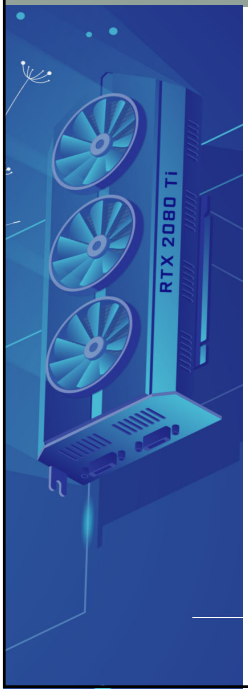
Computer Structure and Language

Hamid Sarbazi-Azad
Department of Computer Engineering
Sharif University of Technology (SUT)
Tehran, Iran

1



Introduction to GPU & CUDA

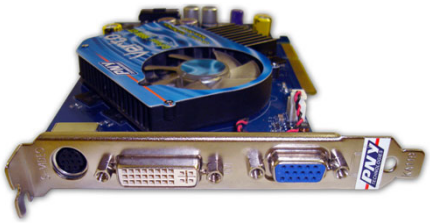
2

GPU Architecture

3

Graphic Cards/ History

- 1980's – No GPUs. Just VGA controller
- 1990's – Add more function into VGA controller
- 1997 – 3D acceleration functions:
  - Hardware for triangle setup and rasterization
  - Texture mapping
  - Shading



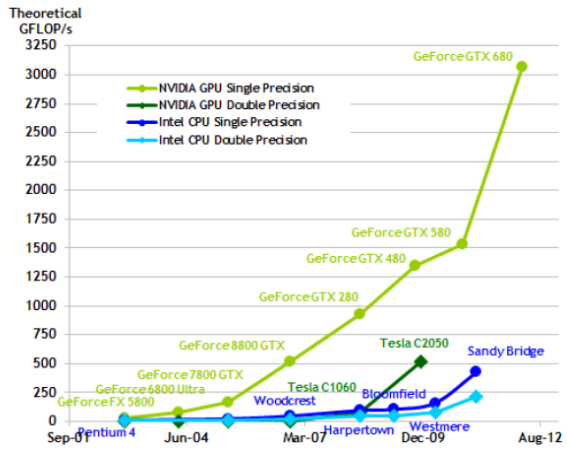
4

GPU History

- 2000 – A single chip graphics processor
  - beginning of GPU term
- 2005 – Massively parallel programmable processors
- 2007 – CUDA (Compute Unified Device Architecture)
  - Nvidia initiated
  - C/C++ extention
- 2008 – OpenCL
  - Apple initiated
  - Based on C99 standard

5

CPU vs. GPU

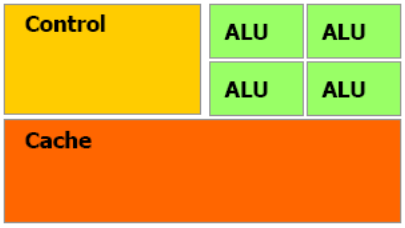


[CUDA_C_Programming Guide]

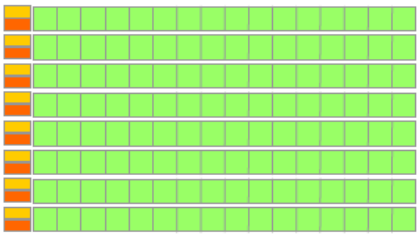
6

CPU vs. GPU (continued)

- CPU



- GPU



[CUDA_C_Programming Guide]

7

CPU vs. GPU (continued)

- CPU
  - Latency oriented Cores
  - Large caches
    - Lessen latency
  - Sophisticated control
    - Branch prediction
    - Data forwarding
  - Powerful ALUs
    - Reduce Latency
- GPU
  - Throughput Oriented Cores
  - Small caches
    - To boost memory throughput
  - Simple control
    - No branch prediction
    - No data forwarding

8

CPU vs. GPU (continued)

- CPU
  - Sequential parts where latency matters
    - 10+X faster than GPU for sequential codes
- GPU
  - Heavily piplined
  - Require massive number of threads to tolerate latencies

9

Case Study

- CPU
  - Intel Core i7:960
    - 4-core, 3.2 GHz
    - 2-way multi-threading
    - 4-way SIMD
    - L1 32KB, L2 256KB, L3 3MB
    - 32 GB/sec
- GPU
  - NVIDIA GTX 280
    - 30 core, 1.3GHz
    - 1024-way multi-threading
    - 8-way SIMD
    - 16KB software managed cache (shared memory)
    - 141 GB/sec

10

Number of Cores

- It is all about the core complexity:
  - The common goal: Improving pipeline efficiency
  - CPU goal: Single-thread performance
    - Exploiting ILP
    - Sophisticated branch predictor
    - Multiple issue logics
  - GPU goal: Throughput
    - Interleaving hundreds of threads

11

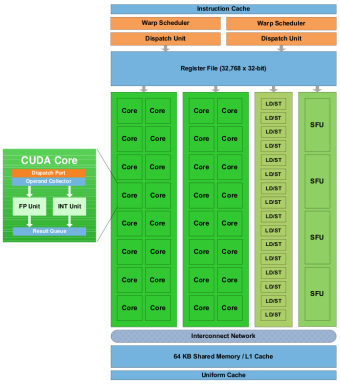
Cache Size

- CPU goal: reducing memory latency
  - Programmer-transparent data caching
    - Increasing the cache size to capture the working set
  - Prefetching (HW/SW)
- GPU goal: hiding memory latency
  - Interleave the execution of hundreds of threads to hide the latency of each other
- Notice:
  - CPU uses multi-threading for latency hiding
  - GPU uses software controlled caching (shared memory) for reducing memory latency

12

Streaming Multiprocessor (SM)

- Pipeline deep-multithreaded SIMD processor
- Multiple SIMD groups
- Resources shared among threads
  - Shared memory
  - Register file
  - L1Cache

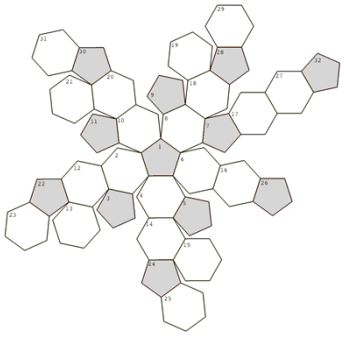


[NVIDIA Fermi Architecture Whitepaper]

13

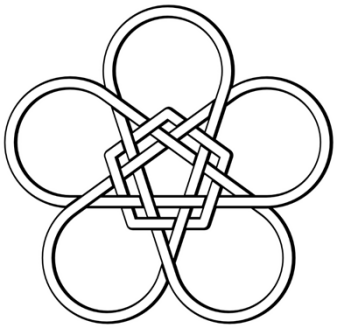
Compute Capability (Nvidia)

- Denotes the Capability of the GPU
  - Major . Minor
  - 1.0 and 1.1
  - 1.2
    - Atomic operation on shared memory
  - 1.3
    - Double-precision operations
  - 2.x and 3.x
    - Informative __syncthreads
    - 64-bit atomic
    - 3D grid
    - Tensor Cores
    - Mix Lib (Curand, CuFFT, ..)



14

GPU Programming Model



15

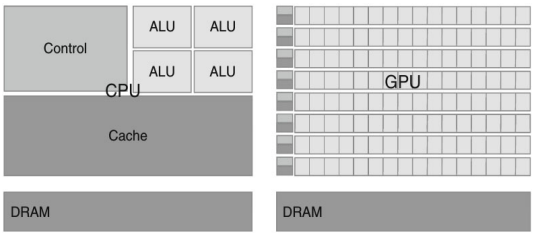
What Is CUDA

- Compute Unified Device Architecture
- Based on industry-standard C
- A handful of language extensions to allow heterogeneous programs
- Straightforward APIs to manage devices, memory, etc

16

Why CUDA?

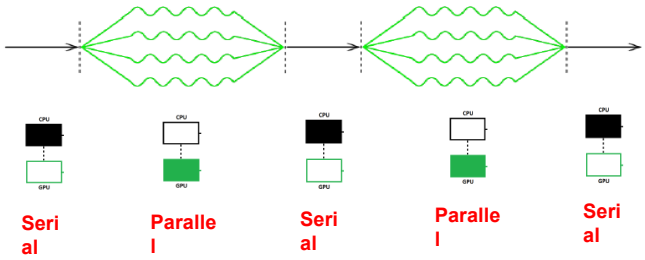
- Massive parallel computing power
- Maximize throughput of all threads
- Using hundreds of ALU inside a GPU



17

CUDA Programming Model

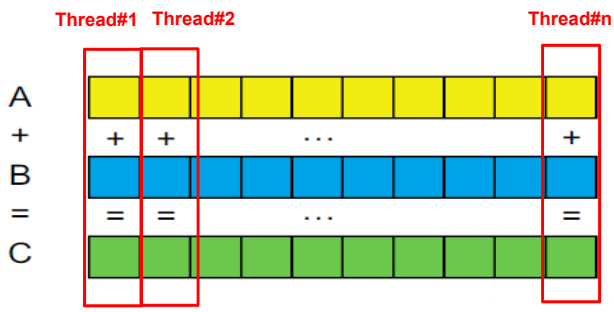
- Heterogeneous Programming
  - program separated into serial regions (run on CPU) & parallel regions (run on GPU)



18

CUDA Programming Model

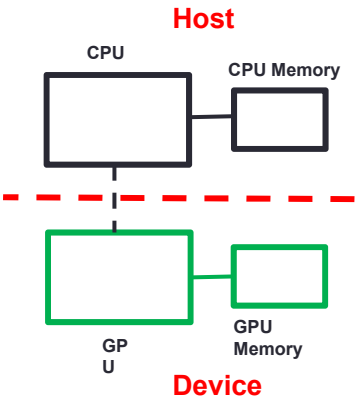
- Parallel regions consist of many calculations that can be executed independently
  - Data Parallelism (e.g. vector addition)



19

A Basic CUDA Program Outline

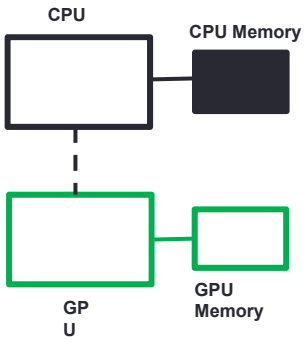
```
Int main() {
// Allocate memory for array on host
//  Allocate memory for array on device
// Fill array on host
// copy data from host array to device array
// Do something on device (e.g. vector addition)
// Copy data from device array to host array
// Check data for correctness
// Free Host Memory
// Free Device Memory  }
```



20

A Basic CUDA Program Outline

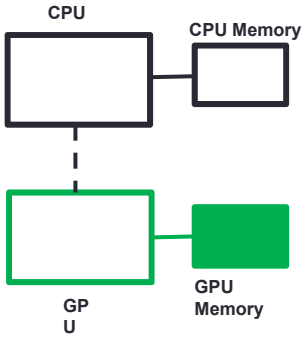
```
Int main() {
// Allocate memory for array on host
//  Allocate memory for array on device
// Fill array on host
// Copy data from host array to device array
// Do something on device (e.g. vector addition)
// Copy data from device array to host array
// Check data for correctness
// Free Host Memory
// Free Device Memory  }
```



21

A Basic CUDA Program Outline

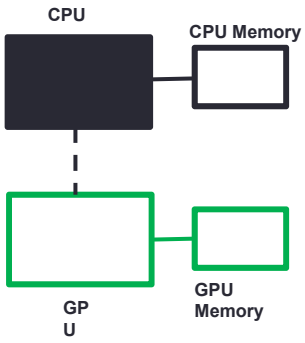
```
Int main() {
// Allocate memory for array on host
//  Allocate memory for array on device
// Fill array on host
// Copy data from host array to device array
// Do something on device (e.g. vector addition)
// Copy data from device array to host array
// Check data for correctness
// Free Host Memory
// Free Device Memory  }
```



22

A Basic CUDA Program Outline

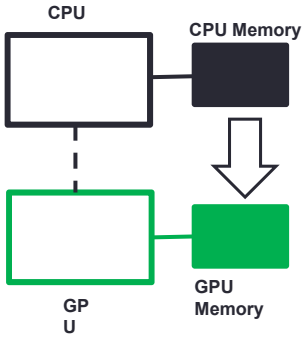
```
Int main() {
// Allocate memory for array on host
//  Allocate memory for array on device
// Fill array on host
// Copy data from host array to device array
// Do something on device (e.g. vector addition)
// Copy data from device array to host array
// Check data for correctness
// Free Host Memory
// Free Device Memory  }
```



23

A Basic CUDA Program Outline

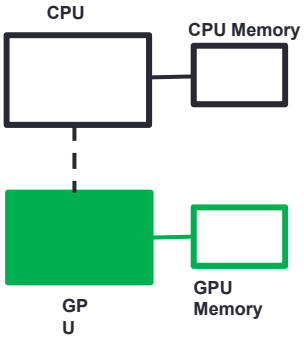
```
Int main() {
// Allocate memory for array on host
//  Allocate memory for array on device
// Fill array on host
// Copy data from host array to device array
// Do something on device (e.g. vector addition)
// Copy data from device array to host array
// Check data for correctness
// Free Host Memory
// Free Device Memory  }
```



24

A Basic CUDA Program Outline

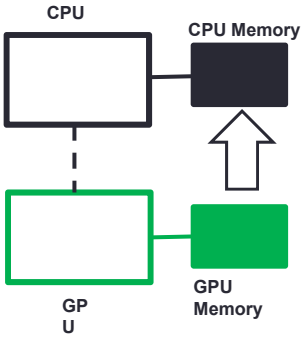
```
Int main() {
// Allocate memory for array on host
//  Allocate memory for array on device
// Fill array on host
// Copy data from host array to device array
// Do something on device (e.g. vector addition)
// Copy data from device array to host array
// Check data for correctness
// Free Host Memory
// Free Device Memory  }
```



25

A Basic CUDA Program Outline

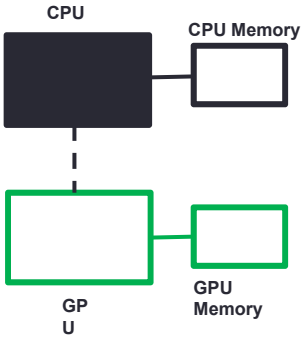
```
Int main() {
// Allocate memory for array on host
//  Allocate memory for array on device
// Fill array on host
// Copy data from host array to device array
// Do something on device (e.g. vector addition)
// Copy data from device array to host array
// Check data for correctness
// Free Host Memory
// Free Device Memory  }
```



26

A Basic CUDA Program Outline

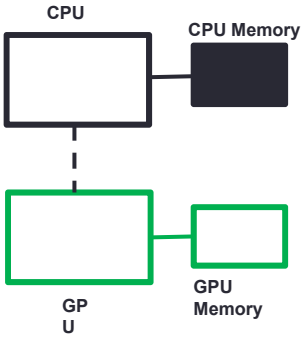
```
Int main() {
// Allocate memory for array on host
//  Allocate memory for array on device
// Fill array on host
// Copy data from host array to device array
// Do something on device (e.g. vector addition)
// Copy data from device array to host array
// Check data for correctness
// Free Host Memory
// Free Device Memory  }
```



27

A Basic CUDA Program Outline

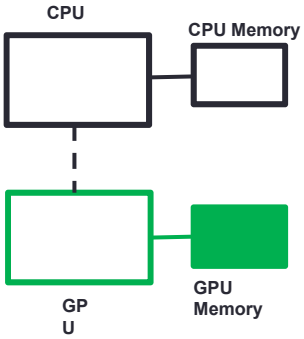
```
Int main() {
// Allocate memory for array on host
//  Allocate memory for array on device
// Fill array on host
// Copy data from host array to device array
// Do something on device (e.g. vector addition)
// Copy data from device array to host array
// Check data for correctness
// Free Host Memory
// Free Device Memory  }
```



28

A Basic CUDA Program Outline

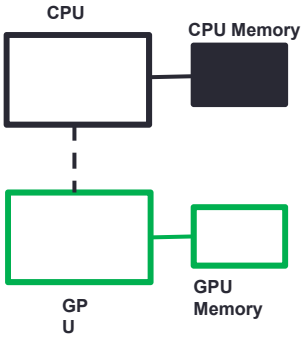
```
Int main() {
// Allocate memory for array on host
//  Allocate memory for array on device
// Fill array on host
// Copy data from host array to device array
// Do something on device (e.g. vector addition)
// Copy data from device array to host array
// Check data for correctness
// Free Host Memory
// Free Device Memory  }
```



29

A Basic CUDA Program Outline

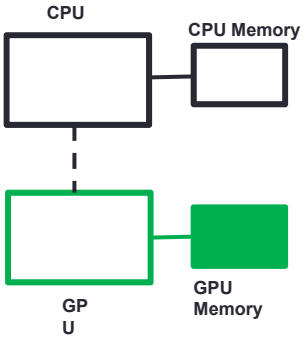
```
Int main() {
    // Allocate memory for array on host
    size_t bytes = N*sizeof(int);
    int *A = (int*)malloc(bytes);
    int *B = (int*)malloc(bytes);
    int *C = (int*)malloc(bytes);
            . . .
}
```



30

A Basic CUDA Program Outline

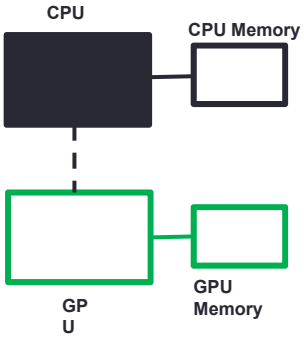
```
Int main() {
        . . .
  // Allocate memory for array on device
   int *d_A, *d_B, *d_C;
   cudaMalloc(&d_A, bytes);
   cudaMalloc(&d_B, bytes);
   cudaMalloc(&d_C, bytes);
        . . .
}
```



31

A Basic CUDA Program Outline

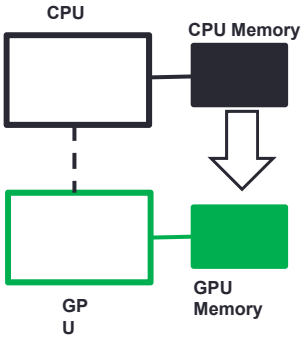
```
Int main() {
        . . .
  // Fill array on host
   for(int i=0; i<N; i++)
   {
        A[i] = 1;
        B[i] = 2;
        C[i] = 0;
   }
        . . .
}
```



32

A Basic CUDA Program Outline

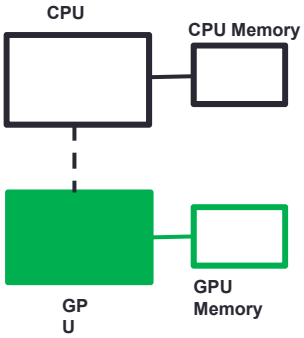
```
Int main() {
        . . .
  // Copy data from host array to device array
  cudaMemcpy(d_A, A, bytes,
             cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, B, bytes,
             cudaMemcpyHostToDevice);
        . . .
}
```



33

A Basic CUDA Program Outline

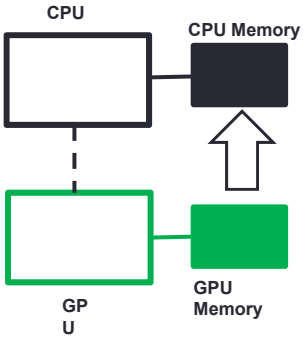
```
Int main() {
        . . .
  // Do something on device (e.g. vector addition)
  // We'll come back to this soon
        . . .
}
```



34

A Basic CUDA Program Outline

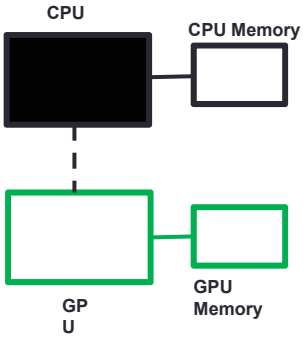
```
Int main() {

        . . .

// Copy data from device array to host array

 cudaMemcpy(C, d_C, bytes,
            cudaMemcpyDeviceToHost);

        . . .

}
```



35

A Basic CUDA Program Outline

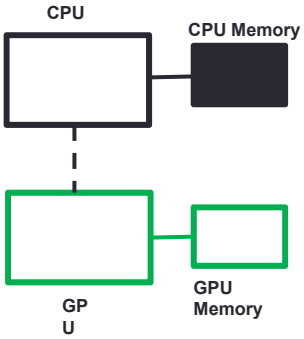
```
Int main() {

        . . .

    // Check data for correctness

    for (int i=0; i<N; i++)
    {
      if(C[i] != 3)
      {
        // Error – value of C[i] is not correct!
      }
    }

    . . .
}
```



36

A Basic CUDA Program Outline

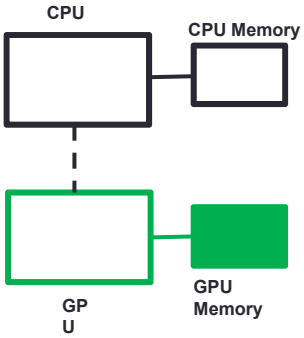
```
Int main() {
        . . .
  // Free Host Memory
  free(A);
  free(B);
  free(C);
        . . .
}
```



37

A Basic CUDA Program Outline

```
Int main() {
        . . .
  // Free Device Memory
  cudaFree(d_A);
  cudaFree(d_B);
  cudaFree(d_C);
}
```



38

CUDA Kernels

- What is difference between serial and parallel implementation?

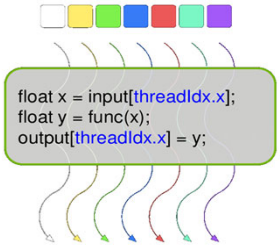
Serial – CPU

```
for (int i=0; i<N; i++){
        C[i] = A[i] + B[i];
}
```

Parallel – GPU

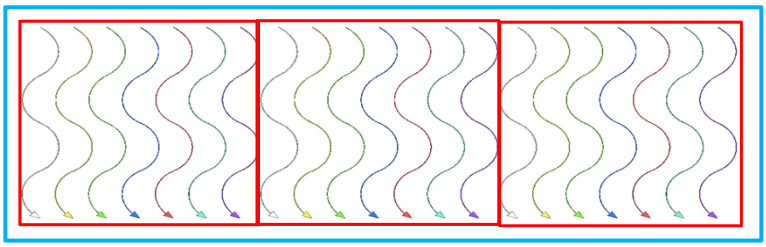
```
C[i] = A[i] + B[i];
```

- A kernel is a function executed on the GI as an array of threads in parallel
- Same code is executed by all threads
  - Single-Program Multiple-Data (SPMD)
- Each thread has:
  - thread ID
  - inputs, and output results



39

CUDA Kernels



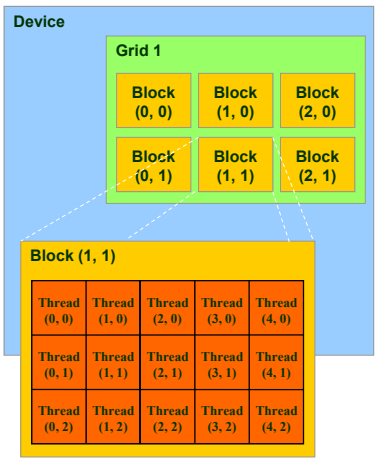
- Threads are grouped into blocks
- Blocks are grouped into a grid

Kernel is executed as a grid of blocks of threads

40

Built-in Variables

- gridDim: Grid dimension
- blockDim: Block dimension
- blockIdx: Block index
- threadIdx: Thread index



41

Execution Configuration

- 1D grid / 1D blocks

  gridDim.x = 1024 blockDim.x = 64
  gridDim.y = 1 blockDim.y = 1
  blockDim.z = 1

```
dim3 gd(1024)
dim3 bd(64)
akernel<<<gd, bd>>>(...)
```

- 2D grid / 3D blocks

  gridDim.x = 4 blockDim.x = 64
  gridDim.y = 128 blockDim.y = 16
  blockDim.z = 4

```
dim3 gd(4, 128)
dim3 bd(64, 16, 4)
akernel<<<gd, bd>>>(...)
```

42

CUDA Function Declarations

Functions	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
  - Can only call **__device__** functions

Vector Addition Kernel (1)

- Device code

```
__global__ vector_addition (int *a, int *b, int *c )
{
    int i = blockIdx.x ;

    c[i] = a[i] + b[i];
}
```

- Host code

```
Int main(){
…
vector_addition <<< N , 1 >>> (d_A, d_B, d_C);
}
```

N blocks

Each block has one thread

Vector Addition Kernel (2)

- Device code

```
__global__ vector_addition (int *a,  int *b,  int *c )
{

    int i = threadId

    c[i] = a[i] + b[i
}
```

What is wrong with this code?

- Host code

One block with N threads

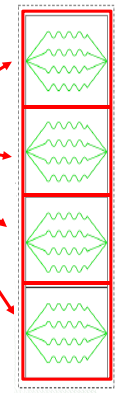
```
Int main(){
…
vector_addition  <<< 1 , N >>> (d_A, d_B, d_C);
}
```


Vector Addition Kernel (3)

- Device code

```
__global__ vector_addition (int *a, int *b, int *c )
{

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

  if (i<N)
          c[i] = a[i] + b[i];
}
```

Blocks

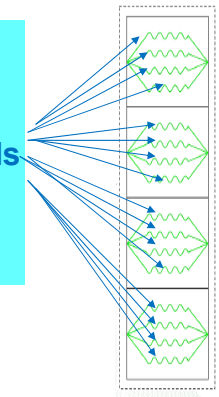


Vector Addition Kernel (3)

- Device code

```
__global__ vector_addition (int *a, int *b, int *c )
{
 int i = blockDim.x * blockIdx.x + threadIdx.x;

 if (i<N)
        c[i] = a[i] + b[i];
}
```

Threads



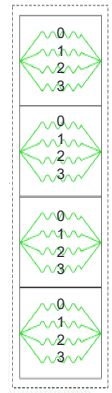
47

Vector Addition Kernel (3)

- Device code

```
__global__ vector_addition (int *a, int *b, int *c )
{
        (4)        (0-3)      (0-3)
 int i = blockDim.x * blockIdx.x + threadIdx.x;

 if (i<N)
        c[i] = a[i] + b[i];
}
```

0
1
2
3

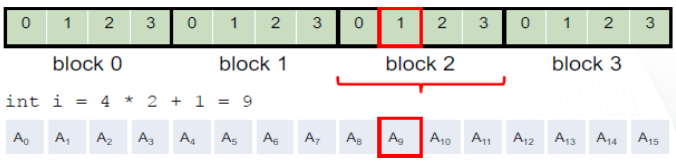


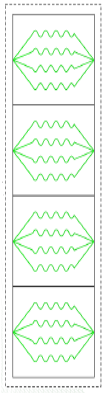
48

Vector Addition Kernel (3)

- Device code

```
__global__ vector_addition (int *a, int *b, int *c )
{
                (4)          (2)          (1)
  int i = blockDim.x * blockIdx.x + threadIdx.x;

  if (i<N)
            c[i] = a[i] + b[i];
}
```



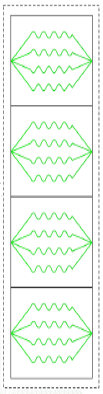


49

Vector Addition Kernel (3)

- Device code

```
__global__ vector_addition (int *a, int *b, int *c )
{

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

  if (i<N)
            c[i] = a[i] + b[i];
}
```

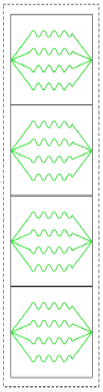


50

Vector Addition Kernel (3)

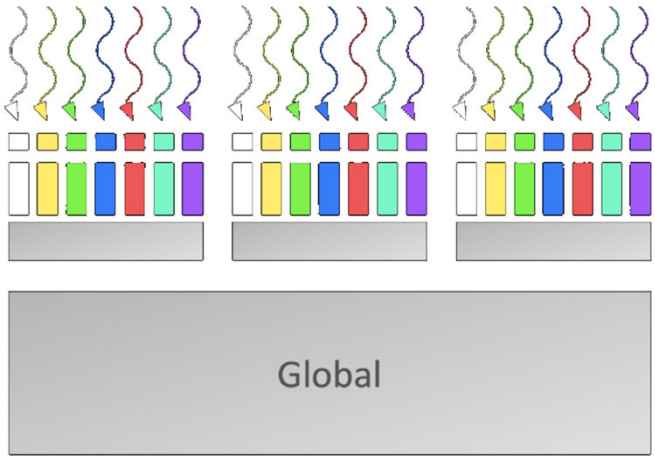
- Device code

```
__global__ vector_addition (int *a, int *b, int *c )
{

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

  if (i<N)
            c[i] = a[i] + b[i];
}
```



51

Memory Model



52

Shared Memory

- Very fast on-chip memory
- Allocated per thread block
  - Allows data sharing between threads in the same block
  - Declared with __shared__ specifier
- Limited amount
- Must take care to avoid race conditions. For example…
  - Say, each thread writes the value 1 to one element of an array element
  - Then one thread sums up the elements of the array
  - Synchronize with __syncthreads()

53

SM EXECUTION & DIVERGENCE

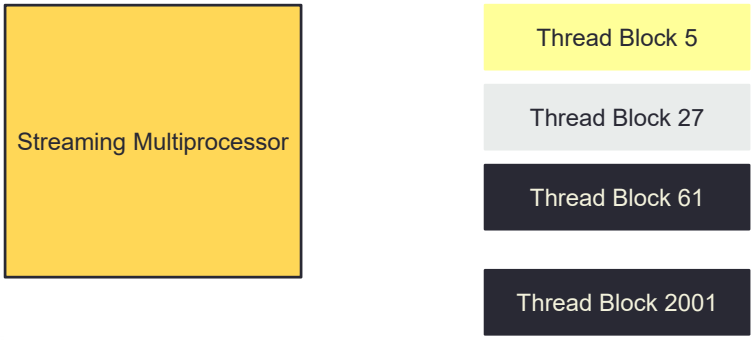
54

How an SM executes threads

- Overview of how a Stream Multiprocessor works
- SIMT Execution
- Divergence

55

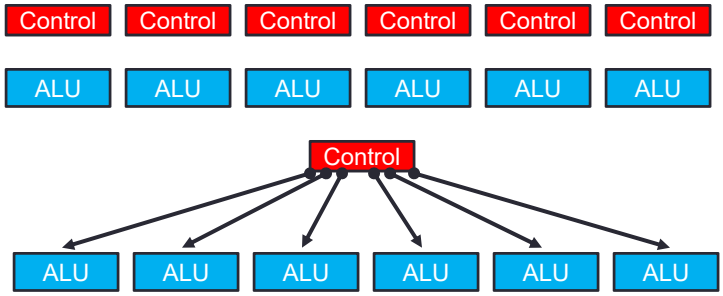
Scheduling Blocks onto SMs



- **HW Schedules thread blocks onto available SMs**
  - **No guarantee of ordering among thread blocks**
  - **HW will schedule thread blocks as soon as a previous thread block finishes**

56

Warps



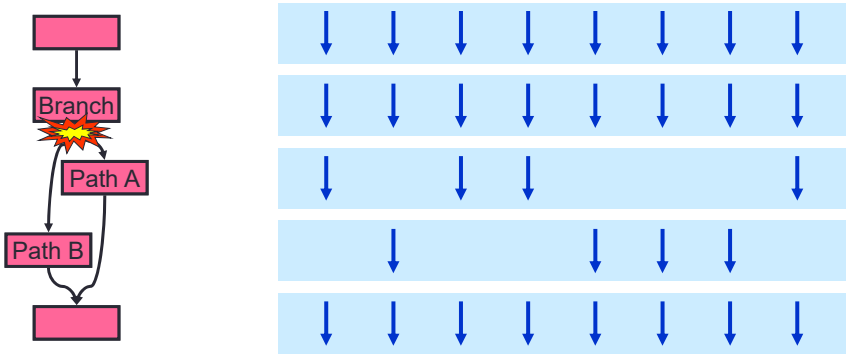
- A warp = 32 threads launched together
  - Usually, execute together as well

Control Flow Divergence

- What happens if you have the following code?

```
if(foo(threadIdx.x))
{
   do_A();
}
else
{
   do_B();
}
```

Control Flow Divergence



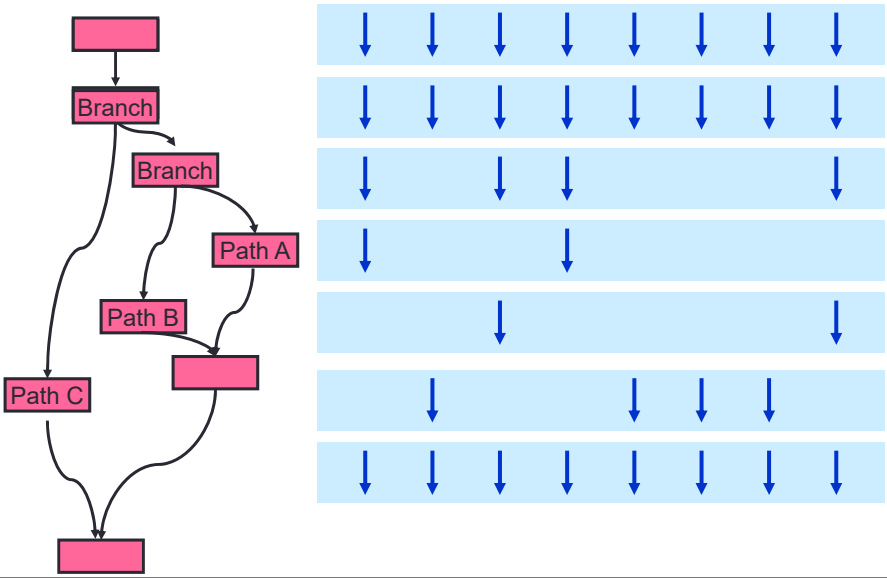
From Fung et al. MICRO '07

Control Flow Divergence

- Nested branches are handled as well

```
if(foo(threadIdx.x))
{
  if(bar(threadIdx.x))
    do_A();
  else
    do_B();
}
else
  do_C();
```

Control Flow Divergence



61

Control Flow Divergence

- You don't have to worry about divergence for correctness (*)
- You might have to think about it for performance
  - Depends on your branch conditions

62

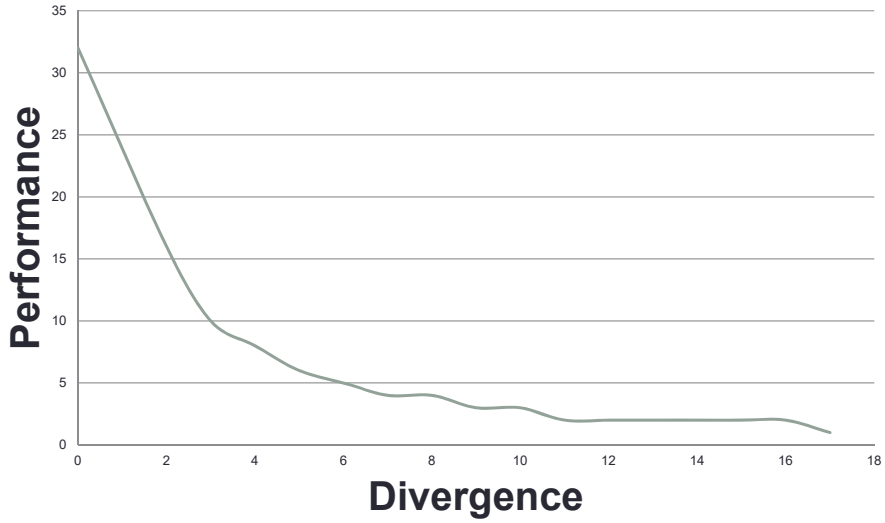
Control Flow Divergence

- Performance drops off with the degree of divergence

```
switch(threadIdx.x % N)
{
  case 0:
    ...
  case 1:
    ...
}
```

63

Divergence



64

END OF SLIDES

65