GPU and CUDA Programming

Single Instruction Stream, Multiple Data Stream (SIMD)

- Same instruction is executed by multiple processors using different data streams
- SIMD exploits data-level parallelism by applying the same operation to multiple items of data in parallel
- Data parallelism for
  - Matrix-oriented scientific computing
  - Media-oriented image and sound processors
- Each processor has its own data memory, but there is a single instruction memory and control processor
- In SIMD, programmer continues to think sequentially, yet achieves parallel speedup
- Types of SIMD architecture:
  - Vector architectures
  - Multimedia extensions to standard instruction sets
  - Graphics processor units (GPUs)

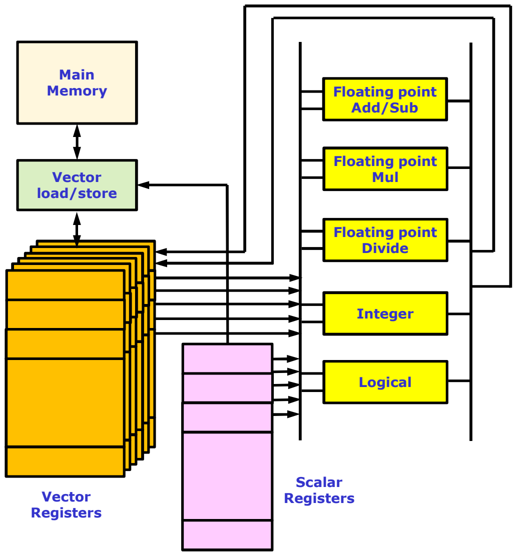
2

Vector Architecture

- It is also called as array processors
- Basic idea:
  - Read sets of data elements scattered about memory
  - Place them into large sequential register files (vector registers)
  - Processes the data using its own functional units
  - Dispense the results back into memory
- Single instruction operates on vectors of data
  - It results in dozes of register-register operations on independent data elements
- Vector load and store are deeply pipelined
- Registers are controlled by compiler
  - Used to hide memory latency
  - Leverage memory bandwidth
  - Program pays long memory latency only once per vector load or store

3

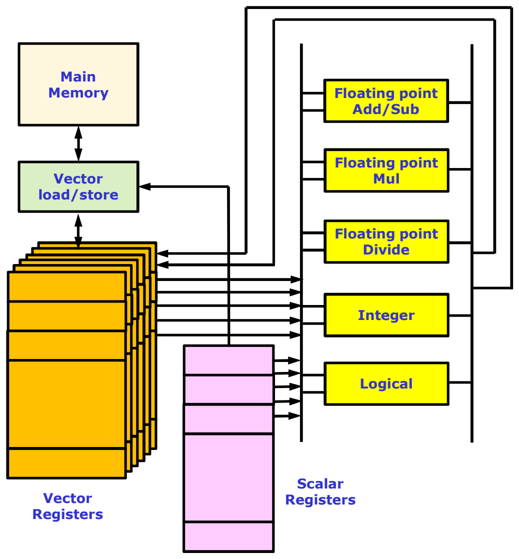
Vector Processor



- Example architecture of vector processor based on Cray-1
- Vector registers
  - Fixed length bank holding a single vector
  - 8-32 vector registers
  - Each hold 64 elements and each element is 64-bit wide
  - Register file has 16 read ports and 8 write ports

4

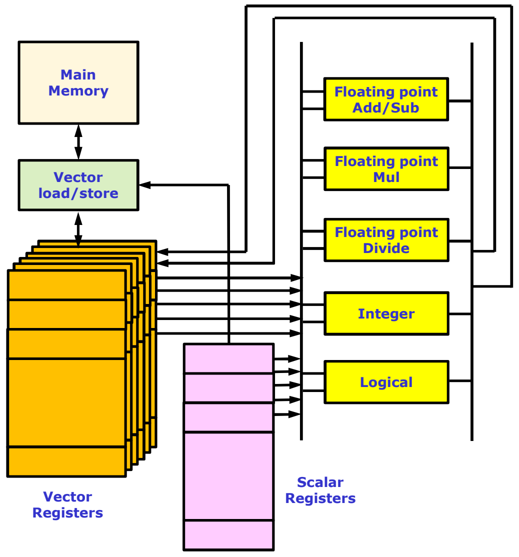
Vector Processor



- Example architecture of vector processor based on Cray-1
- Vector functional units
  - Fully pipelined
  - Data and control hazards are detected
- Vector load-store unit
  - Loads or stores a vector into or from memory
  - Fully pipelined
  - One word per clock cycle after initial latency

5

Vector Processor



- Example architecture of vector processor based on Cray-1
- Scalar registers
  - 32-64 general-purpose registers
  - 32-64 floating-point registers
  - They provide data as input to vector functional units
  - Compute addresses to pass to vector load/store unit

6

Vector Instructions

- VV: two vector operands
- VS or SV: vector and scalar operands
- Example:
  - ADDVV.D V1, V2, V3: Addition of two double-precision vectors
  - ADDVS.D V1, V2, F0: Add content of scalar register to each element in a vector register
- Most vector operations has a vector destination register
- A few vector operations produce a scalar value which is stored in scalar register
- LV/SV: vector load and vector store from address
  - One of the operand is vector register to be loaded or stored
  - Another operand is general purpose register: It is the starting address of the memory
  - Example: LV V1, R1

7

Vector Instructions

- Two special-purpose registers:
  - Vector length: Used when the natural length is not 64
  - Vector mask: Used when loop involve IF statements
- With vector instruction, the system can perform the operations on vector data elements in many ways
  - Operating on many elements simultaneously
- This flexibility allow the vector processors to achieve high performance at low power
- Independence of elements with in a vector instruction set allows scaling of function units without performing additional costly dependency checks
- Vectors naturally accommodate varying data sizes
- Vector register size is viewed as:
  - 64, 64-bit data element
  - 128, 32-bit data element
  - 256, 16-bit data element
  - 512, 8-bit data element

8

Working of Vector Processors

- Example: Y = a * X + Y
  - X and Y are vectors and initially resident in memory
  - a is a scalar

```
for (i=0; i<n; i=i+1)
        Y[i] = a * X[i] + Y[i];
```

- The compiler produces vector instruction
- Resulting code spends time in running in vector mode
- Code is said to be vectorized or vectorizable
- Loop carried dependences: dependences between instructions of loop
  - Loops can be vectorized when they do not have dependences between instructions of loop

9

Working of Vector Processors

- Example: Y = a * X + Y
  - X and Y are vectors and initially resident in memory
  - a is a scalar

```
for (i=0; i <n; i=i+1)
        Y[i] = a * X[i] + Y[i];
```

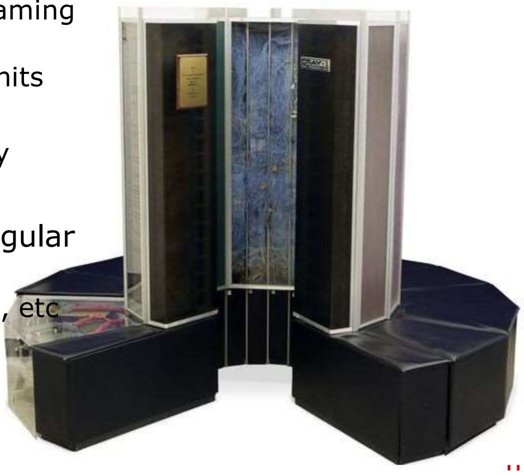
- The compiler produces vector instruction

```
L.D       F0, a         ; Load scalar a
LV        V1, Rx        ; Load vector X starting from Rx
MULVS.D   V2, V1, F0    ; a * X (vector-scalar multiply)
LV        V3, Ry        ; Load vector Y starting from Ry
ADDVV.D   V4, V2, V3    ; a * X + Y
SV        Ry, V4        ; Store vector Y starting from Ry
```

10

Vector Supercomputer

- The Cray-1 was the first supercomputer to successfully implement the vector processor design
- Specialised hardwares
  - Collection of Vector processors (Streaming Multiprocessors)
  - multiple arithmetic units
  - Multiple ALUs
  - Large shared memory
- Very expensive
- Suitable for regular problems:
  - matrix manipulations, etc



Multimedia Extensions to Standard Instruction Sets

- Focussed in most of the instruction set architectures (ISAs) that support multimedia applications
- X86 architecture: SIMD instruction extension
  - MMX (Multimedia Extension)
  - SSE (Streaming SIMD Extensions)
  - AVX (Advanced Vector Extensions)
- These are vector type instructions on conventional CPUs
- They execute the same single instruction on multiple data operands
  - SSE comes with 128-bit registers
  - AVX extends this to 256-bit

12

Graphics Processing Units (GPUs)

- GPU share features with vector architecture and multimedia instruction extension
- GPU have its own distinguishing characteristics
- The ecosystem in which GPU is a part has
  - A system processor
  - A system memory
  - GPU and
  - Graphics memory
- This ecosystem is referred as heterogeneous architecture
- GPU with hundreds of parallel floating-point units, makes high performance computing more accessible
- This potential is combined with a programming language made GPUs easier to program
- Primary ancestors of GPUs are graphics accelerators
- Now a days, GPU are moving towards mainstream computing

13

GPU vs CPU

- GPU and CPU are architecturally very different devices

- CPU is designed for running a small number of potentially quite complex tasks
- CPU design is aimed at systems that execute a number of discrete and unconnected tasks
  - CPUs suitable for running operating systems and application softwares

- GPU is designed for running a large number of quite simple task
- GPU design is aimed at problems that can be breakdown into thousands of tiny fragments and worked on individually

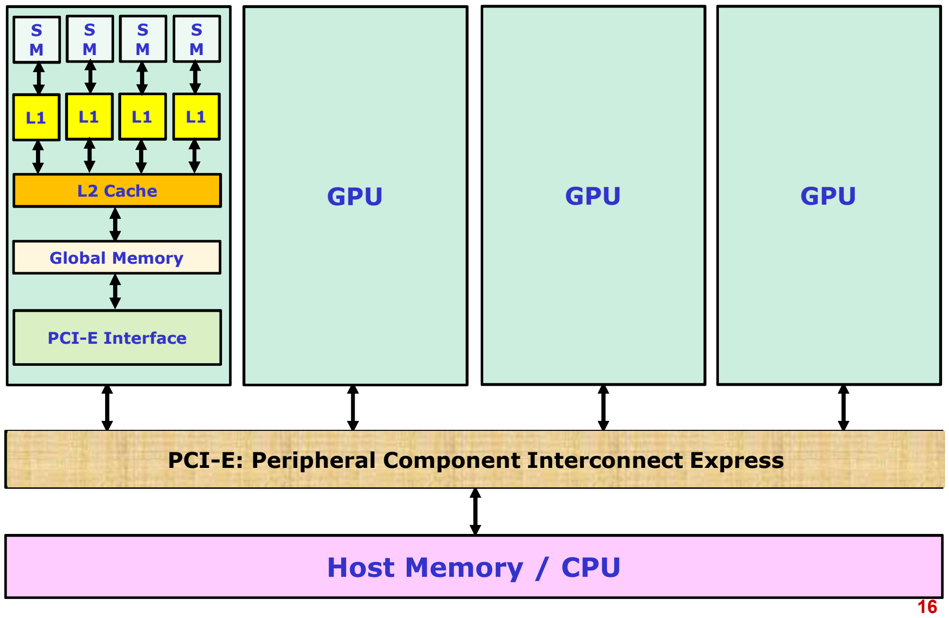
14

GPU vs CPU

- Compiler generate GPU code that will be run only in GPUs
- The CPU and GPU have separate memory spaces
  - The CPU parameters can not be accessed in the GPU code and vice versa
- Challenges for GPU programmer:
  - Getting good performance
  - Coordinating the scheduling of computation on the system processor and GPU
  - Coordinating the transfer of data between system memory and GPU memory

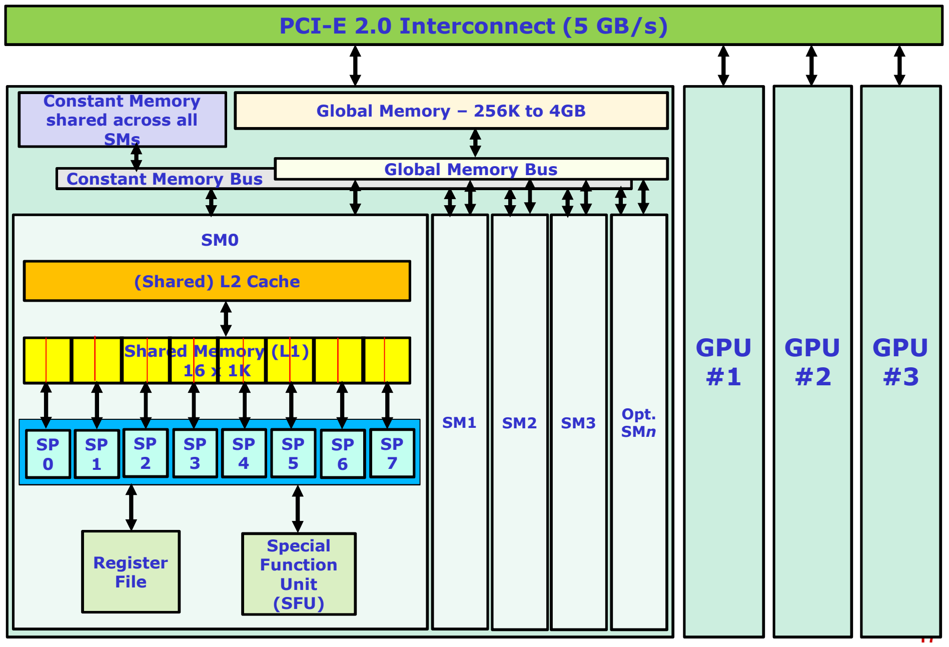
15

GPU Architecture



16

Block Diagram of GPU Card (G80/GT200)


PCI-E 2.0 Interconnect (5 GB/s)
Constant Memory shared across all SMs
Global Memory – 256K to 4GB
Constant Memory Bus
Global Memory Bus
SM0
(Shared) L2 Cache
Shared Memory (L1) 16 x 1K
SP 0
SP 1
SP 2
SP 3
SP 4
SP 5
SP 6
SP 7
Register File
Special Function Unit (SFU)
SM1
SM2
SM3
Opt. SMn
GPU #1
GPU #2
GPU #3

GPU Hardware

- Key blocks in GPU hardware are:
  - Memory (Global, Constant, Shared)
  - Streaming multiprocessors (SMs)
  - Streaming processors (SPs)
- GPU is array of SMs
- A GPU device consists of one or more SMs
- Each SM has *N* cores
- Each core is also called SP (vector processor)
  - G80/GT200: 8 SPs
  - Fermi series: 32-48 SPs
  - Kepler series: 192+ SPs
- Each SM has access to register file

18

GPU Hardware

- Shared memory block is accessible to individual SM
  - Used as program-managed cache
- Global memory:
  - Supplied via Graphic Double Data Rate (GDDR)
  - High performance version of DDR memory
- Constant memory:
  - Used for read-only data
  - A view into the main global memory
- Texture memory:
  - Useful for data where there is interpolation (2D/3D lookup table)
  - A view into the main global memory
- Special purpose units (SPUs):
  - They perform special hardware instructions such as high-speed 24-bit sine/cosine/exponent operations

19

Commercial GPUs

- Consumer graphics cards (GeForce): (Gaming card)
  - GTX 960: 1024 CUDA cores, 2GB GDDR: SMs - 8
  - GTX 980: 2048 CUDA cores, 4GB GDDR: SMs - 16
  - GTX 1080: 2560 CUDA cores, 8 GB GDDR: SMs - 20
  - GTX 1080Ti: 3584 CUDA cores, 11 GB GDDR: SMs - 28
  - RTX 2080Ti: 4352 CUDA cores, 11 GB GDDR: SMs – 68

- Other cards: Tesla K10, Tesla K20, Tesla K40 and latest Tesla V100.

20

Commercial GPUs

- Cards for HPC and AI (Tesla):
  - Tesla K10: 3072 CUDA cores, 8 GB GDDR: SMs – 6
  - Tesla K20: 2688 CUDA cores, 6 GB GDDR: SMs – 6
  - Tesla K40: 2880 CUDA cores, 12 GB GDDR: SMs – 13
  - Tesla K80: 4992 CUDA cores, 24 GB GDDR: SMs – 15
  - Tesla V100: World's most advanced data centre GPU to accelerate AI, HPC, and Graphics
    - **5120 CUDA cores**, **640 Tensor cores**, **32 GB GDDR**: **SMs – 84** (V stands for Volta)

- Cards for visualization:
- Cards for embedded systems and IoT:

21

CUDA Programming Model

- NVIDIA developed C-like language and programming environment to improve the productivity of GPU programmers
  - This addresses the challenges of heterogeneous computing and multifaceted parallelism
- NVIDIA called the GPU architecture as CUDA (Compute Unified Device Architecture)
- CUDA produces C/C++ for the system processor (host) and C and C++ dialect for GPU (device)
- OpenGL: Similar programming language as CUDA
- OpenACC: OpenMP like directive version of CUDA
- NVIDIA unified all forms of parallelism (Multithreading, MIMD, SIMD and ILP) in CUDA Thread as a programming primitive for lowest level of parallelism
  - Using this, compiler and hardware group thousands of CUDA Threads together to utilize the various style of parallelism with the GPU

22

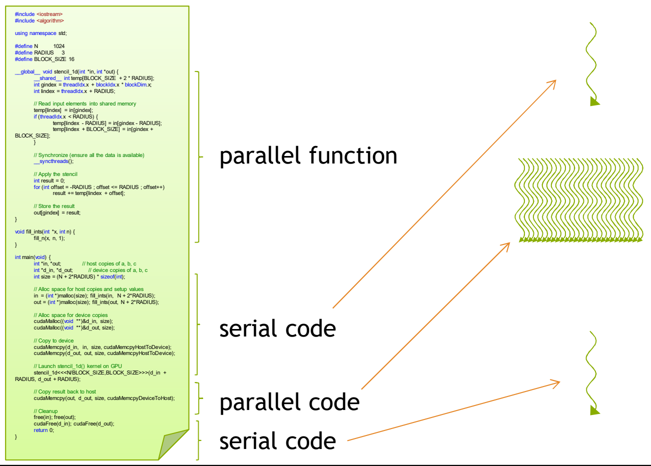
CUDA as Heterogeneous Computing

- NVIDIA classifies the CUDA programming model as Single Instruction, Multiple Thread (SIMT)
- Terminology:
  - Host: The CPU and its memory (host memory)
  - Device: The GPU and its memory (device memory)
- Heterogeneous programming: Both CPU code and GPU code co-exit in the same program

23

CUDA as Heterogeneous Computing

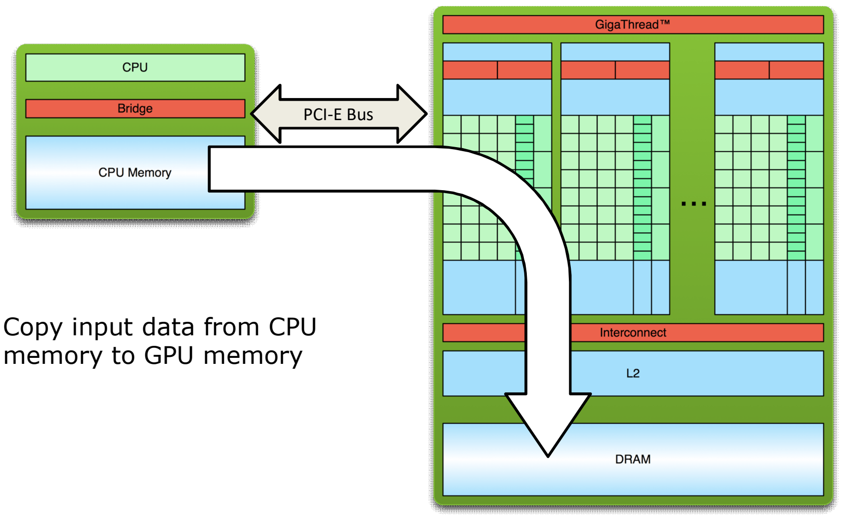
- Heterogeneous programming: Both CPU code and GPU code co-exit in the same program
- Source code will be split into two sections: host code and device code



```
#include <iostream>
#include <algorithm>

using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
        __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
        int gindex = threadIdx.x + blockIdx.x * blockDim.x;
        int lindex = threadIdx.x + RADIUS;

        // Read input elements into shared memory
        temp[lindex] = in[gindex];
        if (threadIdx.x < RADIUS) {
                temp[lindex - RADIUS] = in[gindex - RADIUS];
                temp[lindex + BLOCK_SIZE] = in[gindex +
BLOCK_SIZE];
        }

        // Synchronize (ensure all the data is available)
        __syncthreads();

        // Apply the stencil
        int result = 0;
        for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
                result += temp[lindex + offset];

        // Store the result
        out[gindex] = result;
}

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;          // host copies of a, b, c
        int *d_in, *d_out;      // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in +
RADIUS, d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

24

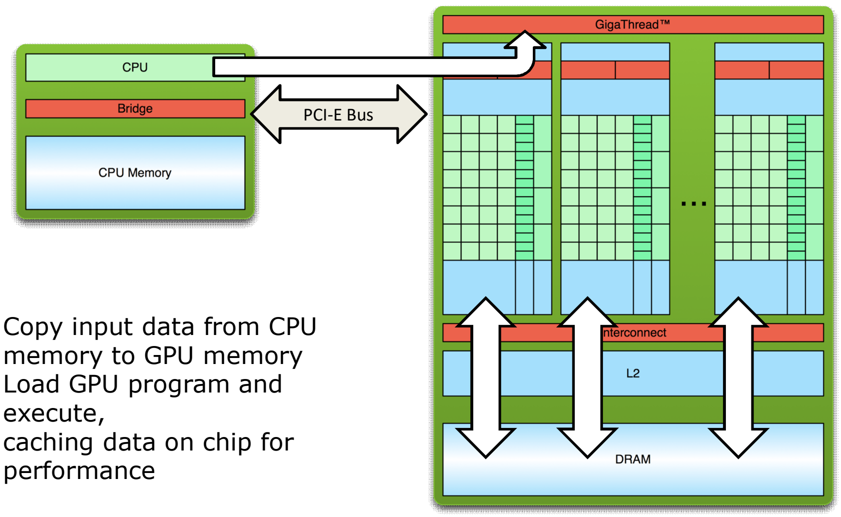
Processing Flow



1. Copy input data from CPU memory to GPU memory

25

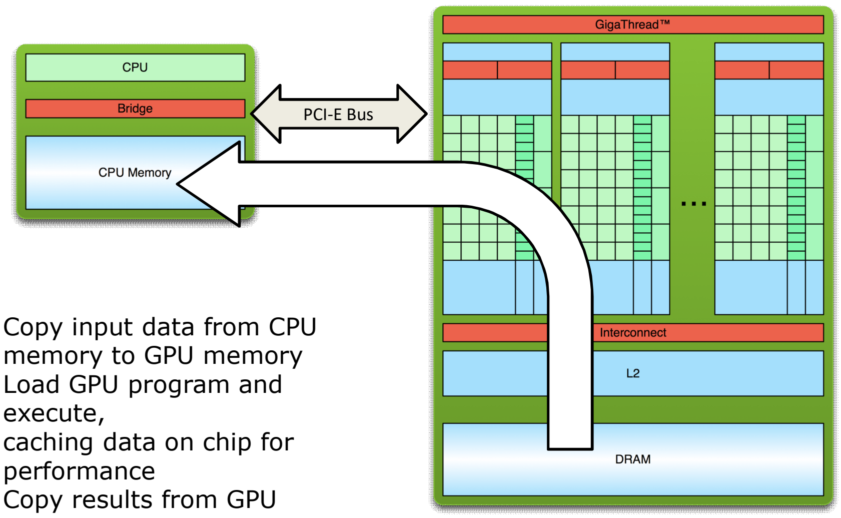
Processing Flow



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

26

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

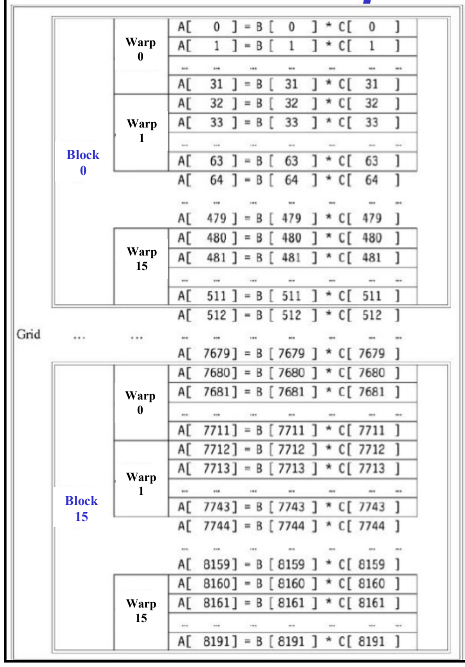
27

GPU Computational Structure

- Grid: Code that runs in GPU that consists of a set of threads
  - Code that works on the whole elements of vector in vector operation (vectorized loop)
- CUDA Thread: Programming primitive for lowest level of parallelism
- Warp: The threads are grouped into 32 thread groups
- Block: Group of fixed number of CUDA threads
  - These blocks are executed in group of 32 threads (warp)
  - 512 threads in a block
- Streaming Multiprocessor (SM): The hardware that executes a whole block of threads
  - SM is a multicore processor
  - Kepler hardware: 2048 or more threads per SM
  - Fermi hardware: 1536 threads per SM
- Each core is called streaming processor (SP) which is like a vector processor
  - A group of 32 threads (warp) are executed together in SP

28

GPU Computational Structure


Grid
Block 0
Warp 0
A[0] = B [0] * C[0]
A[1] = B [1] * C[1]
A[31] = B [31] * C[31]
Warp 1
A[32] = B [32] * C[32]
A[33] = B [33] * C[33]
A[63] = B [63] * C[63]
A[64] = B [64] * C[64]
A[479] = B [479] * C[479]
Warp 15
A[480] = B [480] * C[480]
A[481] = B [481] * C[481]
A[511] = B [511] * C[511]
A[512] = B [512] * C[512]
A[7679] = B [7679] * C[7679]
Block 15
Warp 0
A[7680] = B [7680] * C[7680]
A[7681] = B [7681] * C[7681]
A[7711] = B [7711] * C[7711]
Warp 1
A[7712] = B [7712] * C[7712]
A[7713] = B [7713] * C[7713]
A[7743] = B [7743] * C[7743]
A[7744] = B [7744] * C[7744]
A[8159] = B [8159] * C[8159]
Warp 15
A[8160] = B [8160] * C[8160]
A[8161] = B [8161] * C[8161]
A[8191] = B [8191] * C[8191]

- Example: Multiplication of two vector, each of length 8192 elements
- GPU code works on the whole 8192 element multiply is called Grid
- Each element is associated with a Thread
- To break it down to manageable sizes, a Grid is composed of Blocks
- Each Block is up to 512 elements i.e. 512 threads
  - Example: 8192/512 = 16 Blocks
- 32 elements (32 threads) grouped to form a Warp
  - Example: Number of Warps per Block = 512/32 = 16
- Grid and Blocks are programming abstraction implemented in GPU hardware that helps programmers to organize CUDA code

29

NVIDIA GPU Computational Structure

- GPU hardware creates, manages, schedules and executes its Thread of SIMD Instructions
- SIMD instructions are called PTX (Parallel Thread eXecution) Instructions
- SIMD instructions are 32 wide
- Each thread of SIMD instruction would compute 32 of the elements (i.e. group of 32 threads – warp)
- Warp consists of a thread of SIMD instruction and is executed on SP
- SP must have parallel functional units to perform the operation – SIMD lanes
  - Fermi hardware: 32-wide thread (warp) of SIMD instruction is mapped onto 16 physical SIMD lanes
    - 2 clock cycles to compute
  - Kepler hardware: 32-wide thread (warp) of SIMD instruction is mapped onto 32 physical SIMD lanes
    - 1 clock cycles to compute

30

Example: GTX 1080 Ti

- Number of SMs: 28
- Number of cores: 3584
- Number of cores (SPs): 3584/28 = 128 cores/SM
- Each core (SP) is like a vector processor executing SIMP instruction of length 32 (i.e. 32 threads or warp)
- Number of CUDA threads per SM = 128*32 = 4096
- Each block contain 512 CUDA threads
- Number of blocks per SM = 4096/512 = 8 blocks/SM
- Total number of threads we can schedule = 28*4096 = 1,14,688
  - 1,14,688 iterations are one clock cycle

31

NVIDIA GPU Computational Structure

- GPU hardware has two levels of hardware schedulers:

1. Block Scheduler:
   - Assign Blocks to SM
   - It ensures that Blocks are assigned to the processor whose local memories have corresponding data
   - Suppose there are 1024 Blocks to schedule and 8 SMs to schedule these onto
   - Each SM will receive 1024/8 = 128 complete Blocks
   - Suppose we have only 6 SMs
   - 1024 Blocks divided between 6 SMs as 170 complete Blocks each, plus 4 blocks left over
   - The situation of having leftover Blocks arises anytime when number of Blocks is not a multiple of number of SMs

32

NVIDIA GPU Computational Structure

- GPU hardware has two levels of hardware schedulers:

2. Warp (Thread) Scheduler
   - Scheduling threads on SIMD instructions (Warp)
   - Schedules when threads of SIMD instructions (Warp) should run
   - This scheduler is within a SM
   - Assign Warps to SP
   - Warp scheduler includes score board that lets it know which warp is ready to run
   - It sends them off to dispatch unit to be run on SPs

- The number of Blocks and the number of threads to be used in the program can be controlled by the programmer

33

NVIDIA GPU Computational Structure

- Each SP execute a 32-wide thread (Warp) at a time
  - For this it has to load 32 elements of data from memory to registers
- To hold these memory elements, a SP has 32,768 32-bit registers
- Each Warp is limited to no more than 64 registers
- For double precision floating point operands, it uses two adjacent 32-bit registers
- These registers are logically divided across SIMD Lanes
- In Fermi, each of the 16 SIMD Lanes contain 2048 registers
- In Kepler, each of the 32 SIMD Lanes contain 1024 registers

34

GPU Computational Structure

- Grid: Code that runs in GPU that consists of a set of threads
  - Code that works on the whole elements of vector in vector operation (vectorized loop)
- CUDA Thread: Programming primitive for lowest level of parallelism
- Warp: The threads are grouped into 32 thread groups
- Block: Group of fixed number of CUDA threads
  - These blocks are executed in group of 32 threads (warp)
  - 512 threads in a block
- Streaming Multiprocessor (SM): The hardware that executes a whole block of threads
  - SM is a multicore processor
- Each core is called streaming processor (SP) which is like a vector processor
  - A group of 32 threads (warp) are executed together in SP
- Grid and Blocks are programming abstraction implemented in GPU hardware that helps programmers to organize CUDA code

35

Introduction to CUDA C

- In CUDA C, there are two types of codes
  - One running in host
  - Another running in device
- In CUDA one can translate the loop by creating a kernel function
- Kernel function:
  - Function that executes on the GPU only and cannot be executed directly on the CPU
- To distinguish between the functions that on device and functions that run on host, CUDA uses compiler declarations (keywords) added to C functions
  - These are markers to the compiler generate GPU codes when compiling these function

Function declarations	Executed on	Called from
__global__	Device	Host
__device__	Device	Device
__host__	Host	Host

36

Introduction to CUDA C

- Example:

```
__global__ void some_kernel_function(parameters){
}
```

- CUDA defines an extension to C-language used to invoke a kernel

```
some_kernel_function<<<num_blocks,num_threads>>>(parameters);
```

- Triple angle brackets mark a call from *host* code to *device* code
  - Also called a "kernel launch"
- `num_blocks`: Number of blocks in the grid
- `num_threads`: Number of threads in a block
- Additional CUDA identifiers:
  - `blockIdx`: identifier for block
  - `threadIdx`: identifier for threads per block

37

Compiling CUDA C

- `nvcc`: NVIDIA compiler used to compile programs
- `.cu`: File extention used to indicate compiler that the source file is a CUDA program file
  - Example: `nvcc sample_program.cu`
- `nvcc` separates source code into host and device components
  - Device functions (e.g. `some_kernel_function()`) processed by NVIDIA compiler
  - Host functions (e.g. `main()`) processed by standard host compiler
    - `gcc`

38

CUDA Program: Addition of Two Integers

- ***Addition on the device***:
- A simple kernel to add two integers

```
__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}
```

- `__global__` is a CUDA C/C++ keyword meaning
  - `add()` will be compiled and executed on the device
  - `add()` will be called from the host
- `add()` runs on the device, so `a`, `b` and `c` must point to device memory
- We need to allocate memory on the GPU

39

CUDA Program: Addition of Two Integers

- ***Memory management***:
- Host and device memory are separate entities
  - *Device* pointers point to GPU memory
  - *Host* pointers point to CPU memory
- Simple CUDA API for handling device memory
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Similar to the C equivalents `malloc()`, `free()`, `memcpy()`
  - `cudaMemcpy()` has 4 parameters
  - `cudaMemcpy(dest, source, size, direction)`
    - `dest`: destination variable
    - `source`: source variable
    - `size`: number of bytes
    - `direction`: direction of copying – **`cudaMemcpyHostToDevice`** or **`cudaMemcpyDeviceToHost`**

40

CUDA Program: Addition of Two Integers

```
int main(void) {
     int a, b, c;            // host copies of a, b, c
     int *d_a, *d_b, *d_c; // device copies of a, b, c
     int size = sizeof(int);

    // Allocate space for device copies of a, b, c
     cudaMalloc((void **)&d_a, size);
     cudaMalloc((void **)&d_b, size);
     cudaMalloc((void **)&d_c, size);

     a = 2; // Setup input values
     b = 7;

    // Copy inputs to device
    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);
```

41

CUDA Program: Addition of Two Integers

```
int main(void) {
    .
    .
    .
    // Launch add() kernel on GPU
    add<<<1,1>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

42

Running Parallel: Addition of Two Arrays

- Adding N elements of integer array
- `add()` kernel function:

```
add<<< 1, 1 >>>();

add<<< N, 1 >>>();
```

- Instead of executing `add()` once, execute N times in parallel

43

Running Parallel: Addition of Two Arrays

- **`add<<< N, 1 >>>();`**
- Each parallel invocation of **`add()`** is referred to as a block
  - The set of blocks is referred to as a grid
  - Each invocation can refer to its block index using **`blockIdx.x`**

```
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- By using **`blockIdx.x`** to index into the array, each block handles a different index
- On the device, each block can execute in parallel:

Block 0	Block 1	Block 2	Block 3
`c[0] = a[0] + b[0];`	`c[1] = a[1] + b[1];`	`c[2] = a[2] + b[2];`	`c[3] = a[3] + b[3];`

44

Running Parallel: Addition of Two Arrays

- **CUDA Threads**
- `add<<< 1, N >>>();`
- A block can be split into parallel threads
- Let’s change `add()` to use parallel *threads* instead of parallel *blocks*
  - Each invocation can refer to its thread index with in a block using **`threadIdx.x`**

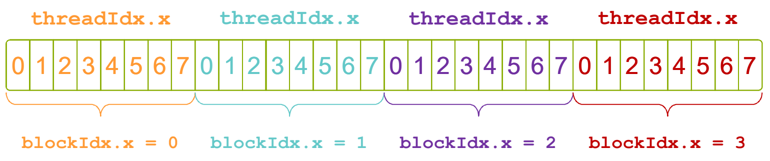
```
__global__ void add(int *a, int *b, int *c) {
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

- By using **`threadIdx.x`** to index into the array, each thread handles a different index
- **`main()`** remains same

45

Combining Blocks and Threads

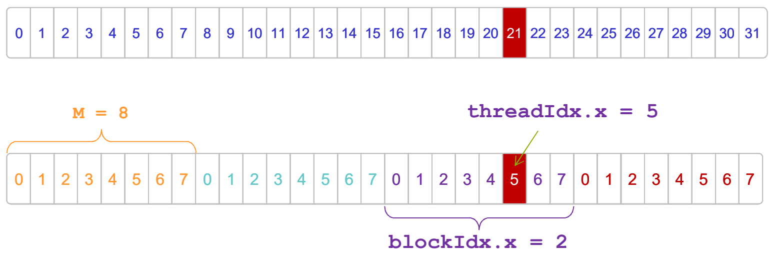
- We have seen parallel vector addition using:
  - Many blocks with one thread each
  - One block with many threads
- Adapt vector addition to use both blocks and threads
- Indexing arrays with blocks and threads:
  - Consider indexing an array with one element per thread
  - As an illustration consider 8 threads/block



46

Indexing Arrays: Example

- Which thread will operate on the red element?



```
int index = threadIdx.x + blockIdx.x * M;
          =      5      +     2      * 8;
          = 21;
```

- `M` = Size of the block (number of threads in a block)

47

Vector Addition with Blocks and Threads

- Use the built-in variable **`blockDim.x`** for threads per block

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

- Combined version of `add()` to use parallel threads *and* parallel blocks

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

- Update the kernel launch:

```
add<<<N/M, M>>>(d_a, d_b, d_c);
```

48

Handling Arbitrary Vector Sizes

- Typical problems are not friendly multiples of `blockDim.x`

- Update the kernel launch:

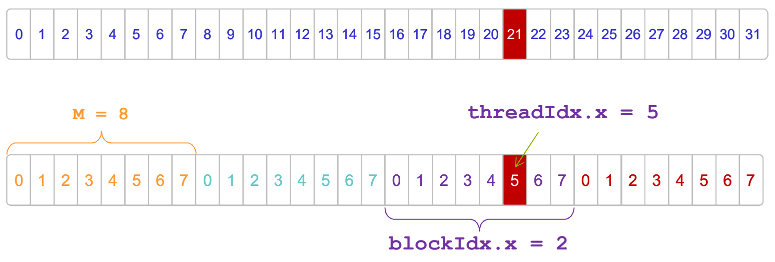
```
add<<<(N + M-1)/M, M>>>(d_a, d_b, d_c, N);
```

- Avoid accessing beyond the end of the arrays:

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

Indexing Arrays: Example

- Which thread will operate on the red element?



```
int index = threadIdx.x + blockIdx.x * M;
          =        5    +      2      * 8;
          = 21;
```

- `M` = Size of the block (number of threads in a block)

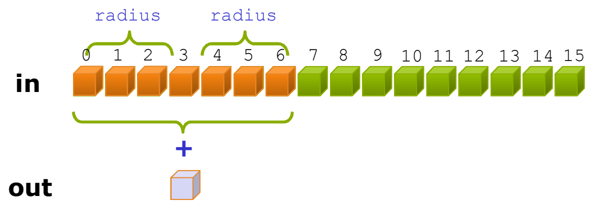
Cooperating Threads

- Unlike parallel blocks, threads have mechanisms to:
  - Communicate
  - Synchronize
- Between the blocks there is no mechanism for synchronization
- Synchronization is needed when threads share same memory and output of one of the threads is considered as input to another

51

1D Stencil

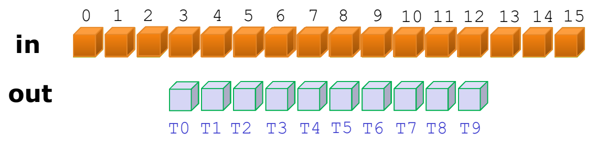
- Consider applying a 1D stencil to a 1D array of elements
- 1D Stencil: Each output element is the sum of input elements within a radius
- If radius is 3, then each output element is the sum of 7 input elements



52

1D Stencil

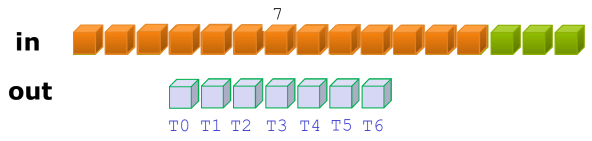
- Consider applying a 1D stencil to a 1D array of elements
- 1D Stencil: Each output element is the sum of input elements within a radius
- If radius is 3, then each output element is the sum of 7 input elements
- Each thread processes one output element
  - **`blockDim.x`** elements per block



53

1D Stencil

- Consider applying a 1D stencil to a 1D array of elements
- 1D Stencil: Each output element is the sum of input elements within a radius
- If radius is 3, then each output element is the sum of 7 input elements
- Each thread processes one output element
  - **`blockDim.x`** elements per block
- Input elements are read several times
  - Each element is read (2*radius+1) times
  - With radius 3, each input element is read seven times



54

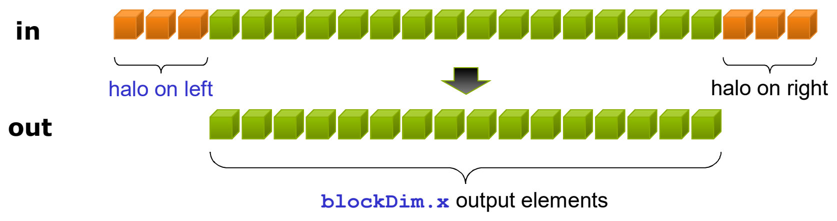
Sharing Data Between Threads

- Within a block, threads share data via shared memory
- By keeping data in shared memory, each thread can take data by avoiding again and again going to global memory
- Declare a data item as shared data item by using `__shared__`
- The data item will be shared within a block
- Data is visible only to the threads with in a block
- Note that this data item in shared memory is visible only to the threads within a block

55

Implementing With Shared Memory

- Cache data in shared memory
  - Read (`blockDim.x` + 2 * radius) input elements from global memory to shared memory
    - Each block needs a halo of radius elements at each boundary
  - Compute `blockDim.x` output elements
  - Write `blockDim.x` output elements to global memory



56

Stencil Kernel

```
BLOCK_SIZE = 16
RADIUS = 3
```

```
__global__ void stencil_1d(int *in, int *out, int RADIUS)
{
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;
```

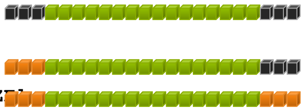
- `gindex` gives the index in global memory
  - It gives the unique thread index with in block
- `lindex` gives the index in the shared memory
  - RADIUS is added to align with RADIUS number of elements on the left

57

Stencil Kernel

```
BLOCK_SIZE = 16
RADIUS = 3
```

```
__global__ void stencil_1d(int *in, int *out, int RADIUS) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZ
  }
  // Apply the stencil
  int result = 0;
  for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];

  // Store the result
  out[gindex] = result;
}
```



58

Data Race!

- The stencil example will not work…
- Suppose thread 15 reads the halo before thread 0 has fetched it…

```
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
  temp[lindex – RADIUS = in[gindex – RADIUS];
  temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}

int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];
```

59

Data Race!

- The stencil example will not work…
- Suppose thread 15 reads the halo before thread 0 has fetched it…

Store at `temp[18]`

```
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
  temp[lindex – RADIUS = in[gindex – RADIUS];
  temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}

int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];
```

60

Data Race!

- The stencil example will not work…
- Suppose thread 15 reads the halo before thread 0 has fetched it…

```
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
  temp[lindex – RADIUS = in[gindex – RADIUS];
  temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}

int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];
```

Store at `temp[18]`

Skipped, as `threadIdx > RADIUS`

61

Data Race!

- The stencil example will not work…
- Suppose thread 15 reads the halo before thread 0 has fetched it…

```
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
  temp[lindex – RADIUS = in[gindex – RADIUS];
  temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}

int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];
```

Store at `temp[18]`

Skipped, as `threadIdx > RADIUS`

Load from `temp[19]`

62

Synchronize Threads

- `void __syncthreads();` is called with in kernel to synchronize the threads
- It acts as a barrier with in a block
- Any thread reaches the call `__syncthreads()` must wait till others threads reach that call
- This ensure all the valid data is available
- Synchronizes all threads within a block
  - Used to prevent RAW / WAR / WAW hazards
- All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block

63

Modified Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;
  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] =
      in[gindex + BLOCK_SIZE];
  }

  // Synchronize (ensure all the data is available)
  __syncthreads();

  // Apply the stencil
  int result = 0;
  for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];
  // Store the result
  out[gindex] = result;
}
```

64

Convolution on 1-dimensional Input

$$s_t = \sum_{a=0}^{6} x_{t-a} w_{-a}$$

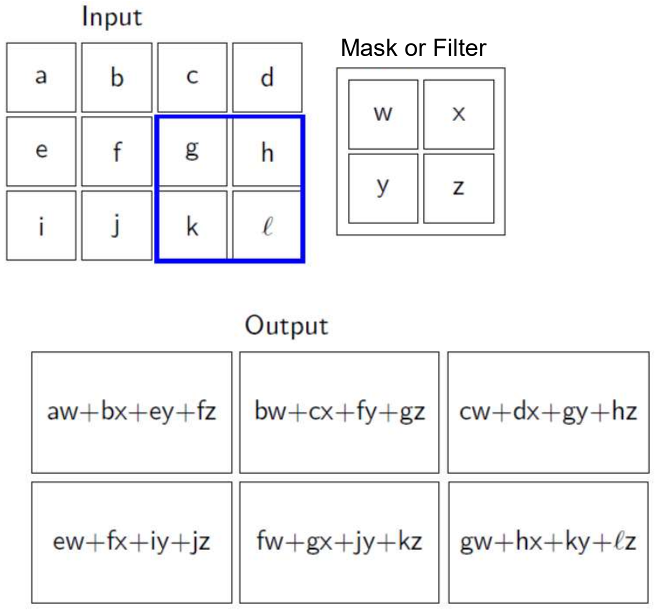
						w_{-6}	w_{-5}	w_{-4}	w_{-3}	w_{-2}	w_{-1}	w_0
W						0.01	0.01	0.02	0.02	1	0.4	0.5
X	1.00	1.10	1.20	1.40	1.70	1.80	1.90	2.10	2.20	2.40	2.50	2.70
S							1.80	1.96	2.11	2.16	2.28	2.42

$$s_6 = x_6 w_0 + x_5 w_{-1} + x_4 w_{-2} + x_3 w_{-3} + x_4 w_{-4} + x_5 w_{-5} + x_6 w_{-6}$$

- The weight array (W) is known as the filter
- We just slide the filter over the input and compute the value of s_t based on a window around x_t

65

Convolution on 2D Input



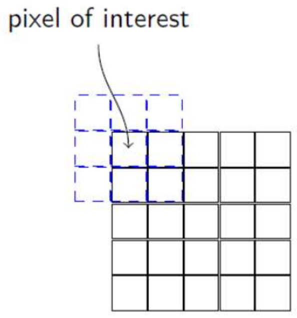
66

Convolution on 2D Input

- Formula for convolution:

$$S_{ij} = (I * K)_{ij} = \sum_{a=\lfloor -\frac{m}{2} \rfloor}^{\lfloor \frac{m}{2} \rfloor} \sum_{b=\lfloor -\frac{n}{2} \rfloor}^{\lfloor \frac{n}{2} \rfloor} I_{i-a,j-b} K_{\frac{m}{2}+a,\frac{n}{2}+b}$$

- m and n are the height and width of the mask
- In other words we will assume that the kernel is cantered on the pixel of interest
- So we will be looking at both preceding and succeeding neighbours



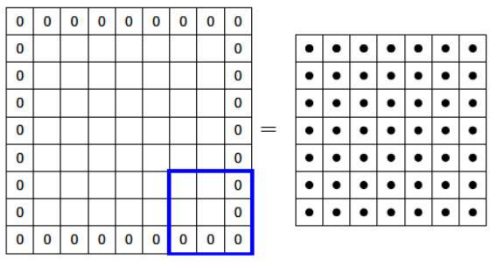
67

Convolution on 2D Input

- Formula for convolution:

$$S_{ij} = (I * K)_{ij} = \sum_{a=\lfloor -\frac{m}{2} \rfloor}^{\lfloor \frac{m}{2} \rfloor} \sum_{b=\lfloor -\frac{n}{2} \rfloor}^{\lfloor \frac{n}{2} \rfloor} I_{i-a,j-b} K_{\frac{m}{2}+a,\frac{n}{2}+b}$$

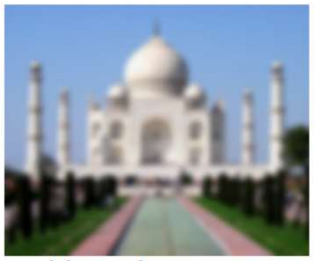
- m and n are the height and width of the mask
- In other words we will assume that the kernel is cantered on the pixel of interest
- So we will be looking at both preceding and succeeding neighbours



68

2D Convolutions on Images

$$* \quad \begin{matrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{matrix} \quad =$$



blurs the image

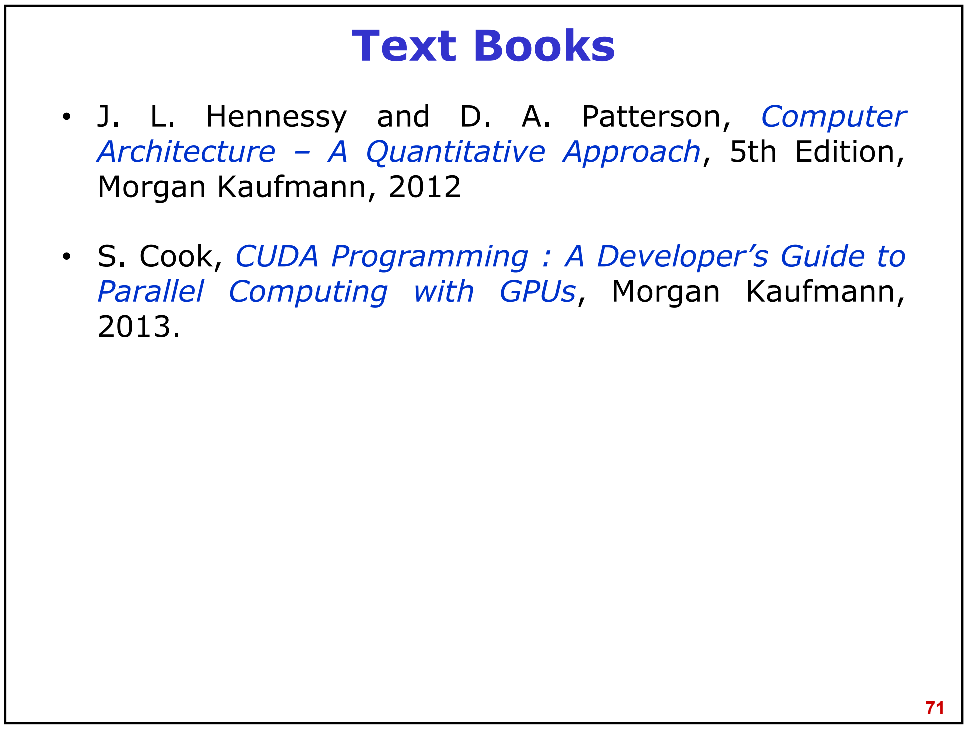
•

69

Assignment

- **Assignment 1:** *Extend the kernel function for 1-D stencil to 1-D convolution*
- **Assignment 2:**
  - *Extend the kernel function for 1-D convolution to 2-D convolution with radius 3 x 3 as in previous slide (as an application of blurring the image).*
  - *Write the kernel function (i) with shared memory and (ii) without shared memory*
- *Submit on or before Sunday, 23 August 2020, 10:00 PM.*

70



Text Books

- J. L. Hennessy and D. A. Patterson, *Computer Architecture – A Quantitative Approach*, 5th Edition, Morgan Kaufmann, 2012
- S. Cook, *CUDA Programming : A Developer's Guide to Parallel Computing with GPUs*, Morgan Kaufmann, 2013.

71