CS 179: GPU Computing

Lecture 2: The Basics

Recap

- Can use GPU to solve highly parallelizable problems
  - Performance benefits vs. CPU
- Straightforward extension to C language

```
__global__ void
cudaAddVectorsKernel(float * a, float * b, float * c) {
    //Decide an index somehow
    c[index] = a[index] + b[index];
}
```

Disclaimer

- Goal for Week 1:
  - Fast-paced introduction
  - “Know enough to be dangerous”
- We will fill in details later!

Our original problem…

- Add two arrays
  - A[] + B[] -> C[]

- Goal: Understand what's going on

CUDA code (first part)

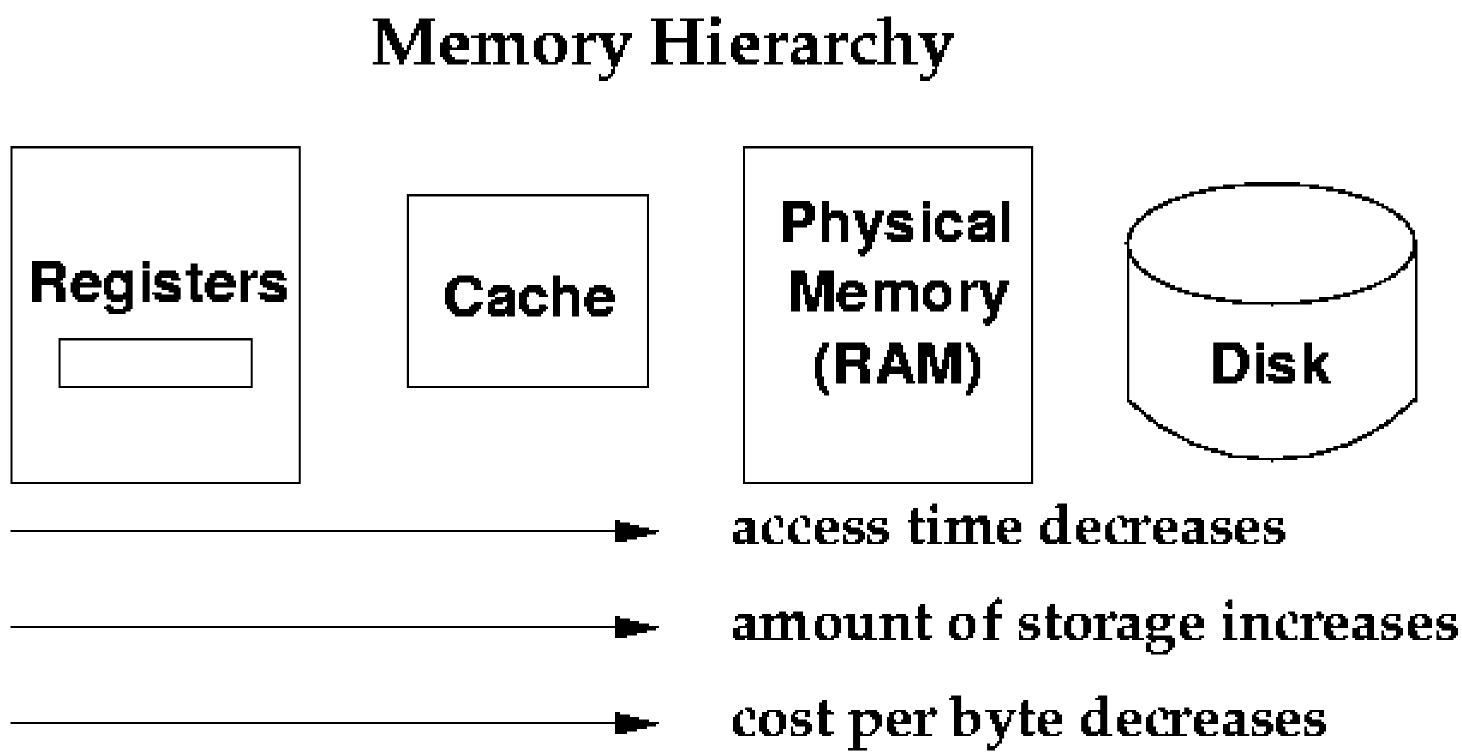
```
void cudaAddVectors(const float* a, const float* b, float* c, size){
    //For now, suppose a and b were created before calling this function

    // dev_a, dev_b (for inputs) and dev_c (for outputs) will be
    // arrays on the GPU.

    float * dev_a;
    float * dev_b;

    float * dev_c;

    // Allocate memory on the GPU for our inputs:
    cudaMalloc((void **) &dev_a, size*sizeof(float));
    cudaMemcpy(dev_a, a, size*sizeof(float), cudaMemcpyHostToDevice);

    cudaMalloc((void **) &dev_b, size*sizeof(float)); // and dev_b
    cudaMemcpy(dev_b, b, size*sizeof(float), cudaMemcpyHostToDevice);


    // Allocate memory on the GPu for our outputs:
    cudaMalloc((void **) &dev_c, size*sizeof(float));
```

Basic “formula”

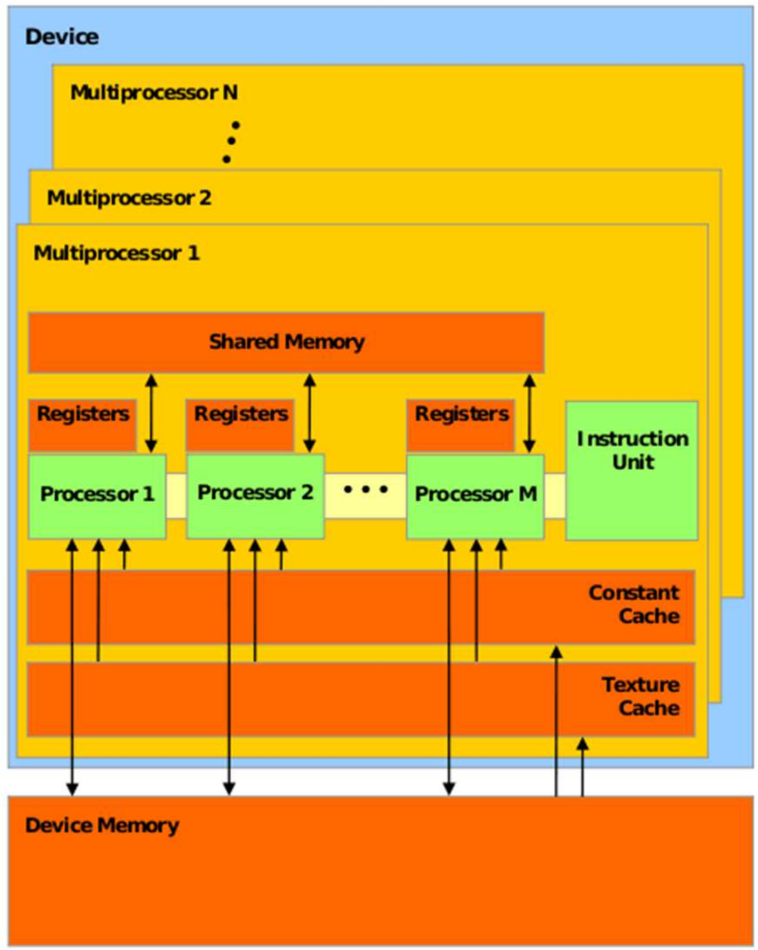
- Setup inputs on the host (CPU-accessible memory)
- Allocate memory for inputs on the GPU
- Copy inputs from host to GPU
- Allocate memory for outputs on the host
- Allocate memory for outputs on the GPU
- Start GPU kernel
- Copy output from GPU to host

```
void cudaAddVectors(const float* a, const float* b, float* c, size){
    //For now, suppose a and b were created before calling this function

    // dev_a, dev_b (for inputs) and dev_c (for outputs) will be
    // arrays on the GPU.

    float * dev_a;
    float * dev_b;

    float * dev_c;

    // Allocate memory on the GPU for our inputs:
    cudaMalloc((void **) &dev_a, size*sizeof(float));
    cudaMemcpy(dev_a, a, size*sizeof(float), cudaMemcpyHostToDevice);

    cudaMalloc((void **) &dev_b, size*sizeof(float)); // and dev_b
    cudaMemcpy(dev_b, b, size*sizeof(float), cudaMemcpyHostToDevice);


    // Allocate memory on the GPu for our outputs:
    cudaMalloc((void **) &dev_c, size*sizeof(float));
```

“Classic” Memory Hierarchy

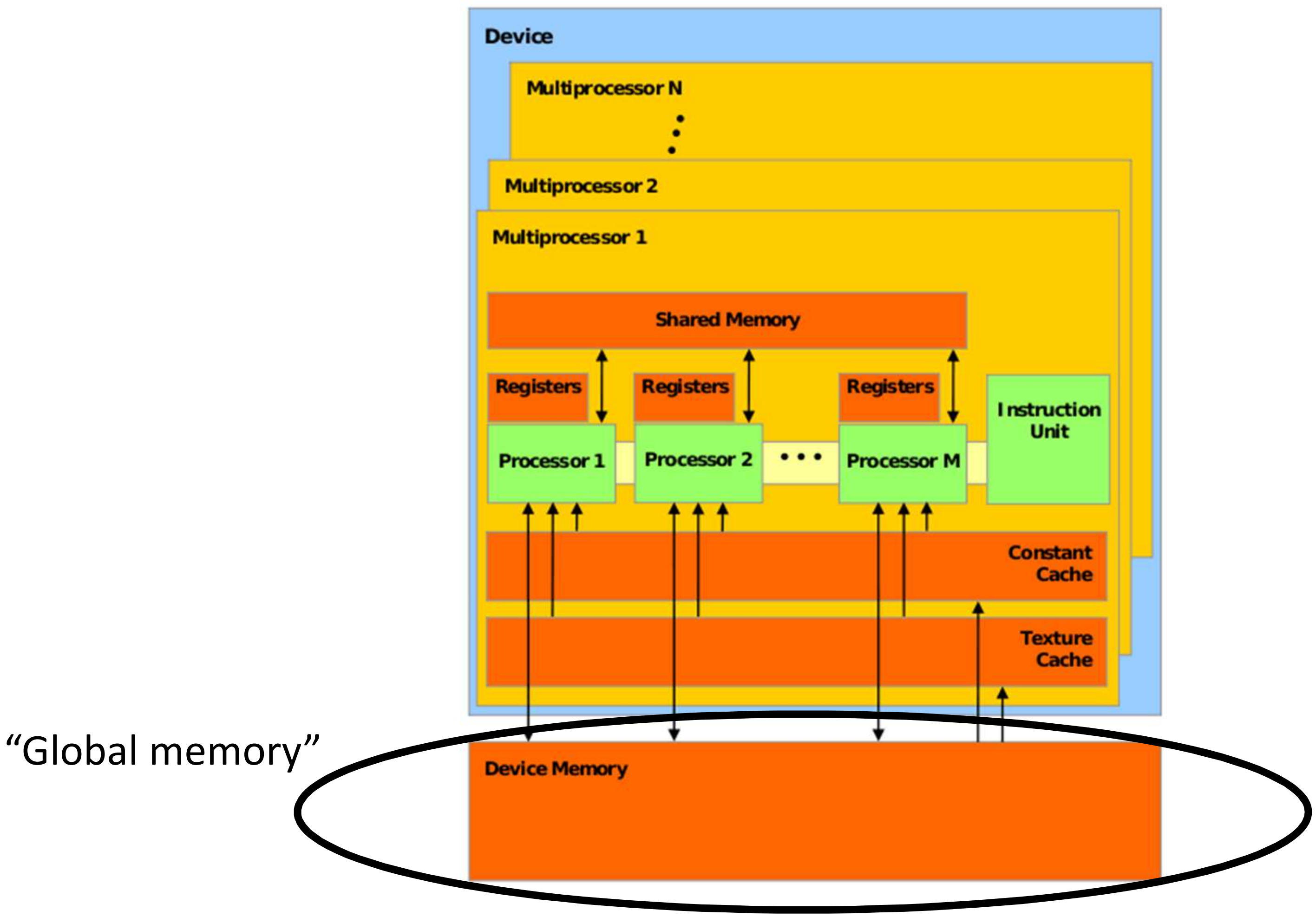


The GPU



Device
Multiprocessor N
Multiprocessor 2
Multiprocessor 1
Shared Memory
Registers
Registers
Registers
Instruction Unit
Processor 1
Processor 2
Processor M
Constant Cache
Texture Cache
Device Memory

The GPU


Device
Multiprocessor N
Multiprocessor 2
Multiprocessor 1
Shared Memory
Registers
Registers
Registers
Instruction Unit
Processor 1
Processor 2
Processor M
Constant Cache
Texture Cache
Device Memory
"Global memory"

```
void cudaAddVectors(const float* a, const float* b, float* c, size){
    //For now, suppose a and b were created before calling this function

    // dev_a, dev_b (for inputs) and dev_c (for outputs) will be
    // arrays on the GPU.

    float * dev_a;
    float * dev_b;

    float * dev_c;

    // Allocate memory on the GPU for our inputs:
    cudaMalloc((void **) &dev_a, size*sizeof(float));
    cudaMemcpy(dev_a, a, size*sizeof(float), cudaMemcpyHostToDevice);

    cudaMalloc((void **) &dev_b, size*sizeof(float)); // and dev_b
    cudaMemcpy(dev_b, b, size*sizeof(float), cudaMemcpyHostToDevice);


    // Allocate memory on the GPu for our outputs:
    cudaMalloc((void **) &dev_c, size*sizeof(float));
```

Pointers

- Difference between CPU and GPU pointers?

Pointers

- Difference between CPU and GPU pointers?
  - None – pointers are just addresses!

Pointers

- Difference between CPU and GPU pointers?
  - None – pointers are just addresses!
  - Up to the programmer to keep track!

Pointers

- Good practice:
  - Special naming conventions, e.g. “dev_” prefix

```
void cudaAddVectors(const float* a, const float* b, float* c, size){
    //For now, suppose a and b were created before calling this function

    // dev_a, dev_b (for inputs) and dev_c (for outputs) will be
    // arrays on the GPU.

    float * dev_a;
    float * dev_b;

    float * dev_c;

    // Allocate memory on the GPU for our inputs:
    cudaMalloc((void **) &dev_a, size*sizeof(float));
    cudaMemcpy(dev_a, a, size*sizeof(float), cudaMemcpyHostToDevice);

    cudaMalloc((void **) &dev_b, size*sizeof(float)); // and dev_b
    cudaMemcpy(dev_b, b, size*sizeof(float), cudaMemcpyHostToDevice);


    // Allocate memory on the GPu for our outputs:
    cudaMalloc((void **) &dev_c, size*sizeof(float));
```

```
void cudaAddVectors(const float* a, const float* b, float* c, size){
    //For now, suppose a and b were created before calling this function

    // dev_a, dev_b (for inputs) and dev_c (for outputs) will be
    // arrays on the GPU.

    float * dev_a;
    float * dev_b;

    float * dev_c;

    // Allocate memory on the GPU for our inputs:
    cudaMalloc((void **) &dev_a, size*sizeof(float));
    cudaMemcpy(dev_a, a, size*sizeof(float), cudaMemcpyHostToDevice);

    cudaMalloc((void **) &dev_b, size*sizeof(float)); // and dev_b
    cudaMemcpy(dev_b, b, size*sizeof(float), cudaMemcpyHostToDevice);


    // Allocate memory on the GPu for our outputs:
    cudaMalloc((void **) &dev_c, size*sizeof(float));
```

Memory allocational

- With the CPU (host memory)…

```
float *c = malloc(N * sizeof(float));
```

  - Attempts to allocate #bytes in argument

Memory allocation

- On the GPU (global memory):

```
float *dev_c;
cudaMalloc(&dev_c, N * sizeof(float));
```

- Signature:

```
cudaError_t cudaMalloc (void ** devPtr, size_t size)
```

  - Attempts to allocate #bytes in arg2
  - arg1 is the *pointer* to the pointer in GPU memory!
    - Passed into function for modification
    - Result after successful call: Memory allocated in location given by dev_c *on GPU*
  - Return value is error code, can be checked

```
void cudaAddVectors(const float* a, const float* b, float* c, size){
    //For now, suppose a and b were created before calling this function

    // dev_a, dev_b (for inputs) and dev_c (for outputs) will be
    // arrays on the GPU.

    float * dev_a;
    float * dev_b;

    float * dev_c;

    // Allocate memory on the GPU for our inputs:
    cudaMalloc((void **) &dev_a, size*sizeof(float));
    cudaMemcpy(dev_a, a, size*sizeof(float), cudaMemcpyHostToDevice);

    cudaMalloc((void **) &dev_b, size*sizeof(float)); // and dev_b
    cudaMemcpy(dev_b, b, size*sizeof(float), cudaMemcpyHostToDevice);


    // Allocate memory on the GPu for our outputs:
    cudaMalloc((void **) &dev_c, size*sizeof(float));
```

```
void cudaAddVectors(const float* a, const float* b, float* c, size){
    //For now, suppose a and b were created before calling this function

    // dev_a, dev_b (for inputs) and dev_c (for outputs) will be
    // arrays on the GPU.

    float * dev_a;
    float * dev_b;

    float * dev_c;

    // Allocate memory on the GPU for our inputs:
    cudaMalloc((void **) &dev_a, size*sizeof(float));
    cudaMemcpy(dev_a, a, size*sizeof(float), cudaMemcpyHostToDevice);

    cudaMalloc((void **) &dev_b, size*sizeof(float)); // and dev_b
    cudaMemcpy(dev_b, b, size*sizeof(float), cudaMemcpyHostToDevice);


    // Allocate memory on the GPu for our outputs:
    cudaMalloc((void **) &dev_c, size*sizeof(float));
```

Memory copying

- With the CPU (host memory)…

```
// pointers source,destination to memory regions
memcpy(destination, source, N);
```

- Signature:

```
void * memcpy (void * destination, const void * source, size_t num);
```

  – Copies *num* bytes from (area pointed to by) source to (area pointed to by) destination

Memory copying

- Versatile cudaMemcpy() equivalent
  - CPU -> GPU
  - GPU -> CPU
  - GPU -> GPU
  - CPU -> CPU

Memory copying

- Signature:

```
cudaError_t cudaMemcpy(void *destination, void *src, size_t count,
enum cudaMemcpyKind kind)
```

Memory copying

- Signature:

```
cudaError_t cudaMemcpy(void *destination, void *src, size_t count,
enum cudaMemcpyKind kind)
```

- Values:
  - cudaMemcpyHostToHost
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice

Memory copying

- Signature:

```
cudaError_t cudaMemcpy(void *destination, void *src, size_t count,
enum cudaMemcpyKind kind)
```

Determines treatment of dst. and src. as CPU or GPU addresses

- Values:
  - cudaMemcpyHostToHost
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice

```
void cudaAddVectors(const float* a, const float* b, float* c, size){
    //For now, suppose a and b were created before calling this function

    // dev_a, dev_b (for inputs) and dev_c (for outputs) will be
    // arrays on the GPU.

    float * dev_a;
    float * dev_b;

    float * dev_c;

    // Allocate memory on the GPU for our inputs:
    cudaMalloc((void **) &dev_a, size*sizeof(float));
    cudaMemcpy(dev_a, a, size*sizeof(float), cudaMemcpyHostToDevice);

    cudaMalloc((void **) &dev_b, size*sizeof(float)); // and dev_b
    cudaMemcpy(dev_b, b, size*sizeof(float), cudaMemcpyHostToDevice);


    // Allocate memory on the GPu for our outputs:
    cudaMalloc((void **) &dev_c, size*sizeof(float));
```

Summary of memory

- CPU vs. GPU pointers
- cudaMalloc()
- cudaMemcpy()

```
    //At lowest, should be 32
    //Limit of 512 (Tesla), 1024 (newer)
    const unsigned int threadsPerBlock = 512;

    //How many blocks we'll end up needing
    const unsigned int blocks = ceil(size/float(threadsPerBlock));

    //Call the kernel!
    cudaAddVectorsKernel<<<blocks, threadsPerBlock>>>
        (dev_a, dev_b, dev_c);

    //Copy output from device to host (assume here that host memory
    //for the output has been calculated)

    cudaMemcpy(c, dev_c, size*sizeof(float), cudaMemcpyDeviceToHost);

    //Free GPU memory
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);
}
```

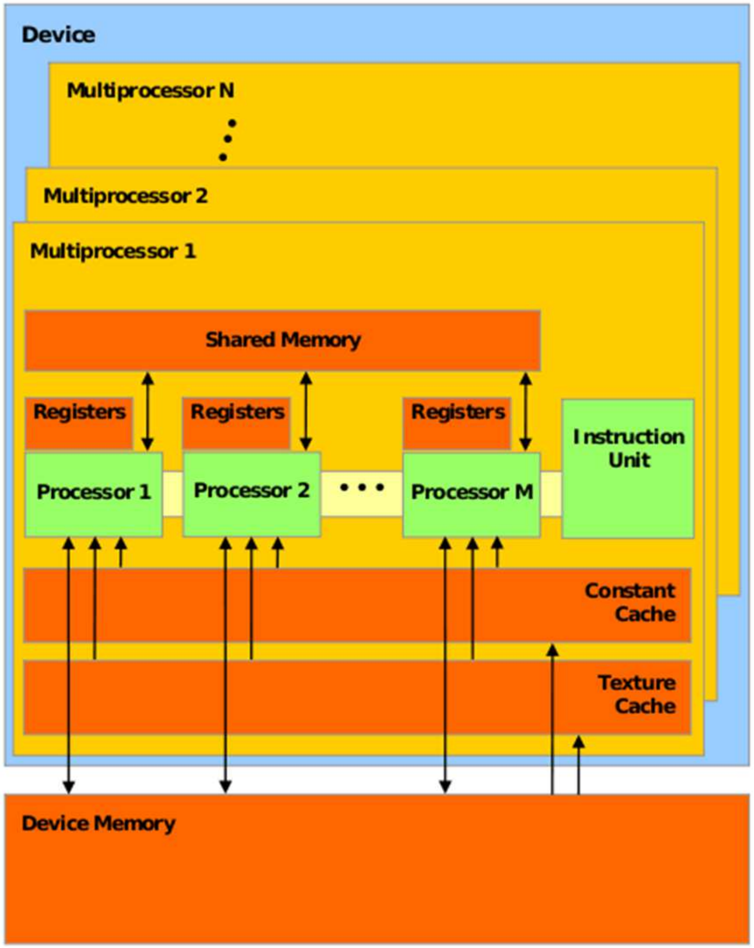
Part 2

```
    //At lowest, should be 32
    //Limit of 512 (Tesla), 1024 (newer)
    const unsigned int threadsPerBlock = 512;

    //How many blocks we'll end up needing
    const unsigned int blocks = ceil(size/float(threadsPerBlock));

    //Call the kernel!
    cudaAddVectorsKernel<<<blocks, threadsPerBlock>>>
        (dev_a, dev_b, dev_c);

    //Copy output from device to host (assume here that host memory
    //for the output has been calculated)

    cudaMemcpy(c, dev_c, size*sizeof(float), cudaMemcpyDeviceToHost);

    //Free GPU memory
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);
}
```

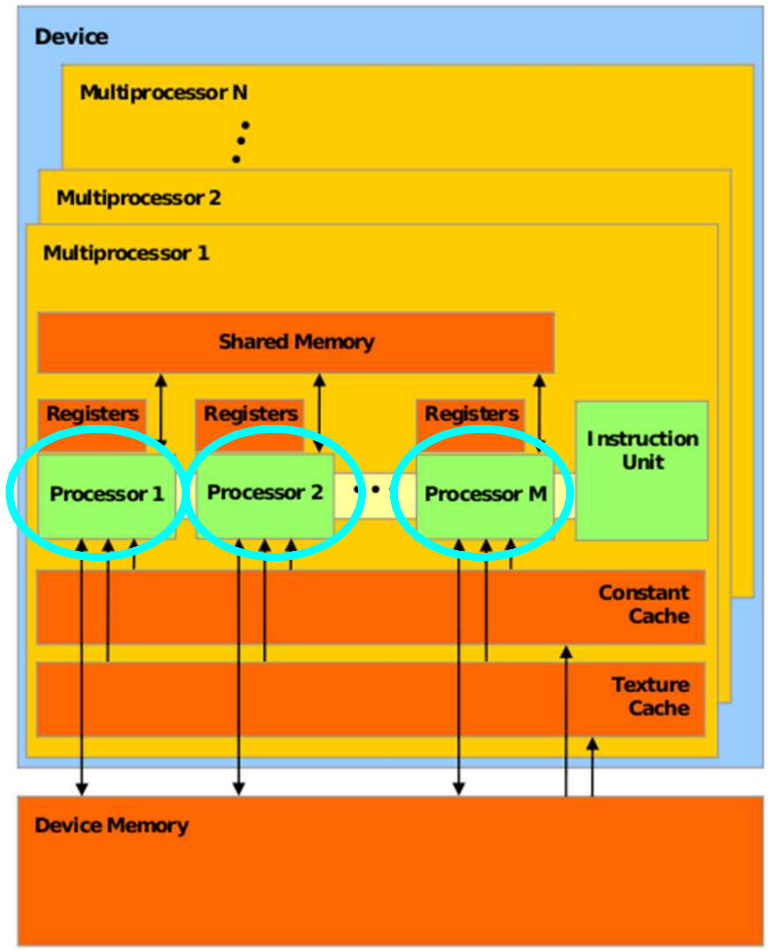
Recall…

- GPUs…
  - Have lots of cores
  - Are suited toward “parallel problems”

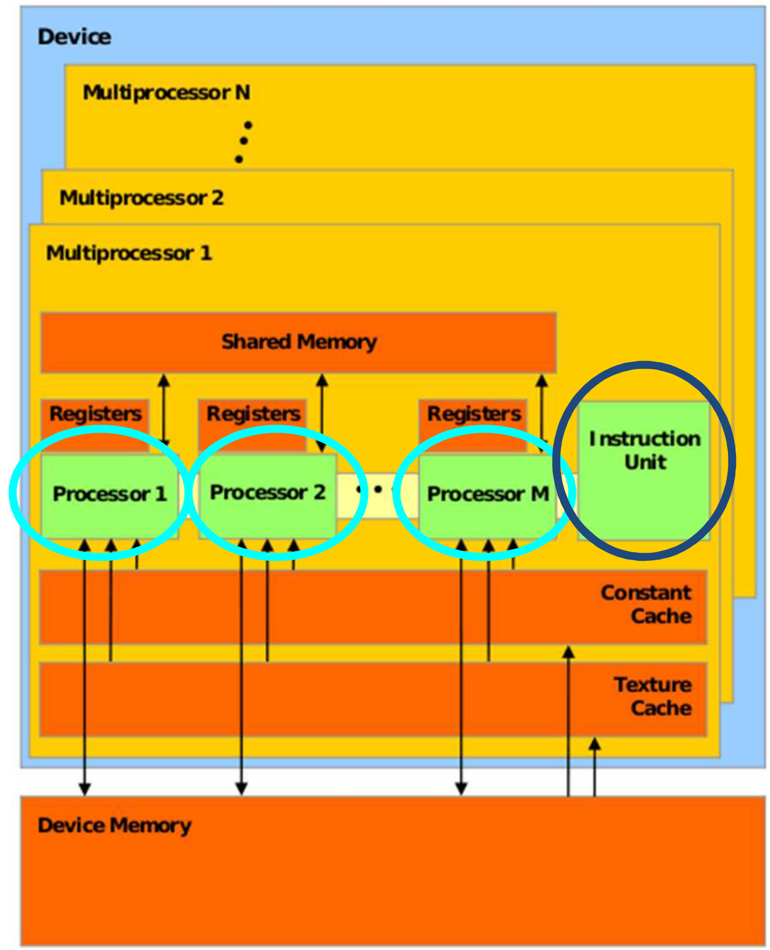
GPU internals



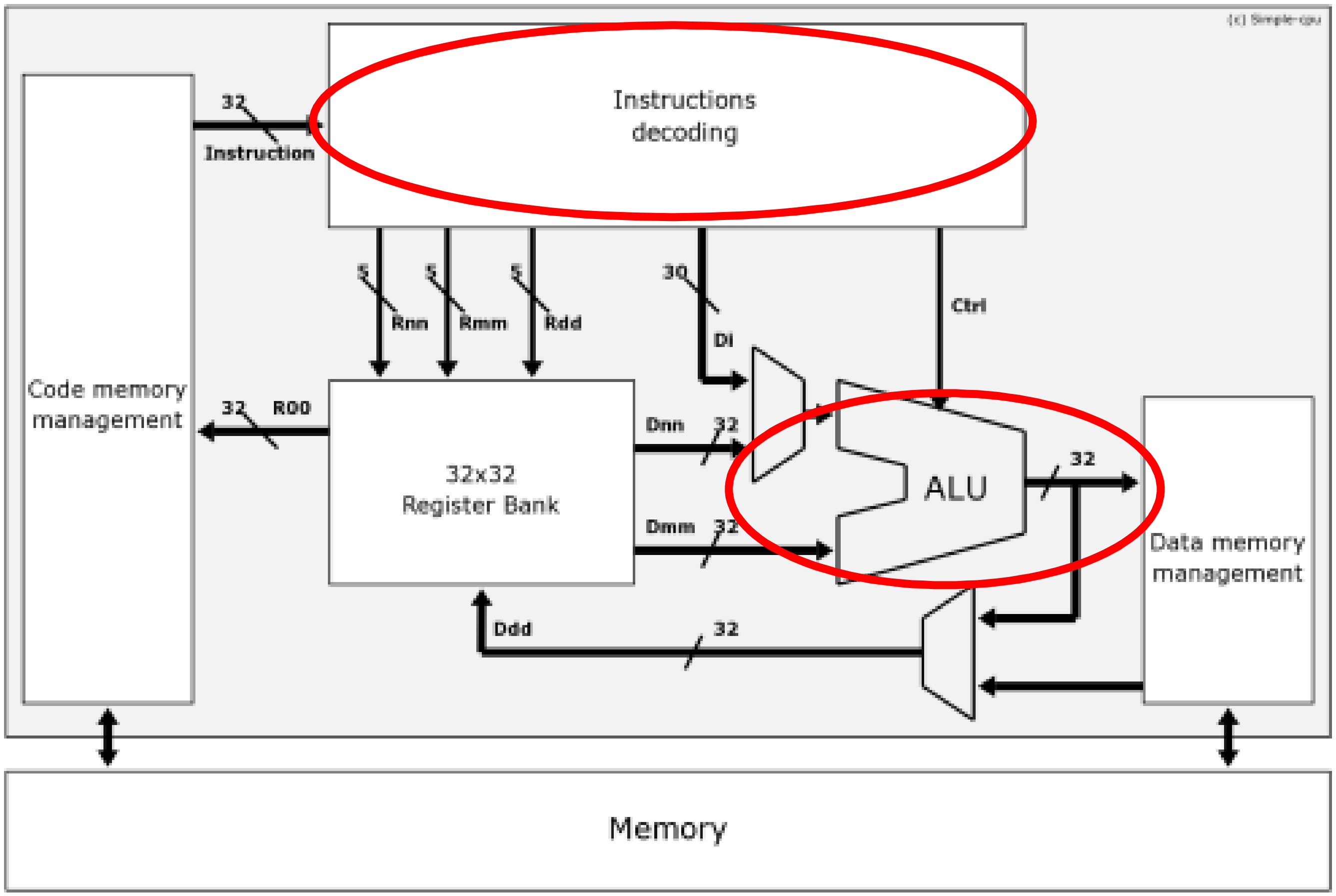
GPU internals



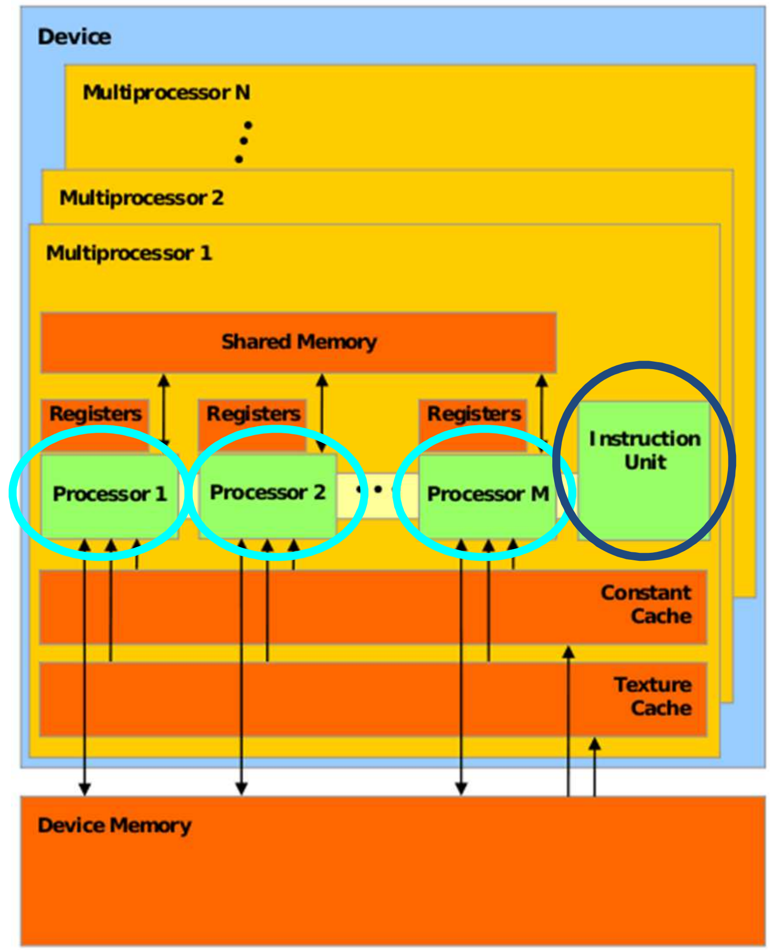
GPU internals



CPU internals



GPU internals

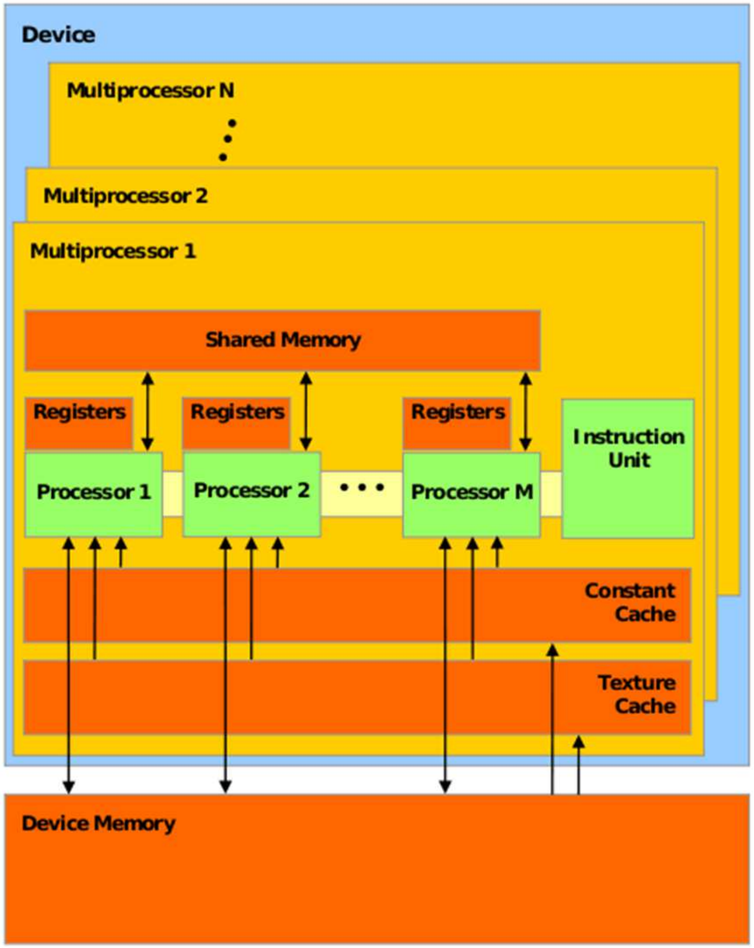


One instruction unit for multiple cores!

Warps

- Groups of threads simultaneously execute same instructions!
  - Called a “warp”
  - (32 threads in a warp under current standards)

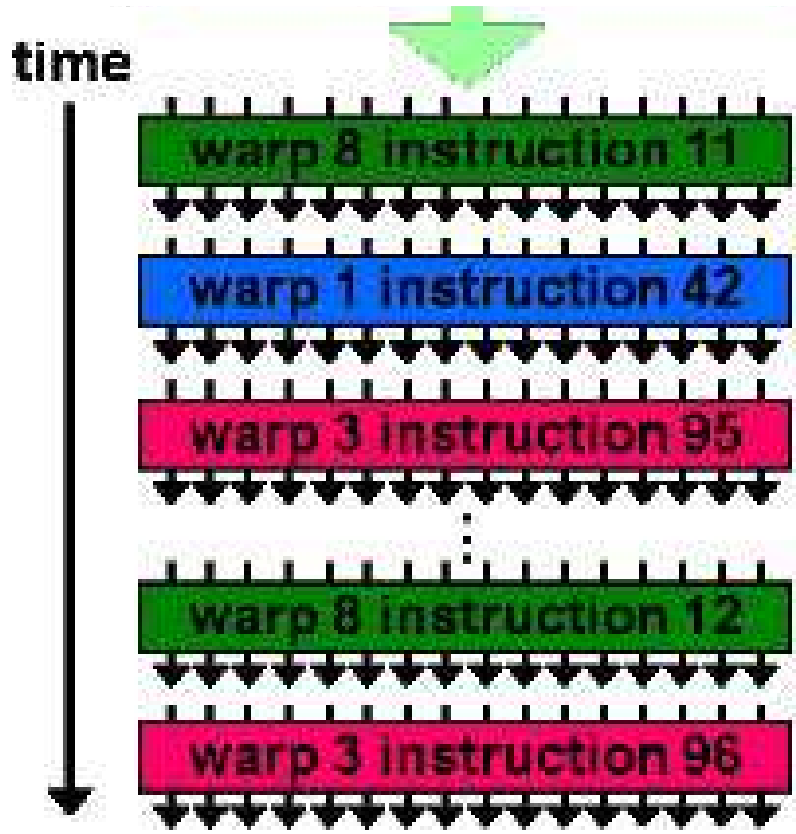
GPU internals



Blocks

- Group of threads scheduled to a multiprocessor
  - Contain multiple warps
  - Has a max. number (varies by GPU, e.g. 512 or 1024)

Multiprocessor execution timeline



Thread groups

- A *grid* (all the threads started…):
  - …contains *blocks* <- assigned to multiprocessors
    - Each *block* contains *warps* <- executed simultaneously
      - Each *warp* contains individual threads

Part 2

```
    //At lowest, should be 32
    //Limit of 512 (Tesla), 1024 (newer)
    const unsigned int threadsPerBlock = 512;

    //How many blocks we'll end up needing
    const unsigned int blocks = ceil(size/float(threadsPerBlock));

    //Call the kernel!
    cudaAddVectorsKernel<<<blocks, threadsPerBlock>>>
        (dev_a, dev_b, dev_c);

    //Copy output from device to host (assume here that host memory
    //for the output has been calculated)

    cudaMemcpy(c, dev_c, size*sizeof(float), cudaMemcpyDeviceToHost);

    //Free GPU memory
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);
}
```

- Moral 1: (from Lecture 1)
  - Start lots of threads!
    - Recall: Low context switch penalty
    - Hide latency
  - Start enough blocks!
    - Occupy SMs
  - e.g. Don't call:

```
kernel<<<1,1>>>();  // 1 block, 1 thread per block
```

  - Call:

```
kernel<<<50,512>>>();// 50 blocks, 512 threads per block
```

- Moral 2:
  - Multiprocessors execute warps (of 32 threads)
    - Block sizes of 32*n (integer n) are best
  - e.g. Don't call:

```
kernel<<<50,97>>>();  // 50 blocks, 97 threads per block
```

  - Call:

```
kernel<<<50,128>>>();// 50 blocks, 128 threads per block
```

Summary (processor internals)

- Key parameters on kernel call:
  - Threads per block
  - Number of blocks

- Choose carefully!

```
    //At lowest, should be 32
    //Limit of 512 (Tesla), 1024 (newer)
    const unsigned int threadsPerBlock = 512;

    //How many blocks we'll end up needing
    const unsigned int blocks = ceil(size/float(threadsPerBlock));

    //Call the kernel!
    cudaAddVectorsKernel<<<blocks, threadsPerBlock>>>
        (dev_a, dev_b, dev_c);

    //Copy output from device to host (assume here that host memory
    //for the output has been calculated)

    cudaMemcpy(c, dev_c, size*sizeof(float), cudaMemcpyDeviceToHost);

    //Free GPU memory
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);
}
```

Kernel argument passing

- Similar to arg-passing in C functions
- Some rules:
  - Don’t pass host-memory pointers
  - Small variables (e.g. individual ints) are fine
  - No pass-by-reference

Kernel function

- Executed by *many* threads
- Threads have unique ID mechanism:
  - Thread index within block
  - Block index

```
__global__ void
cudaAddVectorsKernel(float * a, float * b, float * c) {
    unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
    c[index] = a[index] + b[index];
}
```

- Out of bounds issue:
  - If index > (#elements), illegal access!

```
__global__ void
cudaAddVectorsKernel(float * a, float * b, float * c) {
    unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
    c[index] = a[index] + b[index];
}
```

- Out of bounds issue:
  - If index > (#elements), illegal access!

```
__global__ void
cudaAddVectorsKernel(float * a, float * b, float * c, int size) {
    unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index < size) {
        c[index] = a[index] + b[index];
    }
}
```

- #Threads issue:
  - Cannot start e.g. 1e9 threads!
  - Threads should handle arbitrary # of elements

```
__global__ void
cudaAddVectorsKernel(float * a, float * b, float * c, int size) {
    unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index < size) {
        c[index] = a[index] + b[index];
    }
}
```

- #Threads issue:
  - Cannot start e.g. 1e9 threads!
  - Threads should handle arbitrary # of elements

```
__global__ void
cudaAddVectorsKernel(float * a, float * b, float * c, int size) {
    unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
    while (index < size) {
        c[index] = a[index] + b[index];
        index += blockDim.x * gridDim.x;
    }
}
```

- #Threads issue:
  - Cannot start e.g. 1e9 threads!
  - Threads should handle arbitrary # of elements

```
__global__ void
cudaAddVectorsKernel(float * a, float * b, float * c, int size) {
    unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
    while (index < size) {
        c[index] = a[index] + b[index];
        index += blockDim.x * gridDim.x;
    }
}
```

Total number of blocks

```
    //At lowest, should be 32
    //Limit of 512 (Tesla), 1024 (newer)
    const unsigned int threadsPerBlock = 512;

    //How many blocks we'll end up needing
    const unsigned int blocks = ceil(size/float(threadsPerBlock));

    //Call the kernel!
    cudaAddVectorsKernel<<<blocks, threadsPerBlock>>>
        (dev_a, dev_b, dev_c);

    //Copy output from device to host (assume here that host memory
    //for the output has been calculated)

    cudaMemcpy(c, dev_c, size*sizeof(float), cudaMemcpyDeviceToHost);

    //Free GPU memory
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);
}
```

GPU -> CPU

Host memory pointer
(copy *to* here)

Device memory pointer
(copy *from* here)

```
    //At lowest, should be 32
    //Limit of 512 (Tesla), 1024 (newer)
    const unsigned int threadsPerBlock = 512;

    //How many blocks we'll end up needing
    const unsigned int blocks = ceil(size/float(threadsPerBlock));

    //Call the kernel!
    cudaAddVectorsKernel<<<blocks, threadsPerBlock>>>
        (dev_a, dev_b, dev_c);

    //Copy output from device to host (assume here that host memory
    //for the output has been calculated)

    cudaMemcpy(c, dev_c, size*sizeof(float), cudaMemcpyDeviceToHost);

    //Free GPU memory
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);
}
```

- cudaFree()
  - Equivalent to host memory’s free() function
  - (As on host) Free memory after completion!

Summary

- GPU global memory:
  - Pointers (CPU vs GPU)
  - cudaMalloc() and cudaMemcpy()
- GPU processor details:
  - Thread group hierarchy
  - Launch parameters
- Threads in kernel