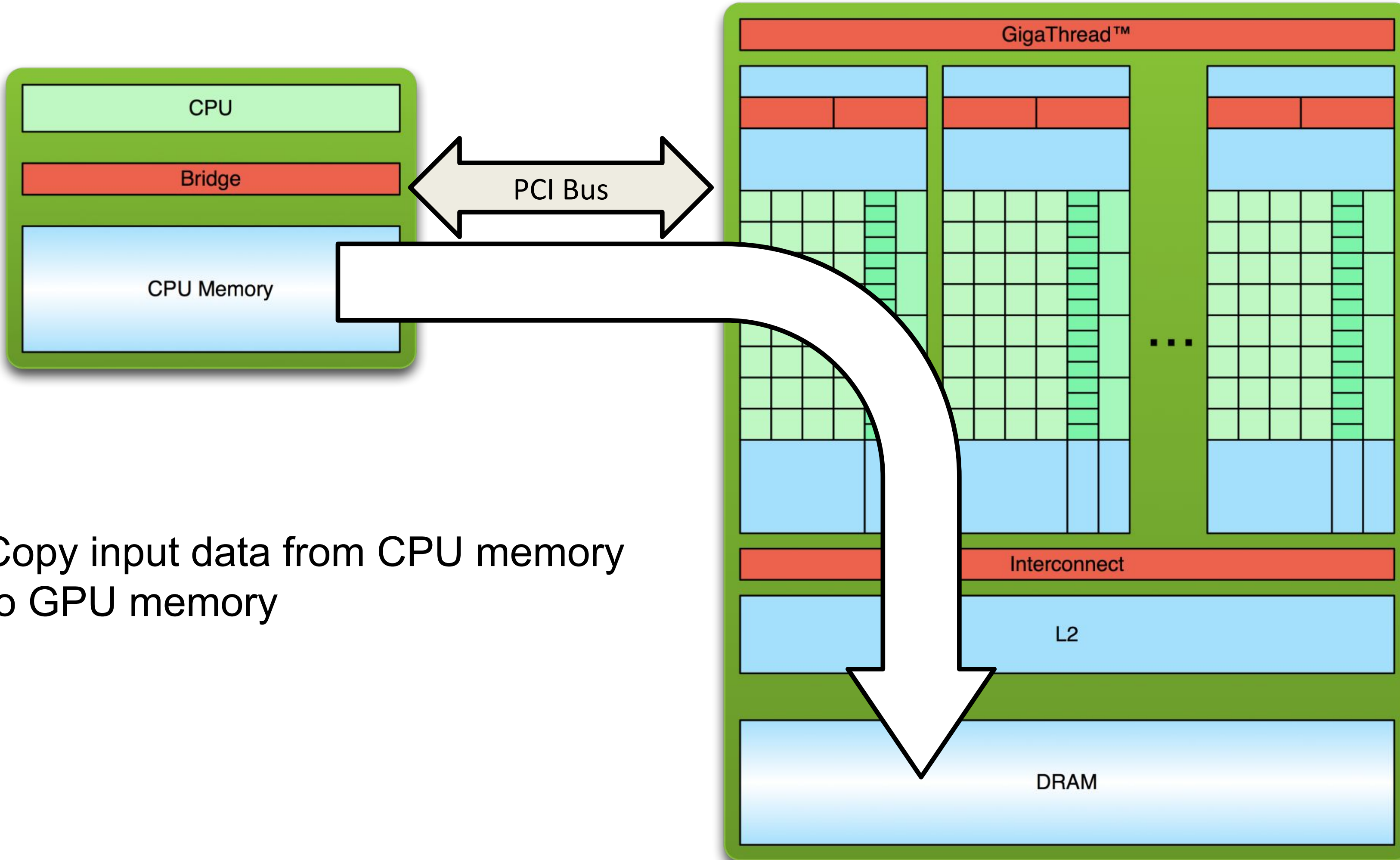
Host

Device

Heterogeneous Computing

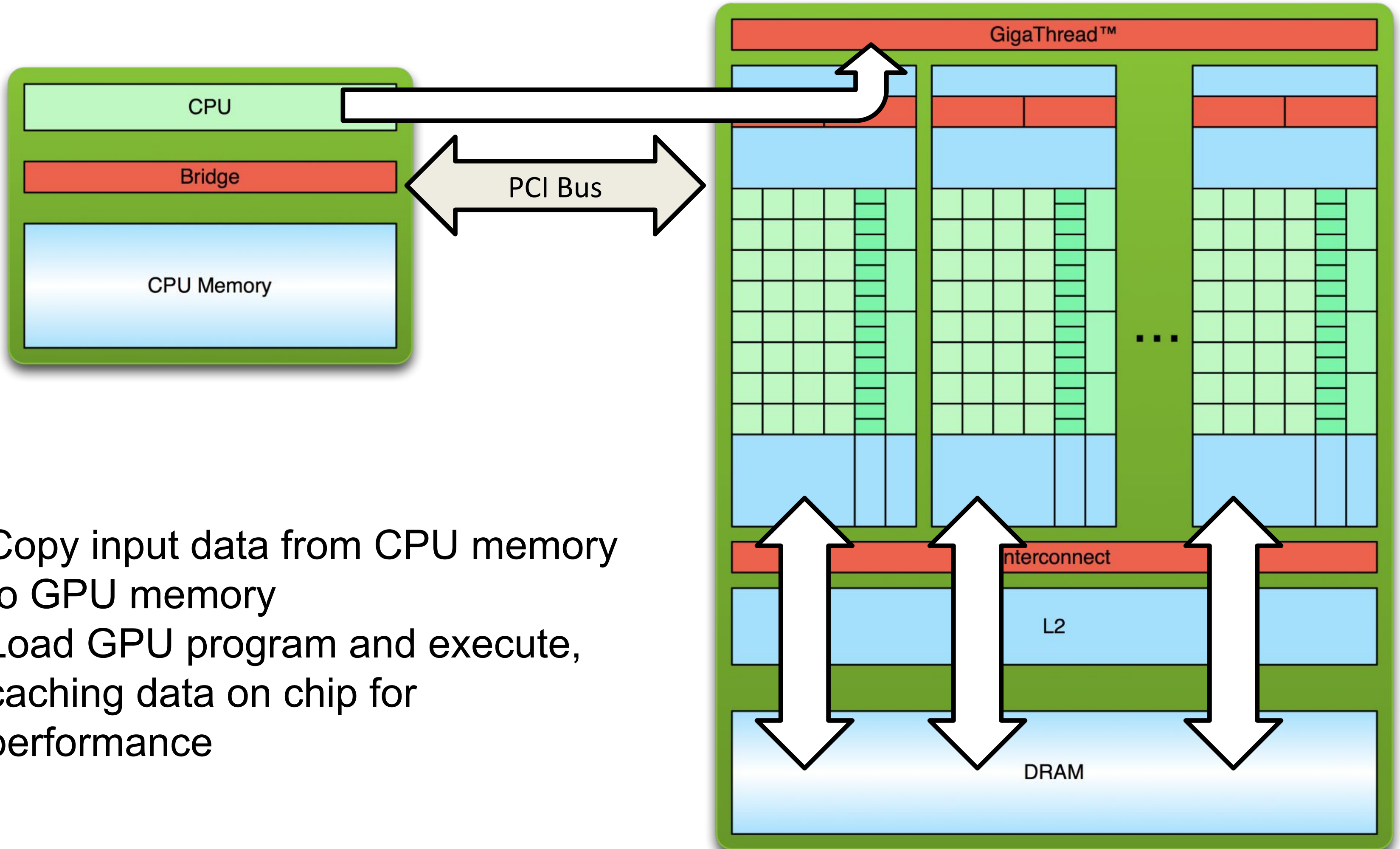


Simple Processing Flow



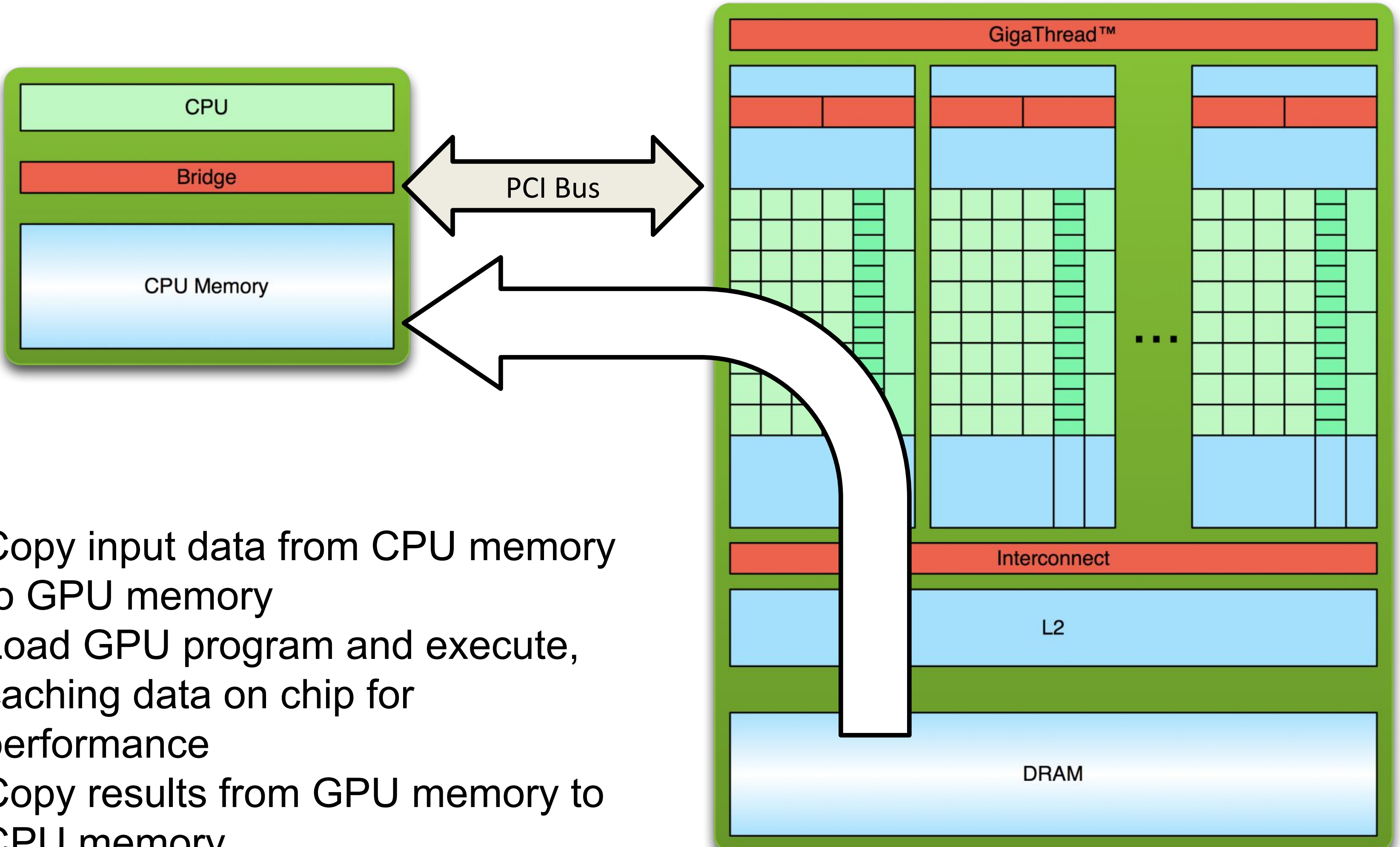
1. Copy input data from CPU memory to GPU memory

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

Flynn's taxonomy

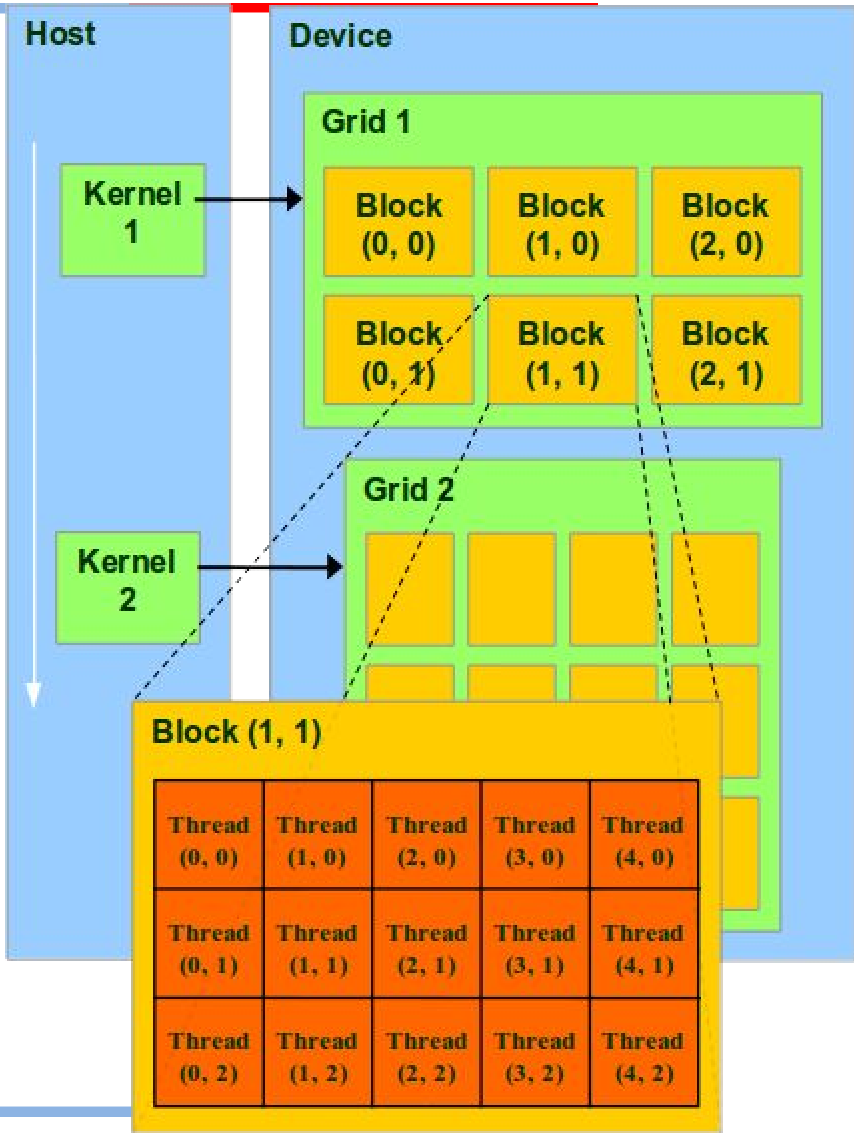
- Classification of computer architectures, proposed by Michael J. Flynn in 1966

  1.1 Single instruction stream single data stream (SISD)
  1.2 Single instruction stream, multiple data streams (SIMD)
  1.3 Multiple instruction streams, single data stream (MISD)
  1.4 Multiple instruction streams, multiple data streams (MIMD)

- Nvidia refers to GPUs as Single Instruction Multiple Thread (SIMT)
  - Every thread of a warp executes the same instruction

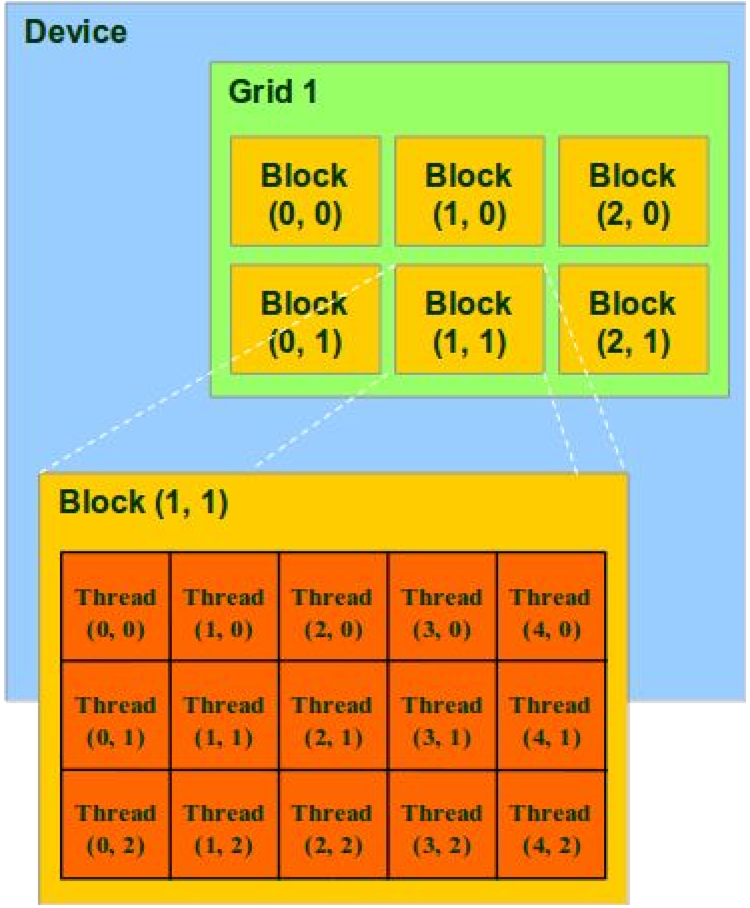
Thread Batching: Grids and Blocks

- A kernel is executed as a grid of thread blocks
  - All threads share data memory space
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate



Block and Thread IDs

- Threads and blocks have IDs
  - So each thread can 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
  - …



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
    - Must return void
- __device__ and __host__ can be used together

Kernel Function

- 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>>>(...);
```

Kernel Launching Parameters

- Launch parameters:
  - grid dimensions (up to 2D), dim3 type
  - thread-block dimensions (up to 3D), dim3 type
  - shared memory: number of bytes per block
    - for external smem variables declared without size
    - Optional, 0 by default
  - Stream ID
    - Optional, 0 by default

E.g.

```
dim3 grid(16, 16);
dim3 block(16,16);
kernel<<<grid, block, 0, 0>>>(...);
kernel<<<32, 512>>>(...);
```

Streams

- A sequence of operations that execute in issue-order on the GPU
- The order in which the operations are added to a stream specifies the order in which they will be executed
- Programming model used to effect concurrency
  - CUDA operations in different streams may run concurrently
  - CUDA operations from different streams may be interleaved
- Streams introduce task parallelism
- Plays an important role in accelerating the applications

The Default Stream

- When no stream is specified, the default stream (also called the "null stream") is used.
- The default stream is different from other streams because it is a synchronizing stream with respect to operations on the device:
  - No operation in the default stream will begin until all previously issued operations *in any stream on the device* have completed
  - An operation in the default stream must complete before any other operation (in any stream on the device) will begin.

The Default Stream

- Example:

```
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
increment<<<1,N>>>(d_a)
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

- All three operations are issued on the same stream
- Since the host-to-device data transfer on the first line is synchronous
  - CPU thread will not reach the kernel call on the second line until the host-to-device transfer is complete.
- Once the kernel is issued, the CPU thread moves to the third line
  - Transfer on that line cannot begin due to the device-side order of execution.

Non-Default Streams

- Non-default streams in CUDA C/C++ are declared, created, and destroyed in host code
- Example:
  - cudaStream_t stream1;
    cudaError_t result;
    result = cudaStreamCreate(&stream1)
    result = cudaStreamDestroy(stream1)
- Data transfer to a non-default stream by using the [cudaMemcpyAsync]() function
- Example:
  - result = cudaMemcpyAsync(d_a, a, N, cudaMemcpyHostToDevice, stream1)
  - cudaMemcpyAsync() is non-blocking on the host, so control returns to the host thread immediately after the transfer is issued.

Overlapping Kernel Execution and data transfer - Approach I

Example:

```
for (int i = 0; i < nStreams; ++i) {
   int offset = i * streamSize;
   cudaMemcpyAsync(&d_a[offset], &a[offset], streamBytes,
 cudaMemcpyHostToDevice, stream[i]);
   kernel<<<streamSize/blockSize, blockSize, 0,
 stream[i]>>>(d_a, offset);
   cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes,
 cudaMemcpyDeviceToHost, stream[i]);
 }
```

Overlapping Kernel Execution and data transfer - Approach II

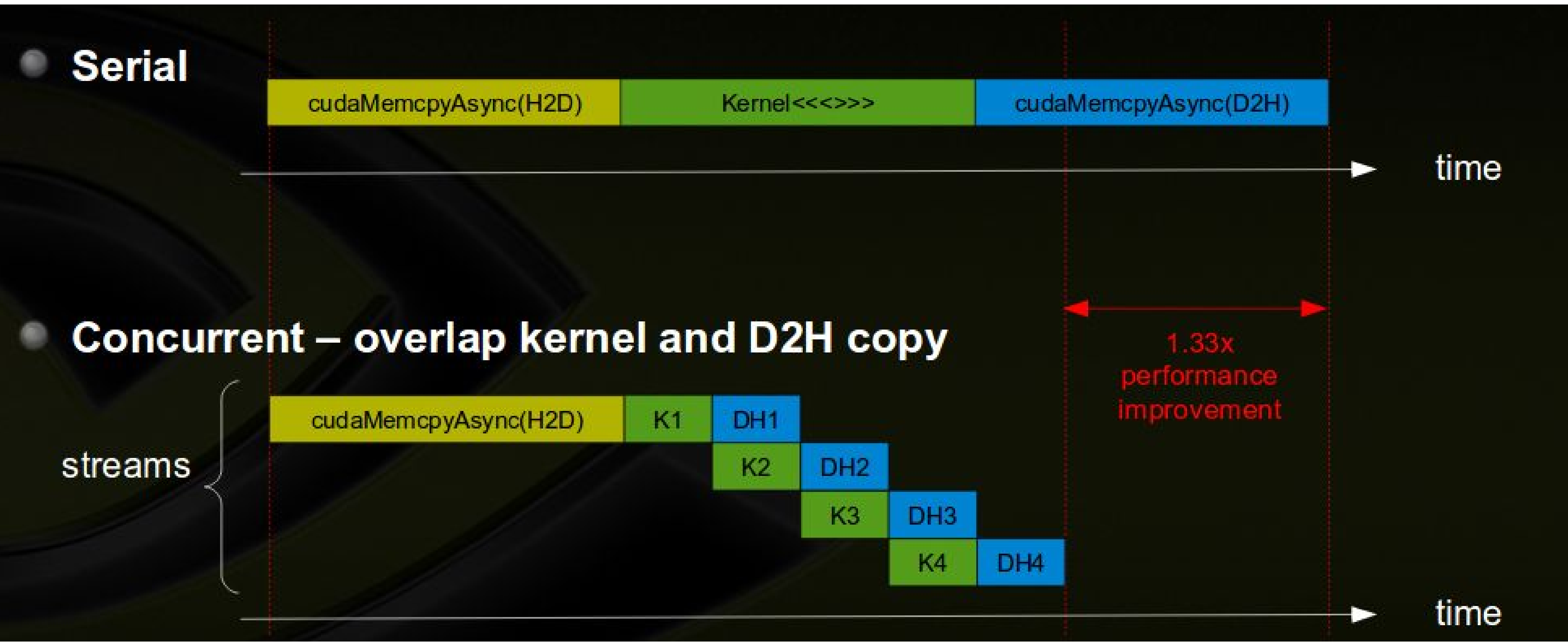
- Group similar operation together

```
for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  cudaMemcpyAsync(&d_a[offset], &a[offset],
              streamBytes, cudaMemcpyHostToDevice, cudaMemcpyHostToDevice,
 stream[i]); }

 for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);     }

 for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  cudaMemcpyAsync(&a[offset], &d_a[offset],
              streamBytes, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToHost,
 stream[i]);       }
```

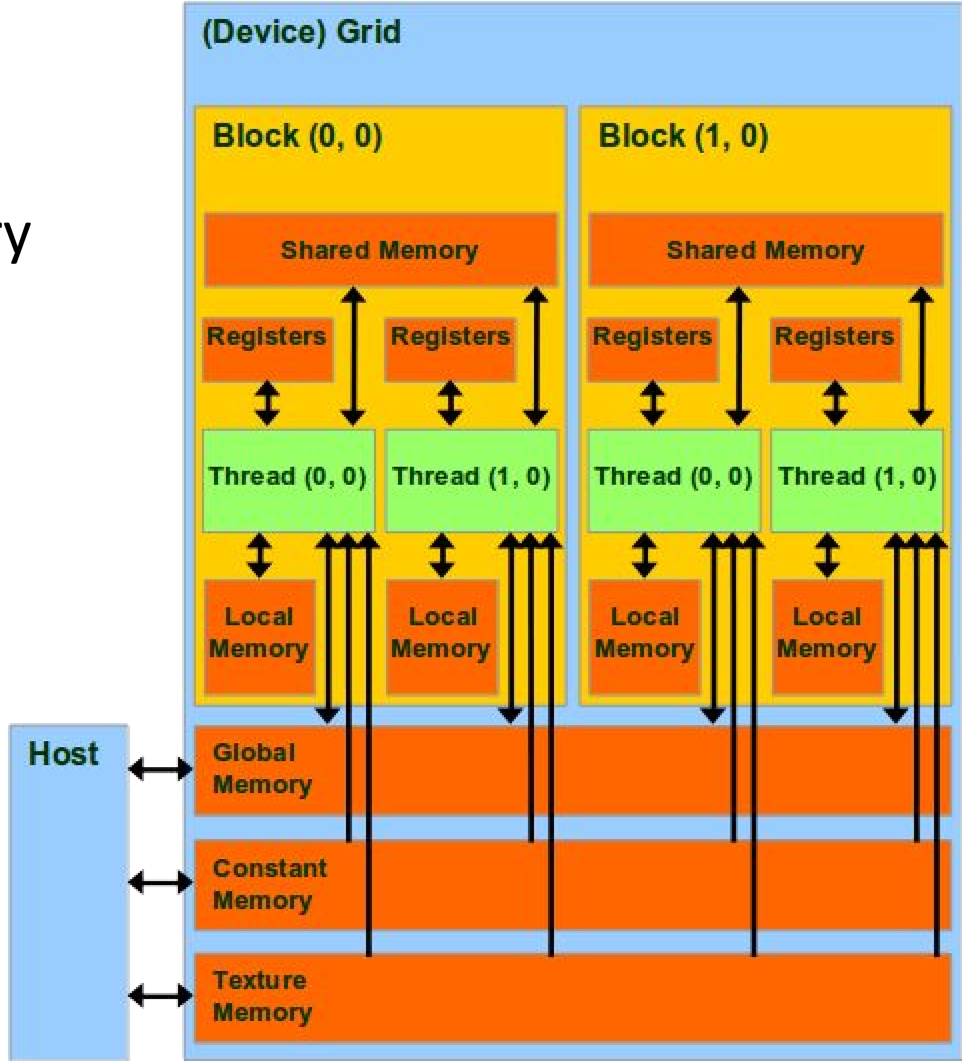
Execution Time



CUDA Device Memory Space Overview

- Each thread 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
  - Read only per-grid texture memory

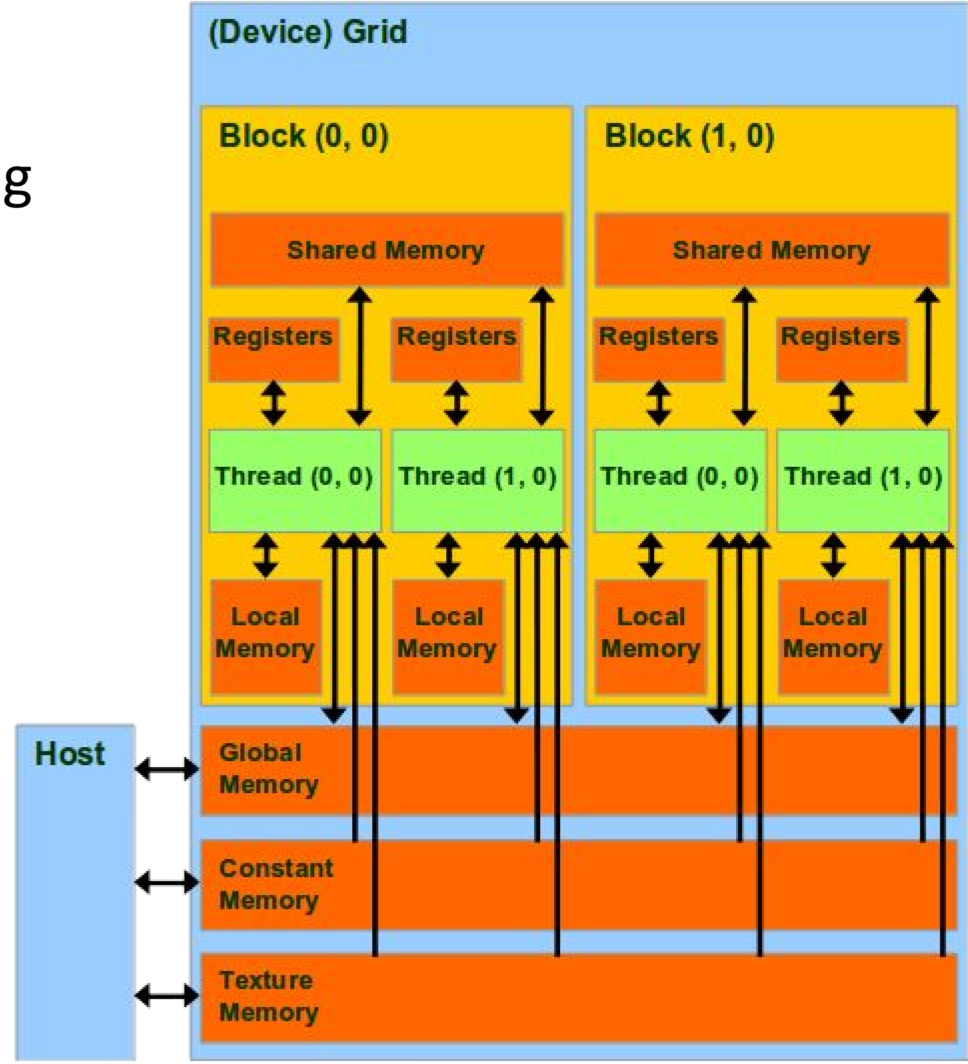
- The host can R/W global, constant, and texture memories



Global, Constant, and Texture Memories

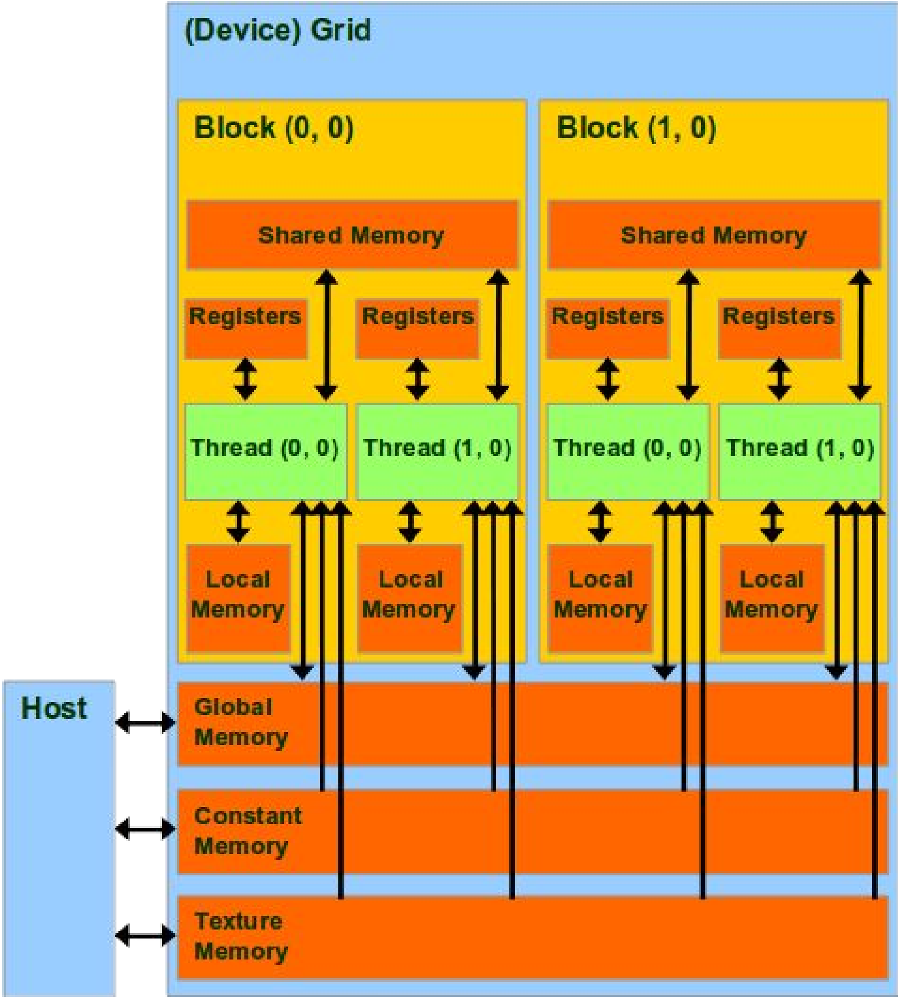
- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads

- Texture and Constant Memories
  - Constants initialized by host
  - Contents visible to all threads



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

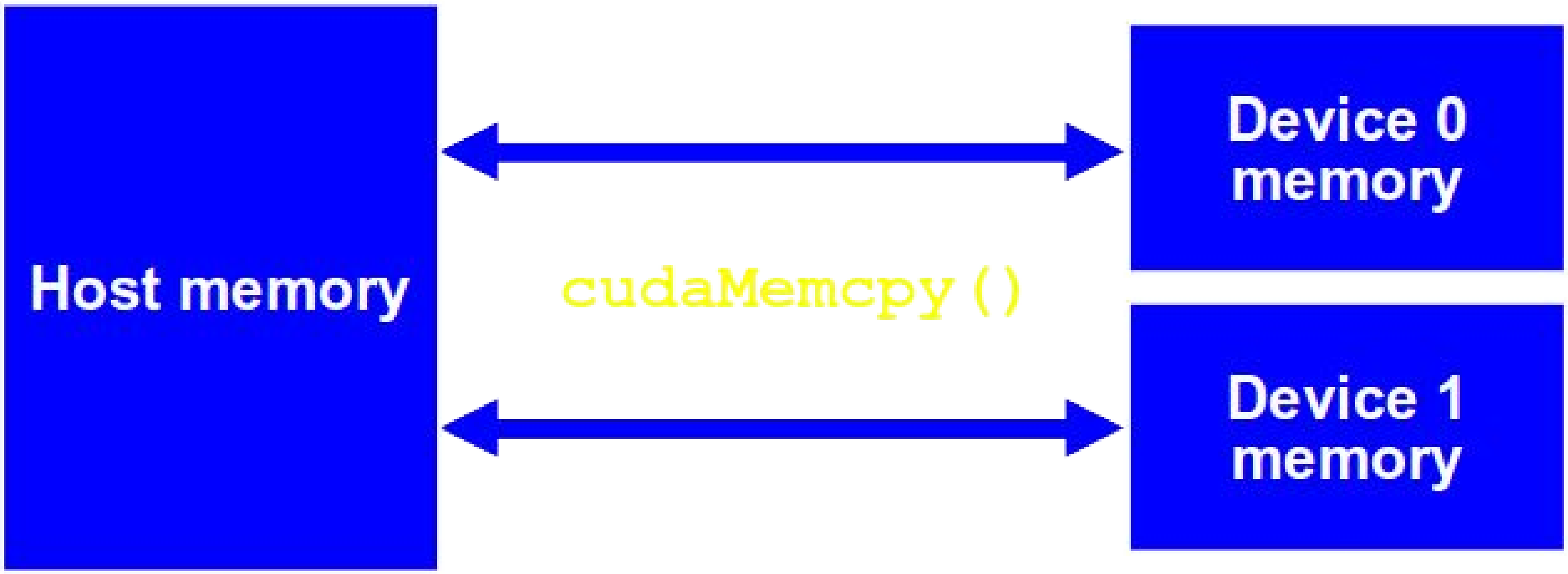


CUDA Device Memory Allocation

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

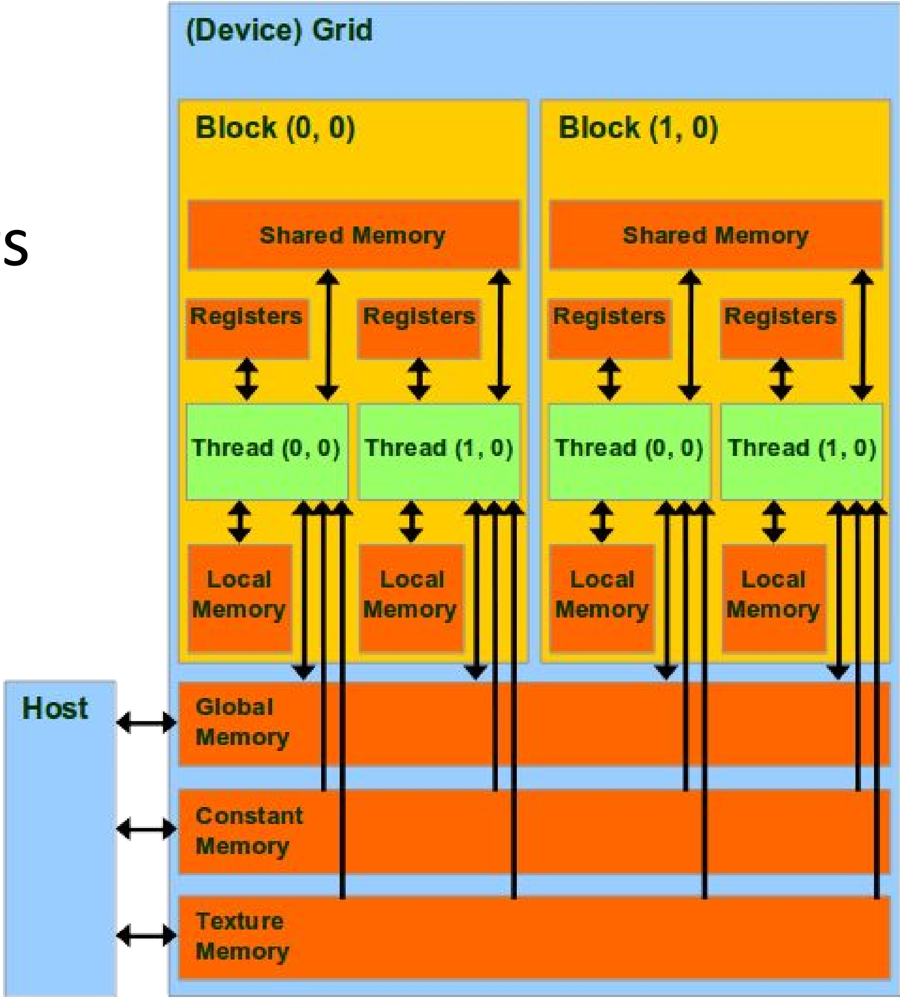
```
BLOCK_SIZE = 64;
Matrix Md;
int size = BLOCK_SIZE * BLOCK_SIZE * sizeof(float);
cudaMalloc((void**)&Md.elements, size);
cudaFree(Md.elements);
```

CUDA Memory Model



CUDA Host-Device Data Transfer

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



CUDA Host-Device Data Transfer

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

```
cudaMemcpy(Md.elements, Mh.elements, size,
           cudaMemcpyHostToDevice);
cudaMemcpy(Mh.elements, Md.elements, size,
           cudaMemcpyDeviceToHost);
```

Example 1

- Allocate CPU memory for n integers
- Allocate GPU memory for n integers
- Initialize GPU memory to 0s
- Copy from GPU to CPU
- Print the values

Code Walkthrough

```
#include <stdio.h>
int main()
{
int dimx = 16;
int num_bytes = dimx*sizeof(int);
int *d_a=0, *h_a=0; // device and host
  pointers
```

Code Walkthrough

```
#include <stdio.h>
int main()
{
int dimx = 16;
int num_bytes = dimx*sizeof(int);
int *d_a=0, *h_a=0; // device and host pointers
h_a = (int*)malloc(num_bytes);
cudaMalloc( (void**)&d_a, num_bytes );
if( 0==h_a || 0==d_a)
{
printf("couldn't allocate memory\n");
return 1;
}
```

Code Walkthrough

```
#include <stdio.h>
int main()
{
int dimx = 16;
int num_bytes = dimx*sizeof(int);
int *d_a=0, *h_a=0; // device and host pointers
h_a = (int*)malloc(num_bytes);
cudaMalloc( (void**)&d_a, num_bytes );
if( 0==h_a || 0==d_a)
{
printf("couldn't allocate memory\n");
return 1;
}
cudaMemset( d_a, 0, num_bytes );
cudaMemcpy( h_a, d_a, num_bytes,
  cudaMemcpyDeviceToHost );
```

Code Walkthrough

```
#include <stdio.h>
int main()
{
int dimx = 16;
int num_bytes = dimx*sizeof(int);
int *d_a=0, *h_a=0; // device and host pointers
h_a = (int*)malloc(num_bytes);
cudaMalloc( (void**)&d_a, num_bytes );
if( 0==h_a || 0==d_a)
{
printf("couldn't allocate memory\n");
return 1;
}
cudaMemset( d_a, 0, num_bytes );
cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );
```

```
for(int i=0; i<dimx; i++)
printf("%d ", h_a[i] );
printf("\n");
free( h_a );
cudaFree( d_a );
Return 0;
}
```

Code Walkthrough: Kernel

```
__global__ void kernel( int *a )
{
int idx = blockIdx.x*blockDim.x+threadIdx.x;
a[idx] = 7;
}
```

Code Walkthrough

```
#include <stdio.h>
__global__ void kernel( int *a )
{
int idx = blockIdx.x*blockDim.x +
   threadIdx.x;
a[idx] = 7;
}
int main()
{
int dimx = 16;
int num_bytes = dimx*sizeof(int);
int *d_a=0, *h_a=0; // device and host pointers
h_a = (int*)malloc(num_bytes);
cudaMalloc( (void**)&d_a, num_bytes );
if( 0==h_a || 0==d_a )
{
printf("couldn't allocate memory\n");
return 1;
}
cudaMemset( d_a, 0, num_bytes );
dim3 grid, block;
block.x = 4;
grid.x = dimx / block.x;
kernel<<<grid, block>>>( d_a );
cudaMemcpy( h_a, d_a, num_bytes,
   cudaMemcpyDeviceTo
Host );
for(int i=0; i<dimx; i++)
printf("%d ", h_a[i] );
printf("\n");
free( h_a );
cudaFree( d_a );
return 0;
}
```

Kernel Output

```
__global__ void kernel( int *a )
{
int idx = blockIdx.x*blockDim.x + threadIdx.x;
a[idx] =7;
}
__global__ void kernel( int *a )
{
int idx = blockIdx.x*blockDim.x + threadIdx.x;
a[idx] =blockIdx.x;
}
__global__ void kernel( int *a )
{
int idx = blockIdx.x*blockDim.x + threadIdx.x;
a[idx] =threadIdx.x;
}
```

Kernel Output

```
__global__ void kernel( int *a )
{
int idx = blockIdx.x*blockDim.x + threadIdx.x;
a[idx] =7;
}
```

Output:
7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7

```
__global__ void kernel( int *a )
{
int idx = blockIdx.x*blockDim.x + threadIdx.x;
a[idx] =blockIdx.x;
}
```

Output:
0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3

```
__global__ void kernel( int *a )
{
int idx = blockIdx.x*blockDim.x + threadIdx.x;
a[idx] =threadIdx.x;
}
```

Output:
0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3

Example 2: Vector Addition

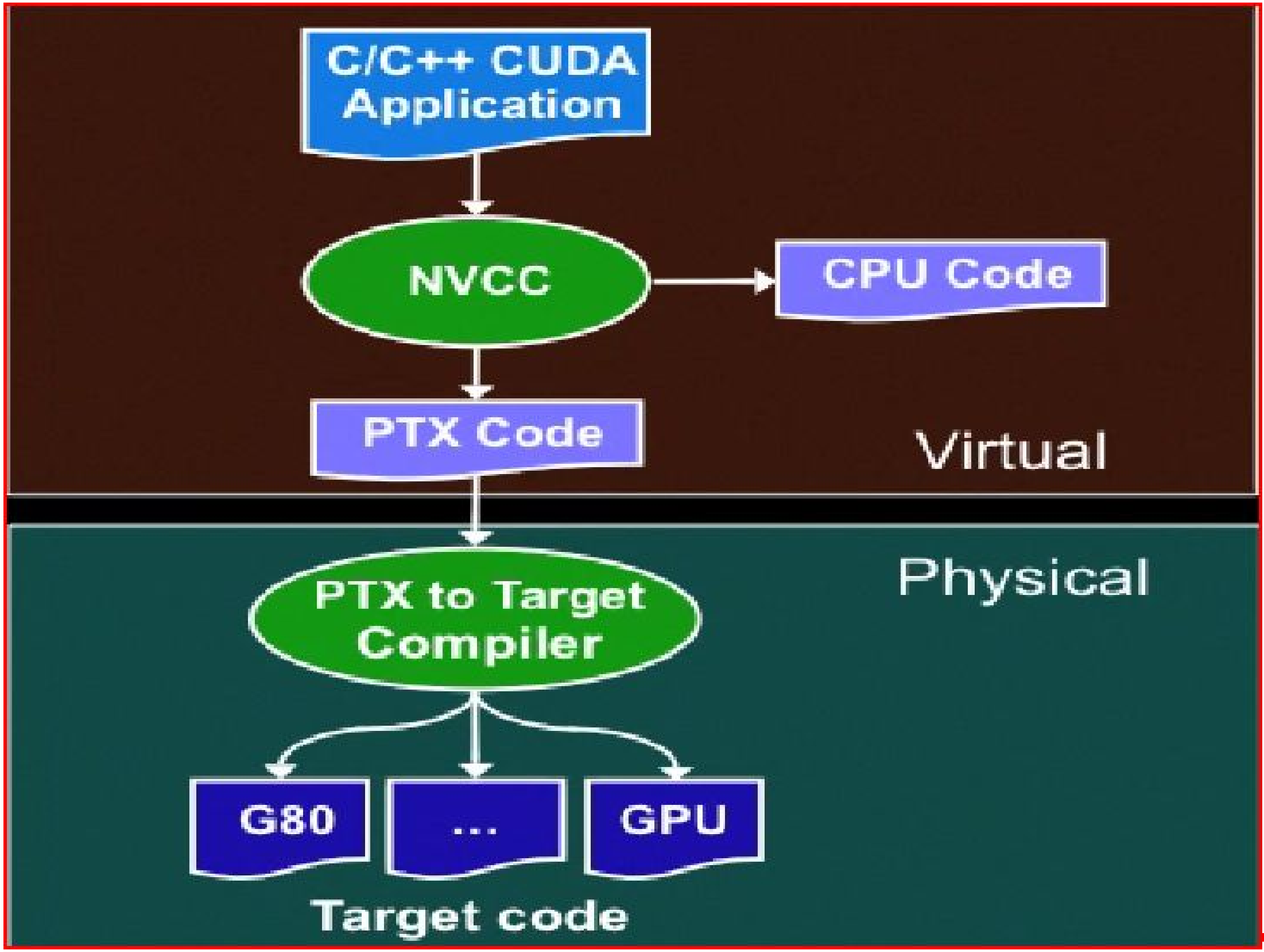
```
// Compute vector sum C = A+B
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

```
int main()
{
    // Run N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Host code for Vector Addition

```
// allocate and initialize host (CPU) memory
float *h_A = …,   *h_B = …;
// allocate device (GPU) memory
float *d_A, *d_B, *d_C;
cudaMalloc( (void**) &d_A, N * sizeof(float));
cudaMalloc( (void**) &d_B, N * sizeof(float));
cudaMalloc( (void**) &d_C, N * sizeof(float));
// copy host memory to device
cudaMemcpy( d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevice) );
cudaMemcpy( d_B, h_B, N * sizeof(float), cudaMemcpyHostToDevice) );
// execute the kernel on N/256 blocks of 256 threads each
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);
```

Execution of CUDA code



Revisiting CUDA Extensions

- Declaration specifiers to indicate where things live

```
__global__ void KernelFunc(...);  // kernel callable from host
__device__ void DeviceFunc(...);  // function callable on device
__device__ int  GlobalVar;        // variable in device memory
__shared__ int  SharedVar;        // in per-block shared memory
```

- Extend function invocation syntax for parallel kernel launch

```
KernelFunc<<<500, 128>>>(...);      // 500 blocks, 128 threads each
```

- Special variables for thread identification in kernels

```
dim3 threadIdx;  dim3 blockIdx;  dim3 blockDim;
```

- Intrinsics that expose specific operations in kernel code

```
__syncthreads();              // barrier synchronization
```

Revisiting CUDA Extensions

- Standard mathematical functions

  sinf, powf, atanf, ceil, min, sqrtf, etc.

- Atomic memory operations

  atomicAdd, atomicMin, atomicAnd, atomicCAS, etc.

- Texture accesses in kernels

  ```
  texture<float,2> my_texture;  // declare texture reference
  float4 texel = texfetch(my_texture, u, v);
  ```

Memory Extensions

- Explicit memory allocation returns pointers to GPU memory

  cudaMalloc(), cudaFree()

- Explicit memory copy for host ↔ device, device ↔ device

  cudaMemcpy(), cudaMemcpy2D(), ...

- Texture management

  cudaBindTexture(), cudaBindTextureToArray(), ...

- OpenGL & DirectX interoperability

  cudaGLMapBufferObject(), cudaD3D9MapVertexBuffer(), …

THANK YOU