

Introduction to GPU Programming

Mubashir Adnan Qureshi

http://www.ncsa.illinois.edu/People/kindr/projects/hpca/files/singapore_p1.pdf

http://developer.download.nvidia.com/CUDA/training/NVIDIA_GPU_Computing_Webinars_CUDA_Memory_Optimization.pdf

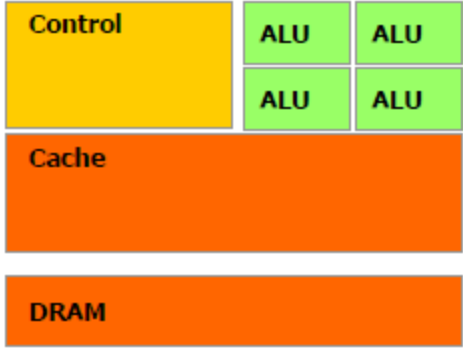
Tutorial Goals

- NVIDIA GPU architecture
- NVIDIA GPU application development flow
- Write and run simple NVIDIA GPU kernels in CUDA
- Be aware of performance limiting factors and understand performance tuning strategies

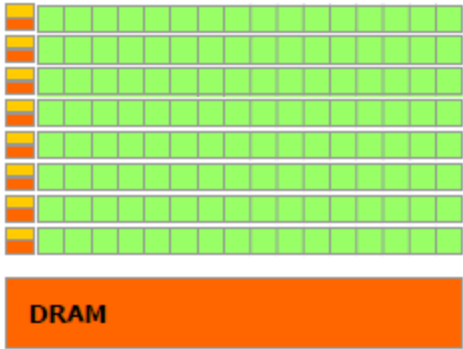
Introduction

- Why use Graphics Processing Units (GPUs) for general-purpose computing
- Modern GPU architecture
 - NVIDIA
- GPU programming overview
 - CUDA C
 - OpenCL

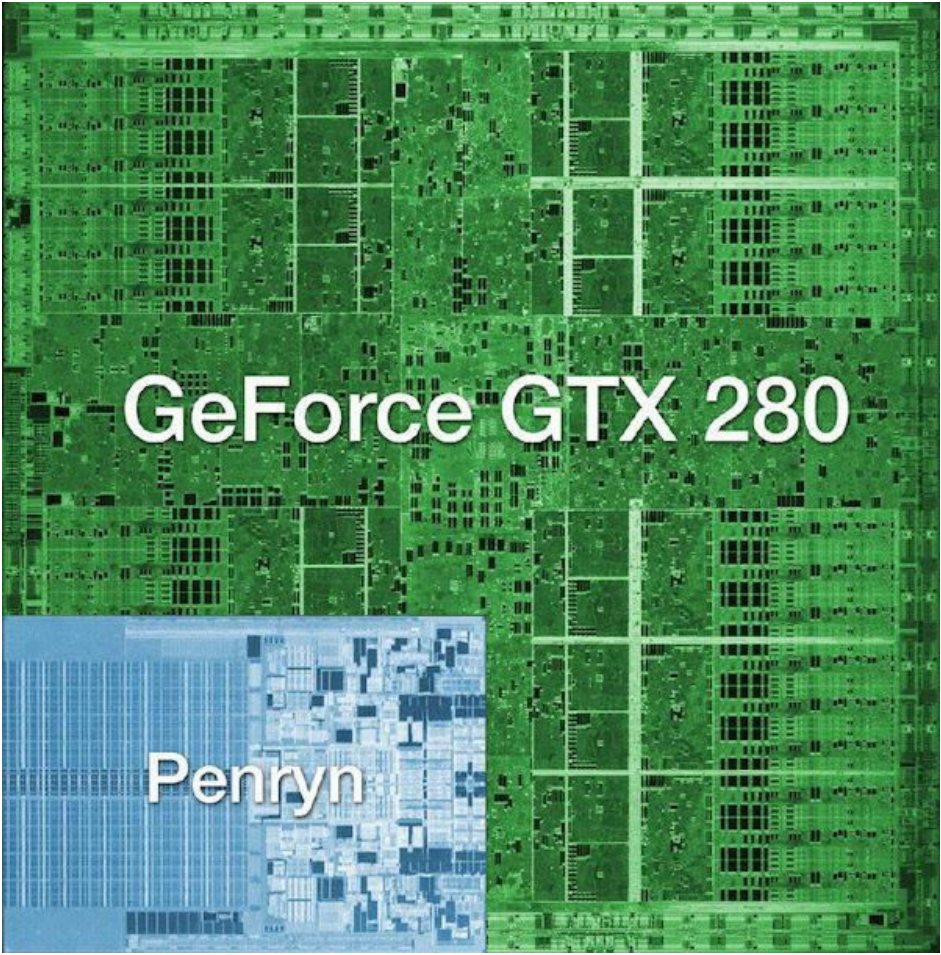
GPU vs. CPU Silicon Use



CPU

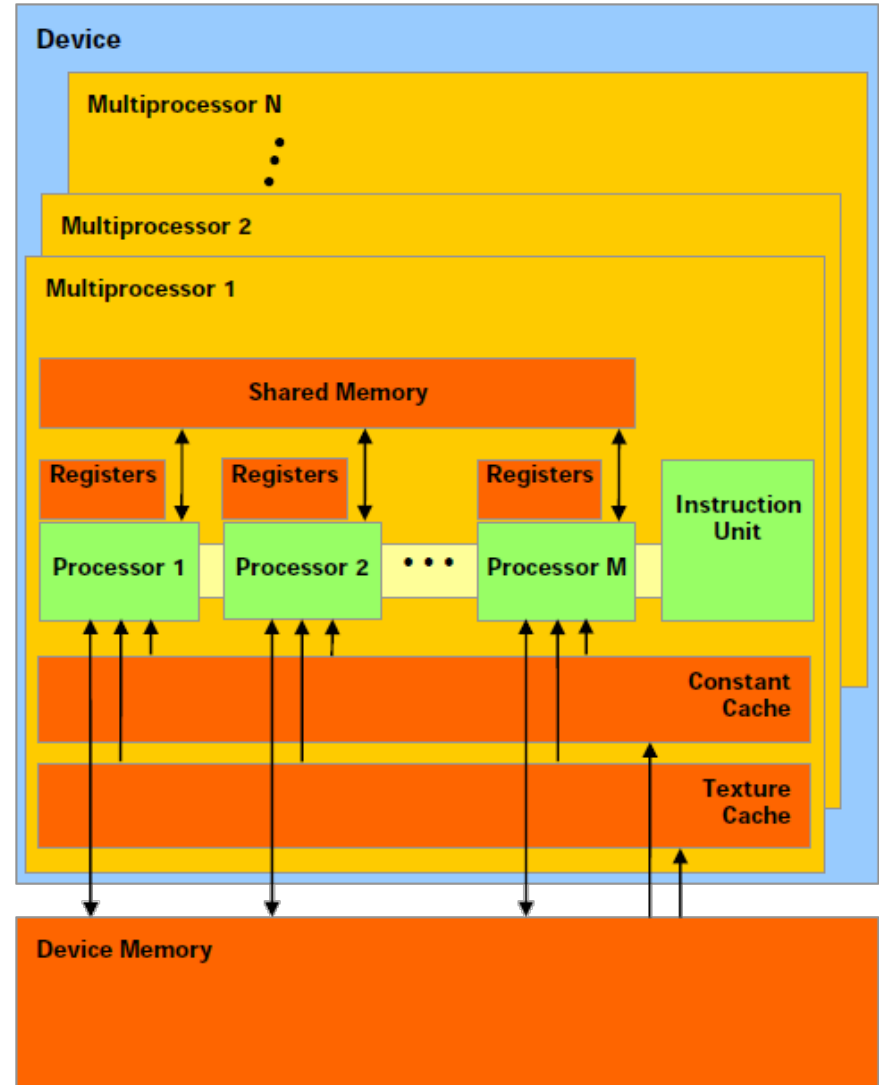


GPU

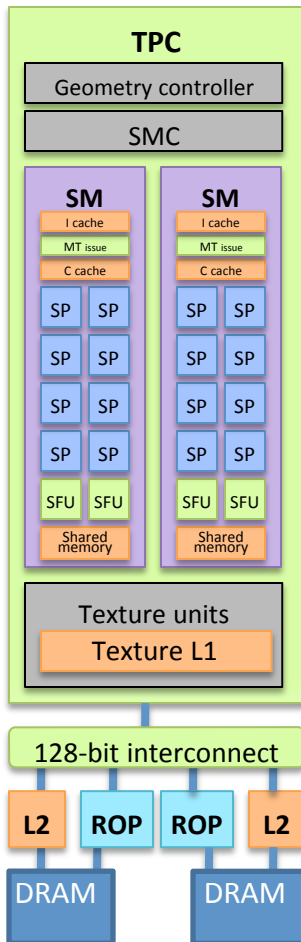


NVIDIA GPU Architecture

- N multiprocessors called SMs
 - Each has M cores called SPs
- SIMD
 - Same instruction executed on SPs
- Device memory shared across all SMs

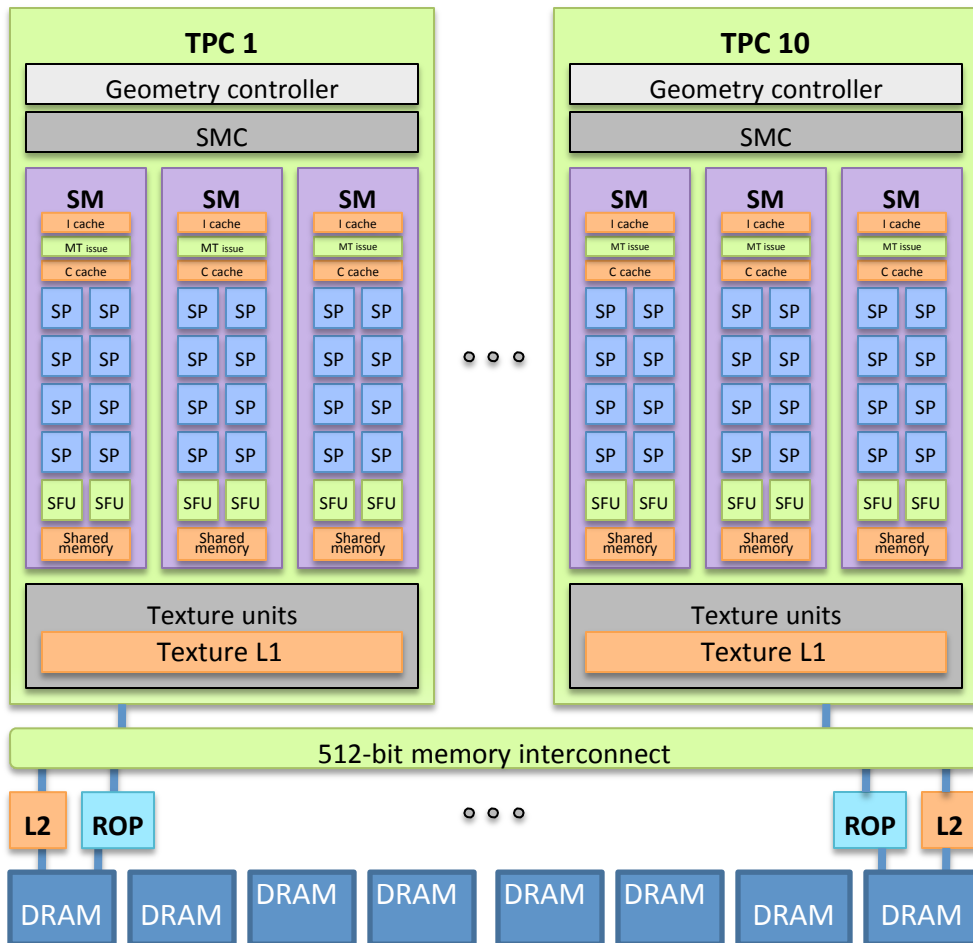


NVIDIA GeForce9400M G GPU



- 16 streaming processors arranged as 2 streaming multiprocessors
- At 0.8 GHz this provides
 - 54 GFLOPS in single-precision (SP)
- 128-bit interface to off-chip GDDR3 memory
 - 21 GB/s bandwidth

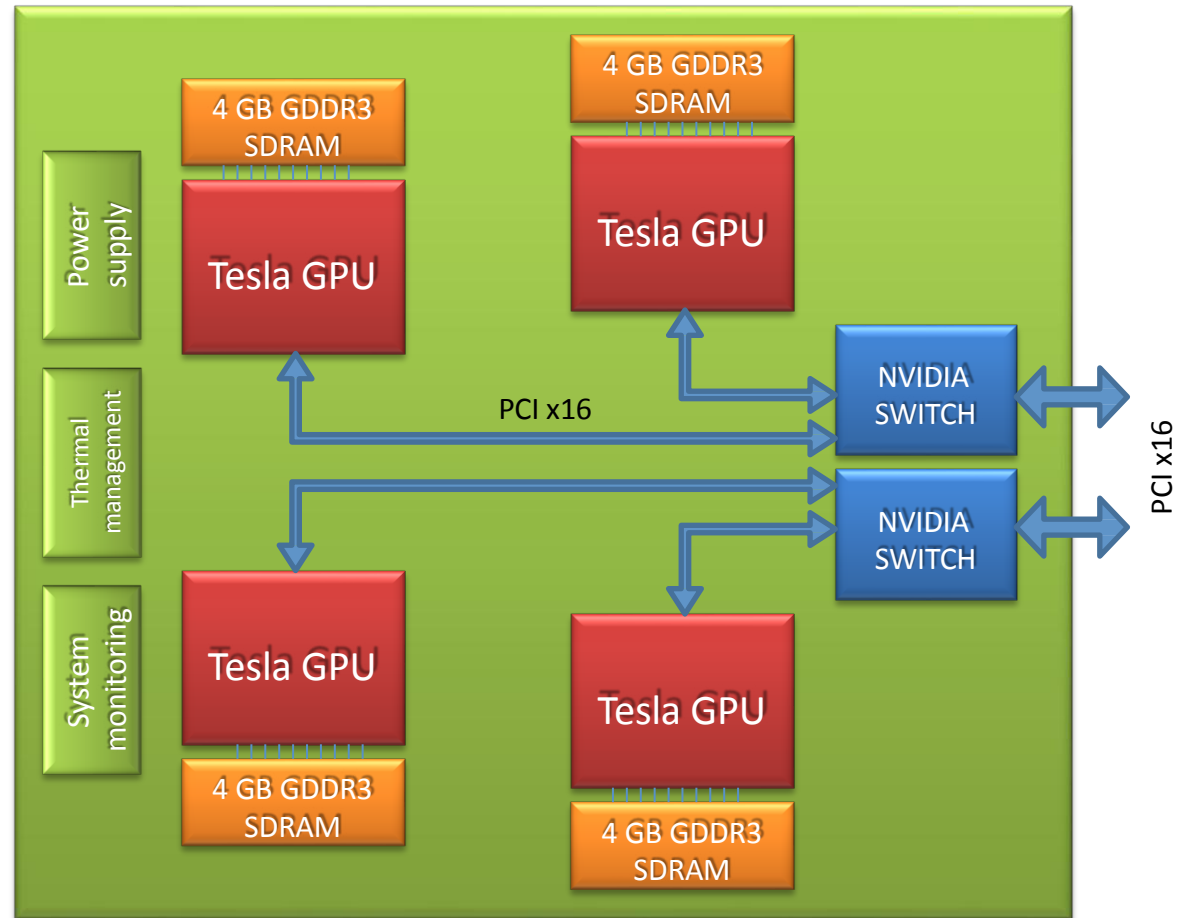
NVIDIA Tesla C1060 GPU



- 240 streaming processors arranged as 30 streaming multiprocessors
- At 1.3 GHz this provides
 - 1 TFLOPS SP
 - 86.4 GFLOPS DP
- 512-bit interface to off-chip GDDR3 memory
 - 102 GB/s bandwidth

NVIDIA Tesla S1070 Computing Server

- 4 T10 GPUs



GPU Use/Programming

- GPU libraries
 - NVIDIA's CUDA BLAS and FFT libraries
 - Many 3rd party libraries
- Low abstraction lightweight GPU programming toolkits
 - **CUDA C**
 - OpenCL

nvcc

- Any source file containing CUDA C language extensions must be compiled with nvcc
- nvcc is a compiler driver that invokes many other tools to accomplish the job
- Basic nvcc usage
 - `nvcc <filename>.cu [-o <executable>]`
 - Builds release mode
 - `nvcc -deviceemu <filename>.cu`
 - Builds device emulation mode (all code runs on CPU)
 - `nvprof <executable>`
 - Profiles the code

Anatomy of a GPU Application

- Host side
- Device side

Reference CPU Version

```
void vecAdd(int N, float* A, float* B, float* C)
{ for (int i = 0; i < N; i++) C[i] = A[i] + B[i];
}
```

← Computational kernel

```
int main(int argc, char **argv)
{
  int N = 16384; // default vector size
```

```
float *A = (float*)malloc(N * sizeof(float));
float *B = (float*)malloc(N * sizeof(float));
float *C = (float*)malloc(N * sizeof(float));
```

← Memory allocation

```
vecAdd(N, A, B, C); // call compute kernel
```

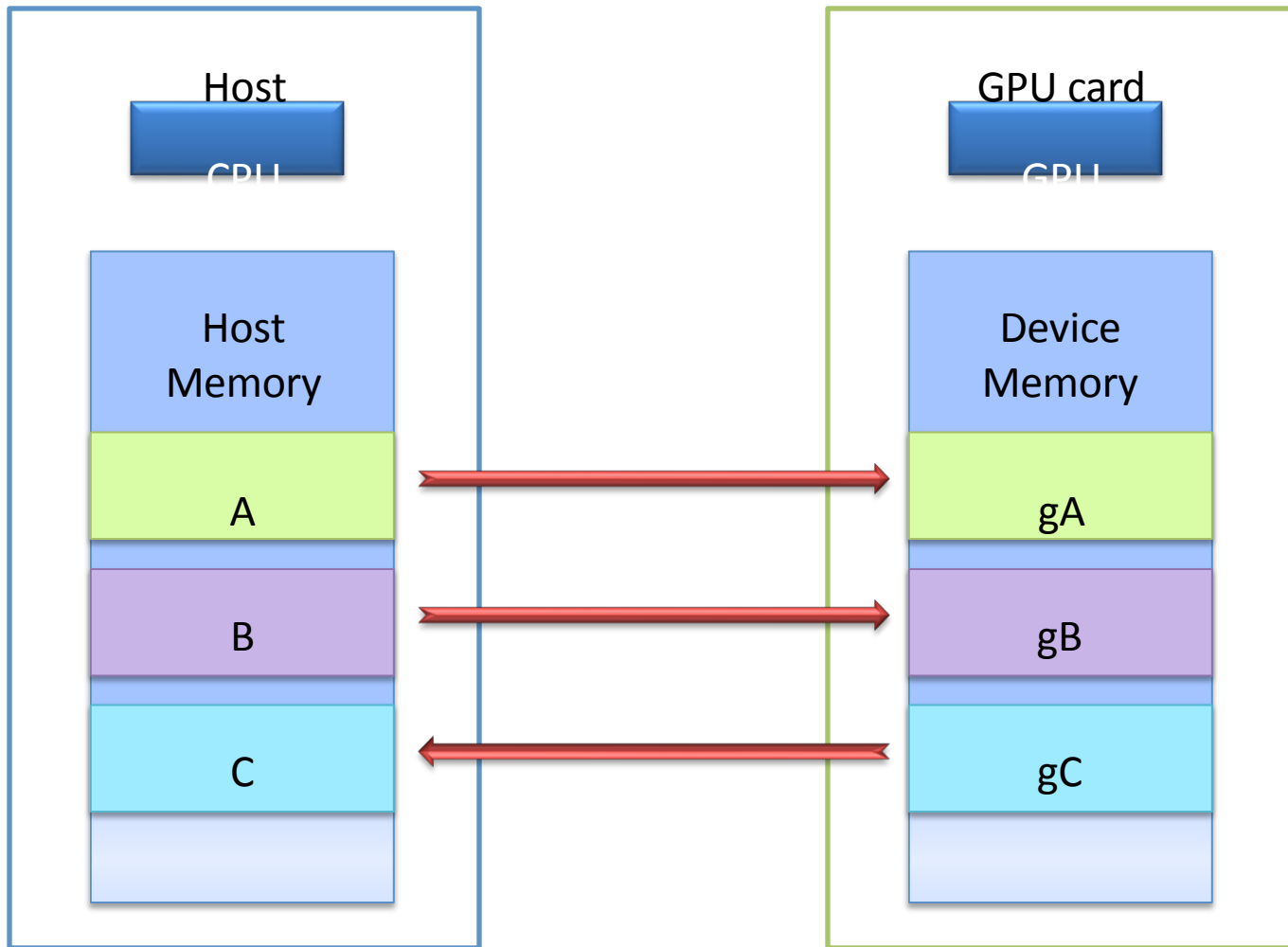
← Kernel invocation

```
free(A); free(B); free(C);
```

← Memory de-allocation

```
}
```

Adding GPU support



Memory Spaces

- **CPU and GPU have separate memory spaces**
 - Data is moved across PCIe bus
 - Use functions to allocate/set/copy memory on GPU
- **Host (CPU) manages device (GPU) memory**
 - `cudaMalloc(void** pointer, size_t nbytes)`
 - `cudaFree(void* pointer)`
 - `cudaMemcpy(void* dst, void* src, size_t nbytes, enum cudaMemcpyKind direction);`
 - returns after the copy is complete
 - blocks CPU thread until all bytes have been copied
 - does not start copying until previous CUDA calls complete
 - `enum cudaMemcpyKind`
 - `cudaMemcpyHostToDevice`
 - `cudaMemcpyDeviceToHost`
 - `cudaMemcpyDeviceToDevice`

Adding GPU support

```
int main(int argc, char **argv)
{
    int N = 16384; // default vector size

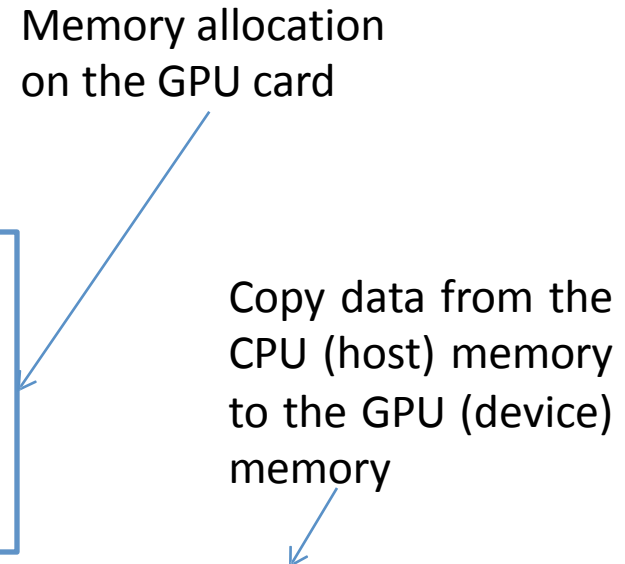
    float *A = (float*)malloc(N * sizeof(float));
    float *B = (float*)malloc(N * sizeof(float));
    float *C = (float*)malloc(N * sizeof(float));
```

```
float *devPtrA, *devPtrB, *devPtrC;
```

```
cudaMalloc((void**)&devPtrA, N * sizeof(float));
cudaMalloc((void**)&devPtrB, N * sizeof(float));
cudaMalloc((void**)&devPtrC, N * sizeof(float));
```

```
cudaMemcpy(devPtrA, A, N * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(devPtrB, B, N * sizeof(float), cudaMemcpyHostToDevice);
```

Memory allocation
on the GPU card



Copy data from the
CPU (host) memory
to the GPU (device)
memory

Adding GPU support

```
vecAdd<<<N/512, 512>>>(devPtrA, devPtrB, devPtrC);
```

Kernel invocation

```
cudaMemcpy(C, devPtrC, N * sizeof(float), cudaMemcpyDeviceToHost);
```

```
cudaFree(devPtrA);  
cudaFree(devPtrB);  
cudaFree(devPtrC);
```

Copy results from
device memory to
the host memory

```
free(A);  
free(B);  
free(C);
```

Device memory
de-allocation

```
}
```


GPU Kernel

- **CPU version**

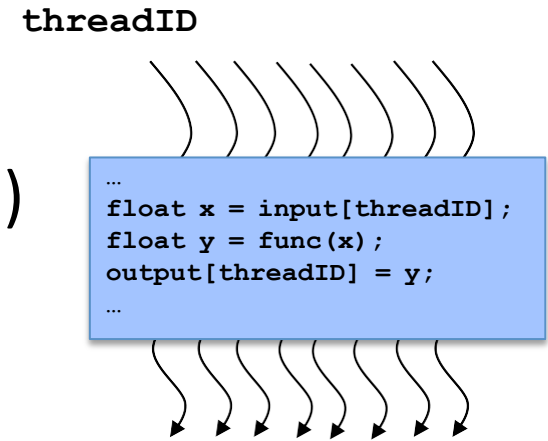
```
void vecAdd(int N, float* A, float* B, float* C)
{
    for (int i = 0; i < N; i++)
        C[i] = A[i] + B[i];
}
```

- **GPU version**

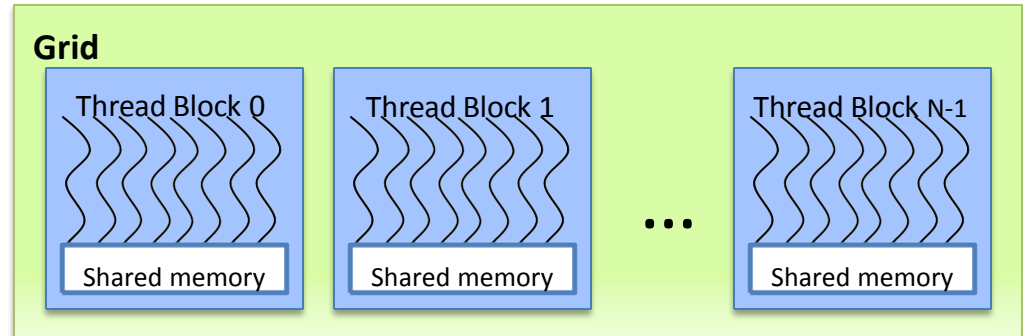
```
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    C[i] = A[i] + B[i];
}
```

CUDA Programming Model

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



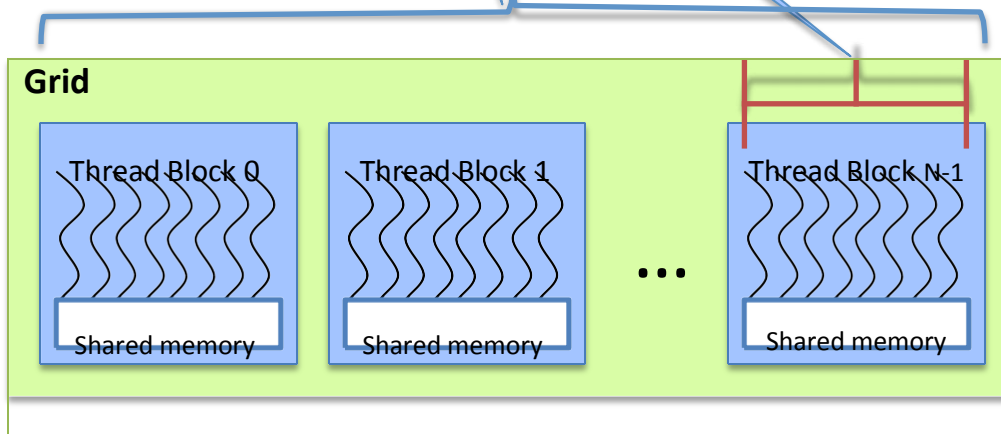
- Threads are arranged as a grid of thread blocks
 - Threads within a block have access to a segment of shared memory



Kernel Invocation Syntax

grid & thread block dimensionality

```
vecAdd<<<32, 512>>>(devPtrA, devPtrB, devPtrC);
```



```
int i = blockIdx.x * blockDim.x + threadIdx.x;
```

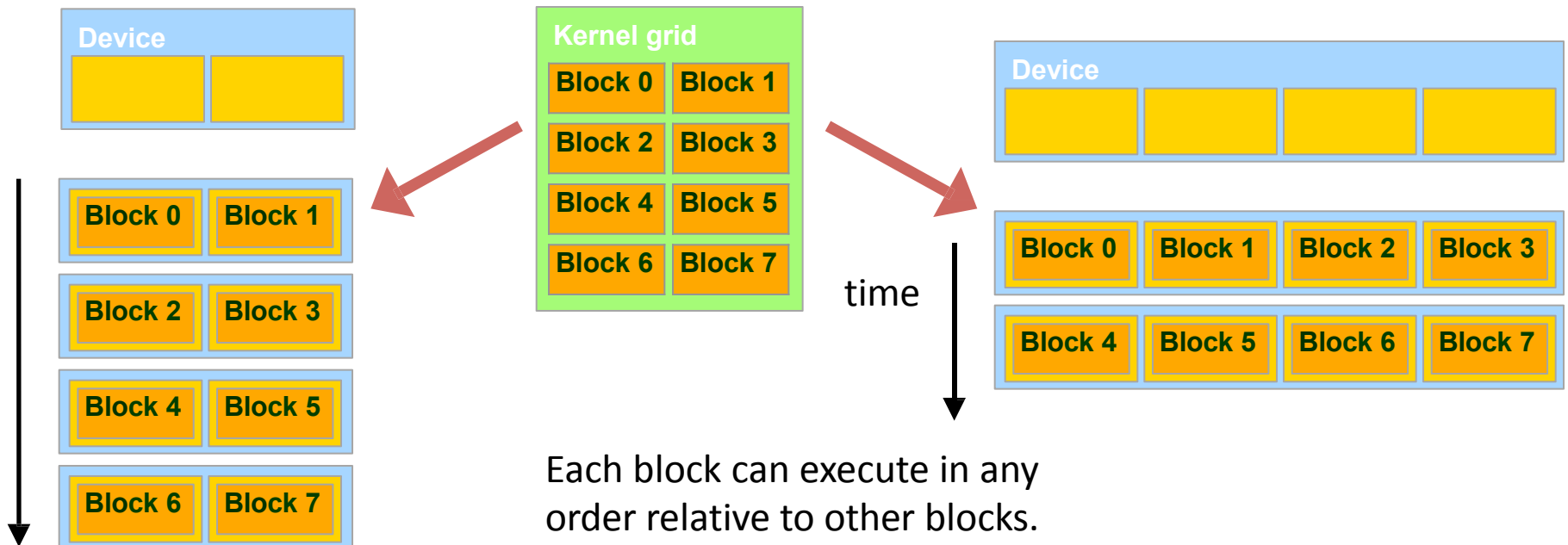
block ID within a grid

number of threads per block

thread ID within a thread block

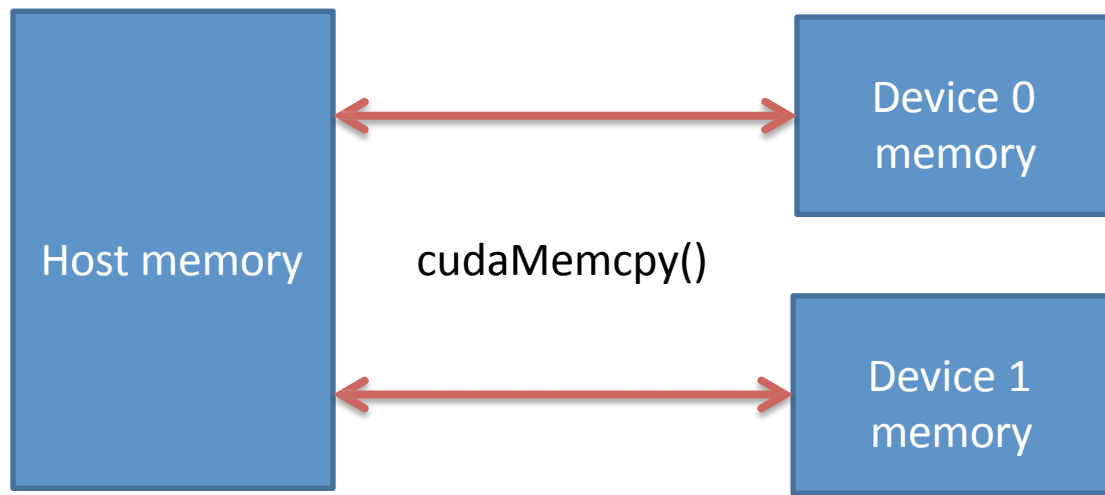
Mapping Threads to the Hardware

- Blocks of threads are transparently assigned to SMs
 - A block of threads executes on one SM & does not migrate
 - Several blocks can reside concurrently on one SM
- Blocks must be independent
 - Any possible interleaving of blocks should be valid
 - Blocks may coordinate but not synchronize
 - Thread blocks can run in any order



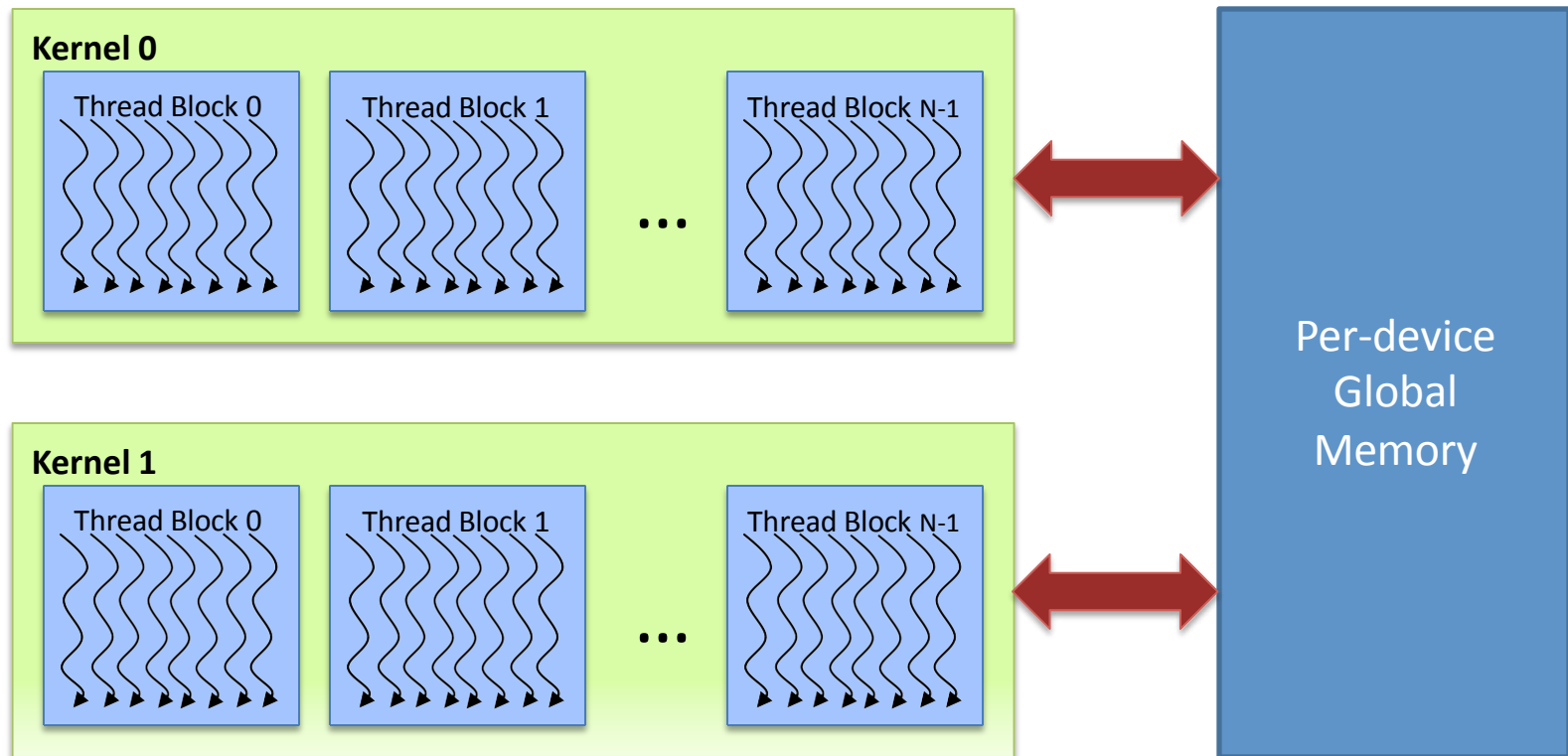
GPU Memory Hierarchy

- Global (device) memory
 - Accessible by all threads as well as host (CPU)
 - Data lifetime is from allocation to deallocation



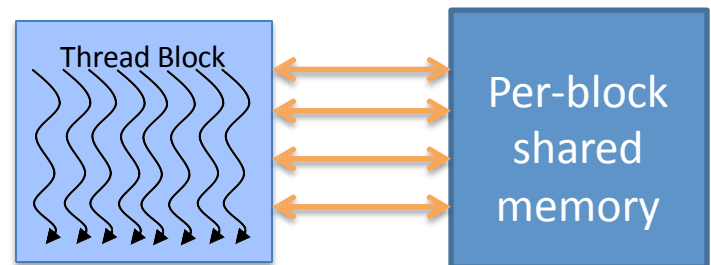
GPU Memory Hierarchy

- Global (device) memory

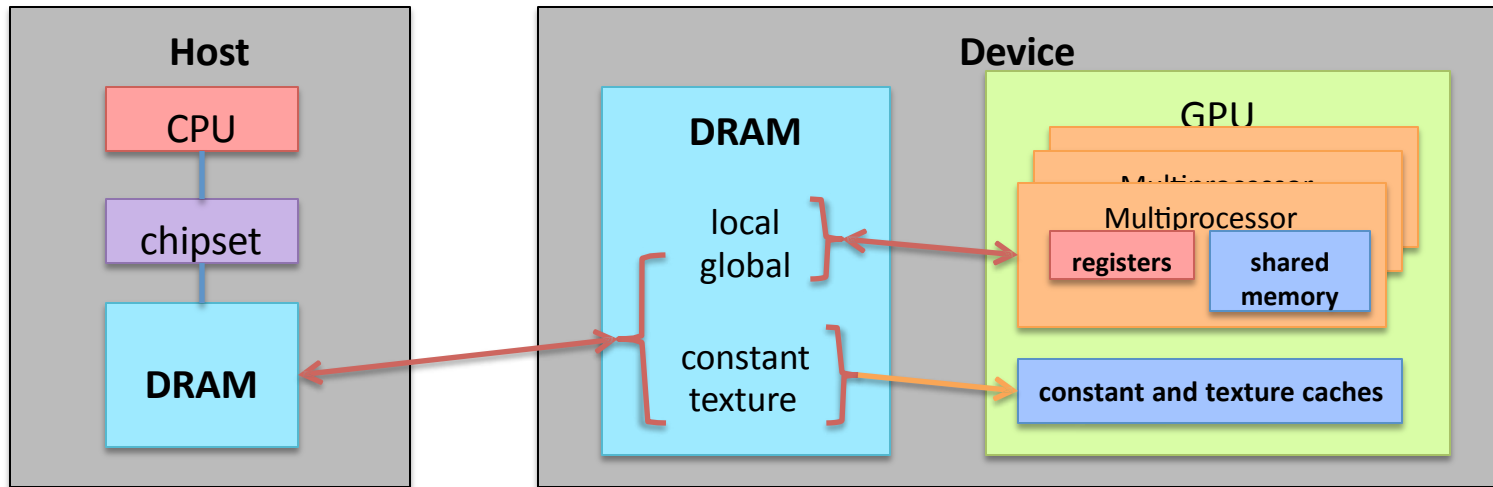


GPU Memory Hierarchy

- Local storage
 - Each thread has own local storage
 - Mostly registers (managed by the compiler)
 - Data lifetime = thread lifetime
- Shared memory
 - Each thread block has own shared memory
 - Accessible only by threads within that block
 - Data lifetime = block lifetime



GPU Memory Hierarchy



Memory	Location	Cached	Access	Scope	Lifetime
Register	On-chip	N/A	R/W	One thread	Thread
Local	Off-chip	No	R/W	One thread	Thread
Shared	On-chip	N/A	R/W	All threads in a block	Block
Global	Off-chip	No	R/W	All threads + host	Application
Constant	Off-chip	Yes	R	All threads + host	Application
Texture	Off-chip	Yes	R	All threads + host	Application

GPU Kernel

- **CPU version**

```
void vecAdd(int N, float* A, float* B, float* C)
{
    for (int i = 0; i < N; i++)
        C[i] = A[i] + B[i];
}
```

- **GPU version**

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

Optimizing Algorithms for GPUs

- Maximize independent parallelism
- Maximize arithmetic intensity (math/bandwidth)
- Sometimes it's better to recompute than to cache GPU
 - GPU spends its transistors on ALUs, not memory
- Do more computation on the GPU to avoid costly data transfers
 - Even low parallelism computations can sometimes be faster than transferring back and forth to host

Optimize Memory Access

- Coalesced vs. Non-coalesced = order of magnitude
 - Global/Local device memory
 - Contiguous threads accessing contiguous memory
- Shared Memory
 - Hundreds of times faster than global memory
 - Threads can cooperate via shared memory
 - Use it to avoid non-coalesced access