CENG443
Heterogeneous Parallel Programming

CUDA Memory

Memory and Registers in Von-Neumann Model

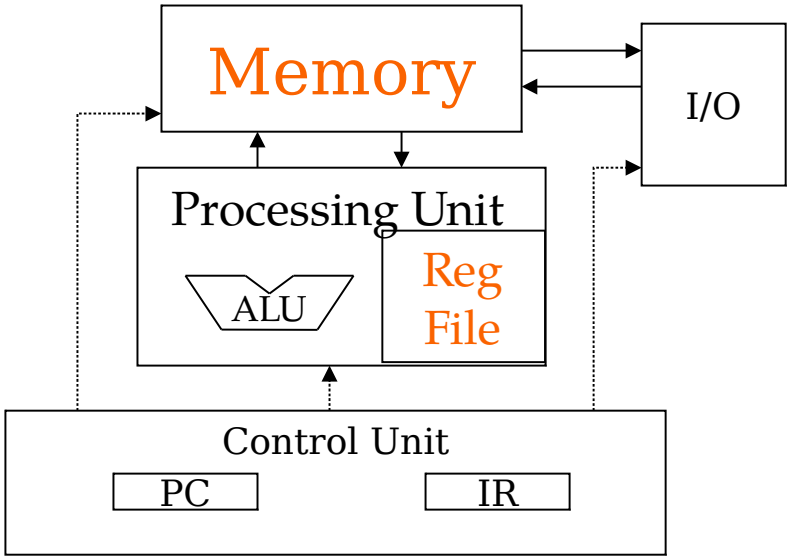


Image Blurring Kernel

```
// Get the average of the surrounding 2xBLUR_SIZE x 2xBLUR_SIZE box
 for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow) {
     for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol) {

         int curRow = Row + blurRow;
         int curCol = Col + blurCol;
         // Verify we have a valid image pixel
         if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
             pixVal += in[curRow * w + curCol];
             pixels++; // Keep track of number of pixels in the accumulated total
          }
     }
 }
```

In every iteration of the inner loop, one global memory access is performed for one floating-point addition

Compute-to-Global Memory Access Ratio

The number of floating-point calculation performed for each access to the global memory within a region of a program

It is 1 for floating-point add operation in image blur kernel

Memory-bound programs

- execution speed is limited by memory access throughput

GPU Performance

GPU device with the global memory bandwidth 1,000 GB/s, or 1 TB/s

4 bytes in each single-precision floating-point value

1000/4=250 giga single-precision operands per second to be loaded

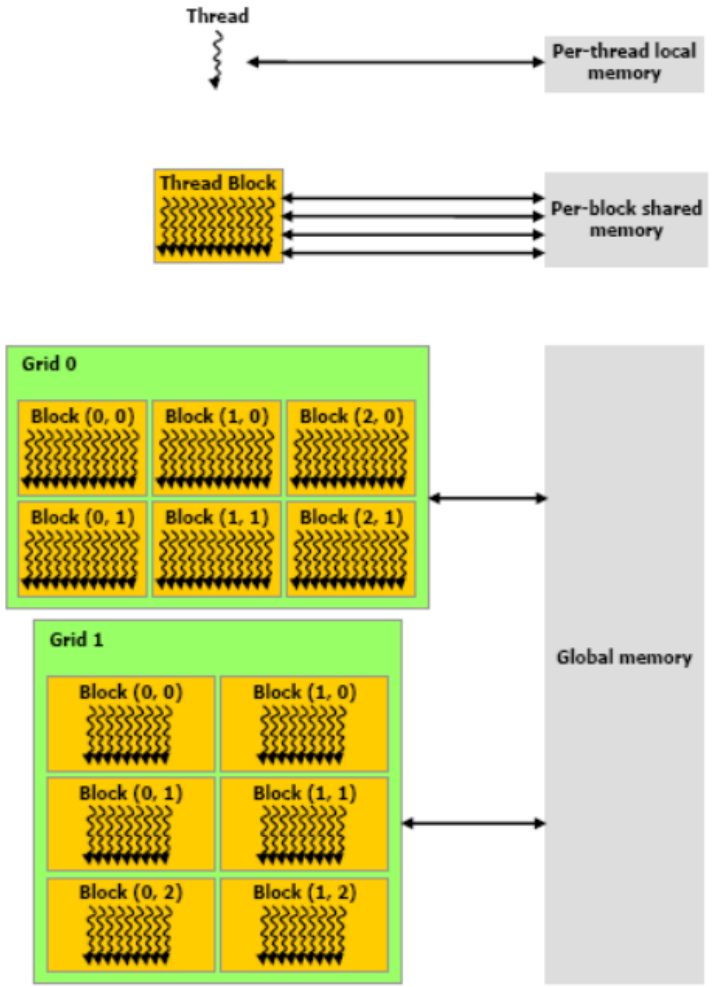
- no more than 250 giga floating-point operations per second (GFLOPS)

Peak single-precision performance of 12 TFLOPS

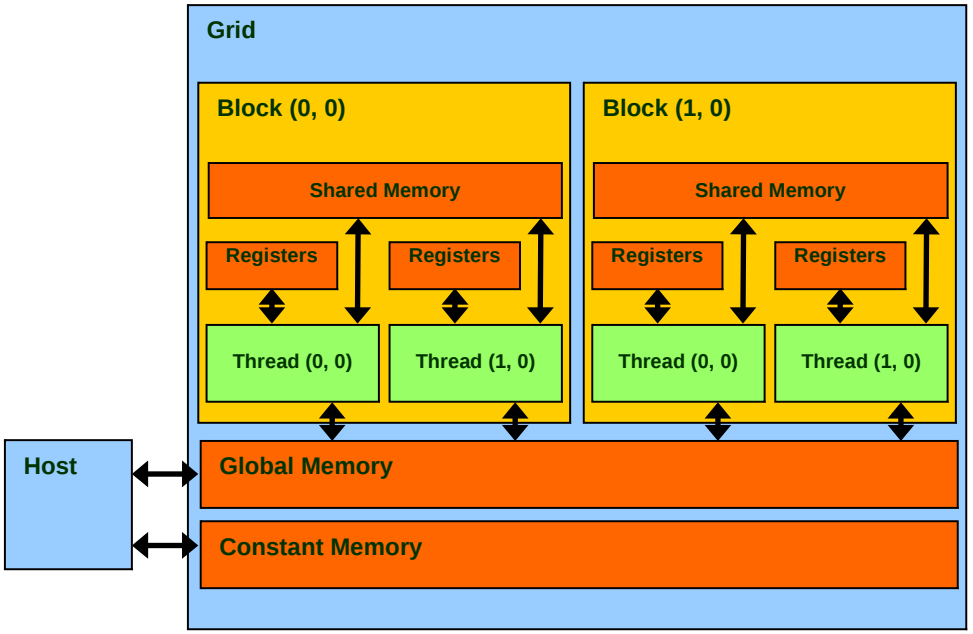
Tiny fraction = 250/12000 = 2%

Need to find ways of reducing global memory accesses!

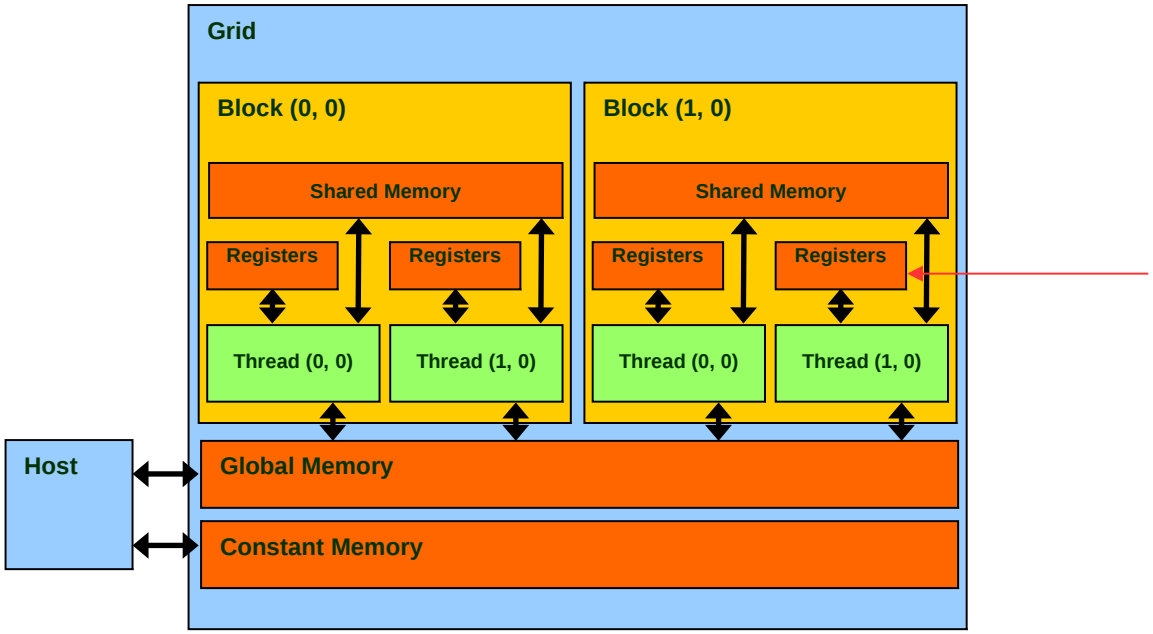
CUDA Memories



CUDA Memories



Registers



- Fastest
- Only accessible by a thread
- Lifetime of a thread
- Limited capacity of registers

Local Memory

Local memory is everything on the stack that can’t fit in registers. The scope of local memory is just the thread.

Local memory is stored in global memory (much slower than registers!)

Local Variables

All scalar variables declared in kernel and device functions are placed into registers

Automatic array variables are not stored in registers (The compiler may decide to store an array into registers if all accesses are done with constant index values)

Similar to scalar variables, the scope of these arrays is limited to individual threads

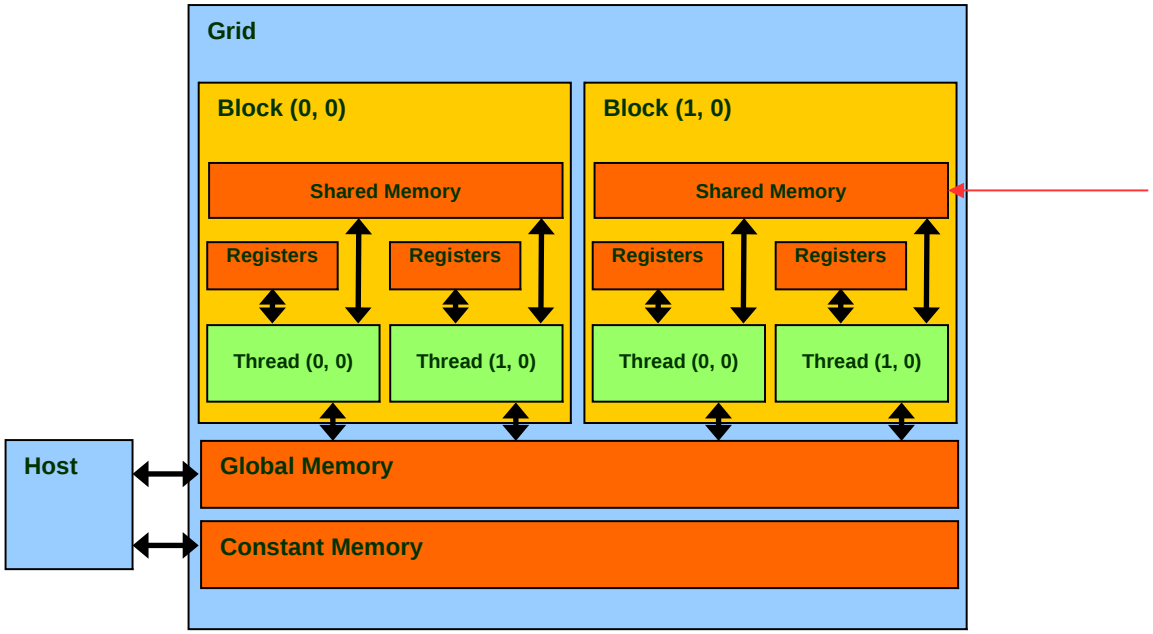
Once a thread terminates its execution, the contents of its local variables also cease to exist

Array variables are rarely used in kernel functions

Image Blurring Kernel

```
// Get the average of the surrounding 2xBLUR_SIZE x 2xBLUR_SIZE box
 for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow) {
    for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol) {

        int curRow = Row + blurRow;
        int curCol = Col + blurCol;
        // Verify we have a valid image pixel
        if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
            pixVal += in[curRow * w + curCol];
            pixels++; // Keep track of number of pixels in the accumulated total
        }
    }
 }
```

Shared Memory



- Extremely fast (~4cycles)
- Highly parallel
- Restricted to a block
- Small (typically 48 KB per SM)

Shared Memory in Fermi

64KB configurable shared memory and L1 cache

48 KB shared memory and 16 KB L1 cache OR

16 shared memory and 48 L1 cache

With shared memory, you get full control as to what gets stored where, while with cache, everything is done automatically

Even though the compiler and the GPU can still be very clever in optimizing memory accesses, you can sometimes still find a better way

Shared Variables

Shared variables reside in the shared memory

The scope of a shared variable is within a thread block, all threads in a block see the same version of a shared variable, subject to race conditions

A private version of the shared variable is created for and used by each thread block during kernel execution

When a kernel terminates its execution, the contents of its shared variables cease to exist

An efficient means for threads within a block to collaborate with one another, to hold the portion of global memory data that are heavily used in a kernel execution phase

Shared Variable Example

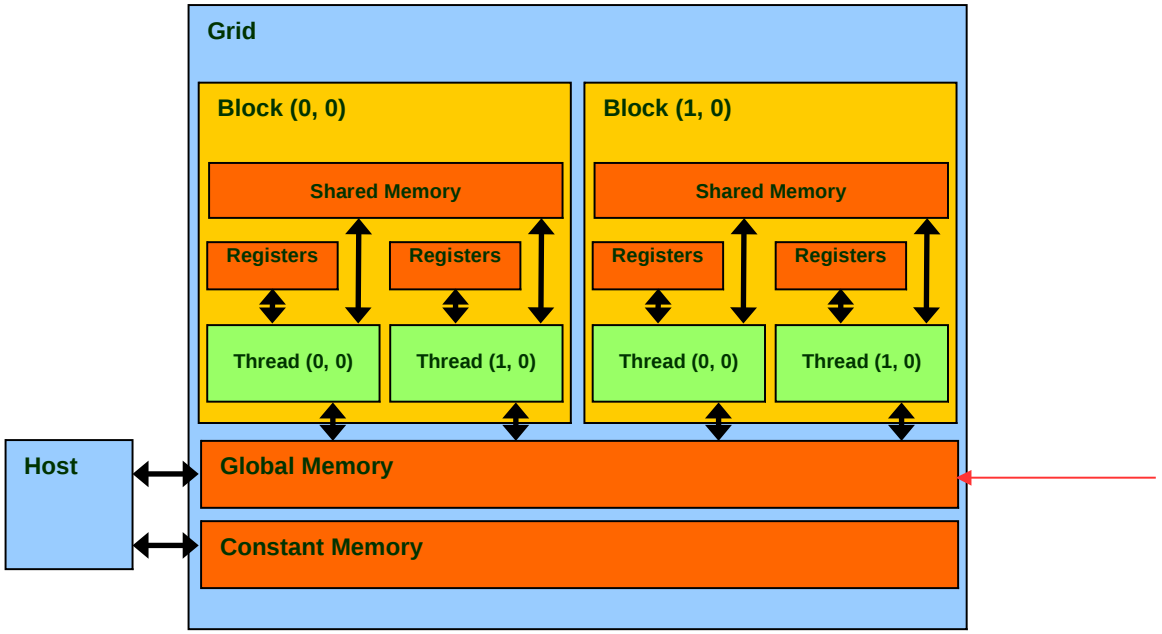
Statically (size known at compile time)

```
__global__ void Kernel(int count)
{
    __shared__ int a[1024];
...
}
```

Dynamically (size not known until runtime)

```
__global__ void Kernel(int count)
{
    extern __shared__ int a[];
...
}
Kernel<<< gridDim, blockDim, numBytesSharedMem >>>(count)
```

Global Memory



- Typically implemented in DRAM
- High access latency:
  400-800 cycles
- Finite access bandwidth
- Potential of traffic congestion
- Throughput up to 177GB/s

Global Variables

Visible to all threads of all kernels, can be used as a means for threads to collaborate across blocks

The only easy way to synchronize between threads from different thread blocks or to ensure data consistency across threads when accessing global memory is by terminating the current kernel execution

Vector Addition Host Code

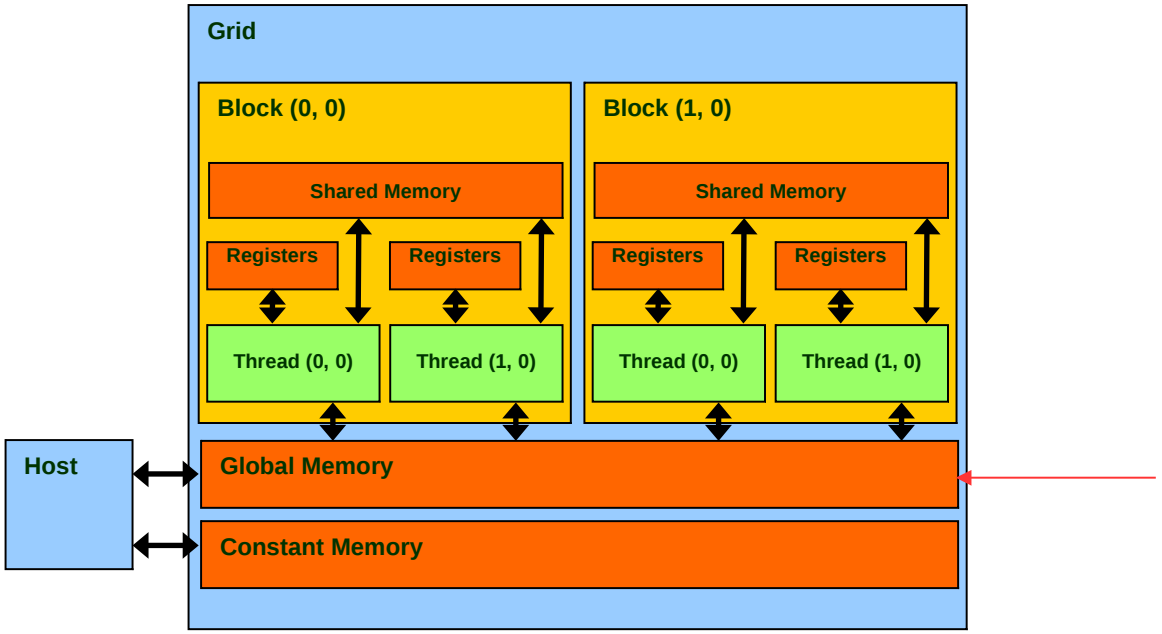
```
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int size = n * sizeof(float); float *d_A, *d_B, *d_C;

    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);



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

Global memory:

Allocate with
cudaMalloc(void** devPtr, size_t size)

Free with
cudaFree(void* devPtr)

Constant Memory



- Read only
- Short latency and high bandwidth when all threads access the same location
- Limited size, 64KB

Constant Variables

Declaration of constant variables must be outside any function body

Only accessible on the device

The scope of a constant variable spans all grids, meaning that all threads in all grids see the same version of a constant variable

Constant variables are often used for variables that provide input values to kernel functions

Constant Memory Example

In global scope (outside of kernel, at top level of program):

```
__constant__ int foo[1024];
int main() {
...
int *test = new int[1024];
memset(test, 0, sizeof(int) * 1024);
for (int i = 0; i < 1024; i++) {
     test[i] = 100;
}
cudaMemcpyToSymbol(foo, test, sizeof(int) * 1024);
…
__global__ void kernel() {
   int tId = threadIdx.x + blockIdx.x * blockDim.x;
   int a = foo[tId];
}
```

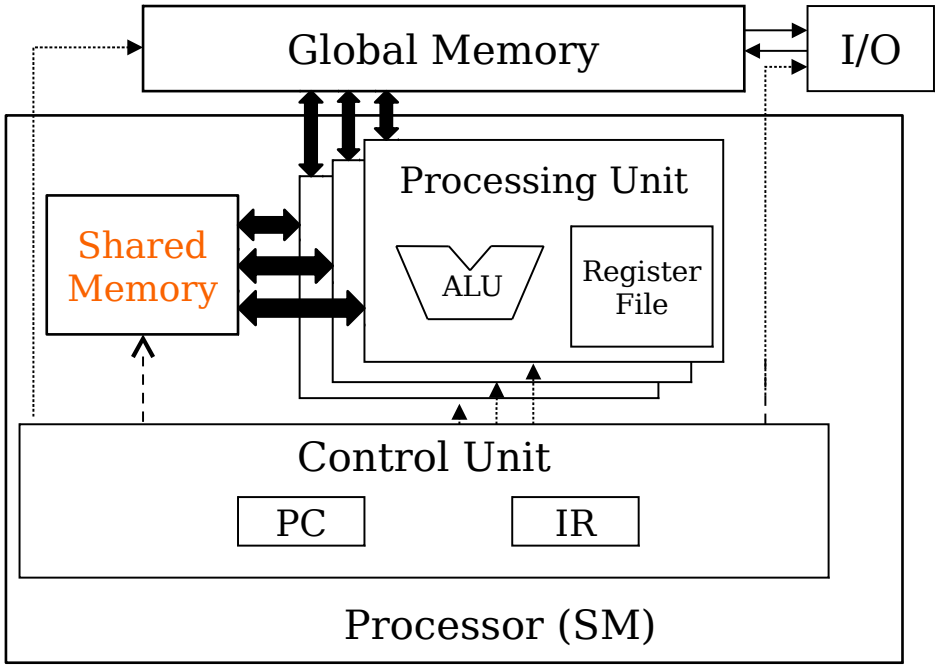
Constant Memory

A single read from constant memory can be broadcast to other “nearby” threads, effectively saving up to 15 reads (to each half-warp)

- If every thread in a half-warp requests data from the same address in constant memory, your GPU will generate only a single read request and subsequently broadcast the data to every thread

Constant memory is cached, so consecutive reads of the same address will not incur any additional memory traffic

Hardware View of CUDA Memories



Declaring CUDA Variables

Variable declaration	Memory	Scope	Lifetime
int LocalVar;	register	thread	kernel
__device__ __shared__ int SharedVar;	shared	block	kernel
__device__ int GlobalVar;	global	grid	application
__device__ __constant__ int ConstantVar;	constant	grid	application

Local variables reside in a register (Except per-thread arrays that reside in global memory)

Kernel Variables

Variable declaration	Memory	Scope	Lifetime
int LocalVar;	register	thread	**kernel**
__device__ __shared__ int SharedVar;	shared	block	**kernel**
__device__ int GlobalVar;	global	grid	application
__device__ __constant__ int ConstantVar;	constant	grid	application

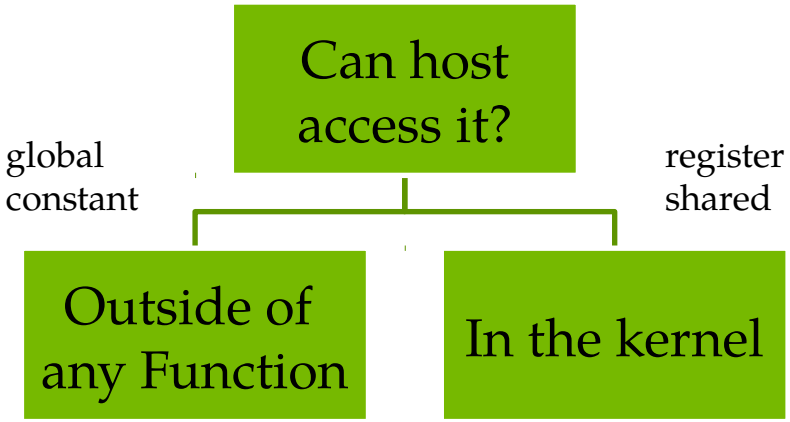
The variable must be declared within the kernel function body; and will be available only within the kernel code.

Application Variables

Variable declaration	Memory	Scope	Lifetime
int LocalVar;	register	thread	kernel
__device__ __shared__ int SharedVar;	shared	block	kernel
__device__ int GlobalVar;	global	grid	**application**
__device__ __constant__ int ConstantVar;	constant	grid	**application**

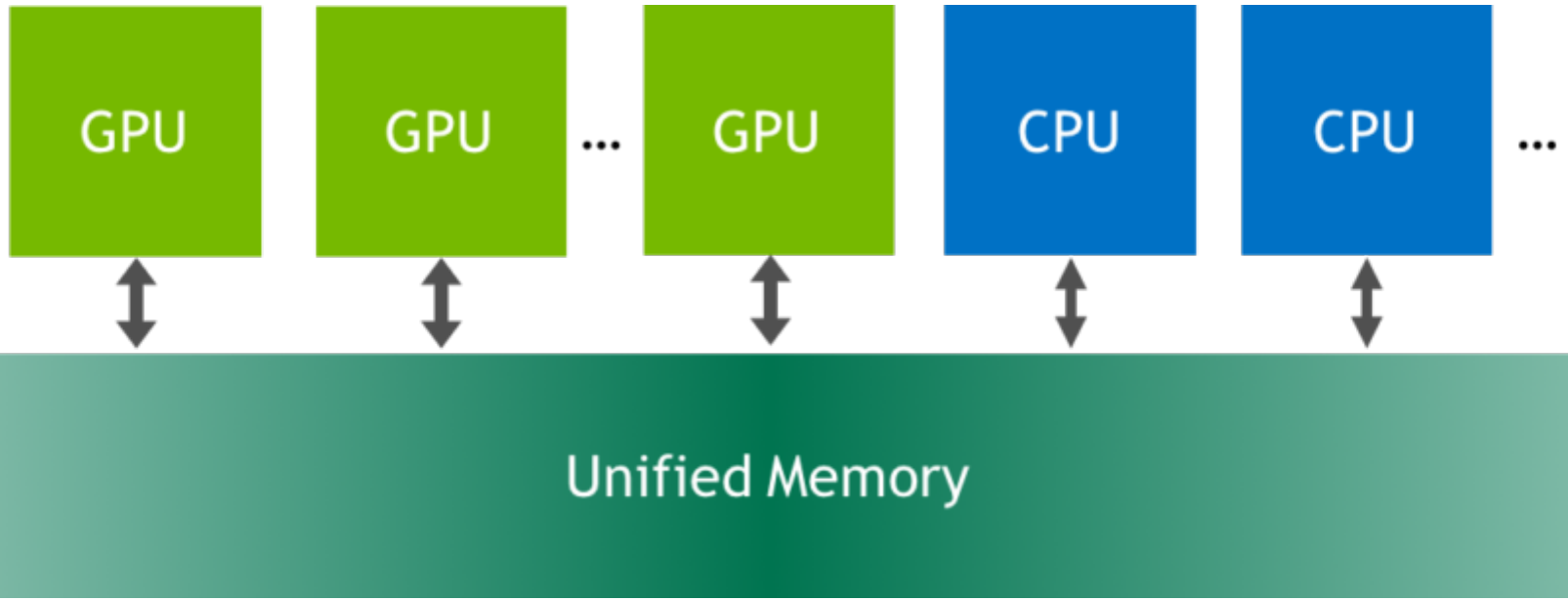
The variable must be declared outside of any function.

Where to Declare Variables?



Unified Memory

Single memory address space accessible from any processor in a system



Unified Memory

Hardware/software implementation

Combines the advantages of explicit copies and zero-copy access

GPU can access any page of the entire system memory and at the same time migrate the data on-demand to its own memory for high bandwidth access

Pages and page table entries may not be created until they are accessed by the GPU or the CPU

```
cudaError_t cudaMallocManaged(void** ptr, size_t size);
```

Shared Memory in CUDA

A special type of memory whose contents are explicitly defined and used in the kernel source code

- One in each SM
- Accessed at much higher speed (in both latency and throughput) than global memory
- Scope of access and sharing - thread blocks
- Lifetime – thread block, contents will disappear after the corresponding thread finishes terminates execution
- Accessed by memory load/store instructions
- A form of scratchpad memory in computer architecture

Tiling for Reduced Memory Traffic

The global memory is large but slow, whereas the shared memory is small but fast

Partition the data into subsets called tiles so that each tile fits into the shared memory