Unit-I

Objectives

- To learn about:
- GPU processor
- CUDA Hardware overview
- Memory Handling with CUDA
- CUDA Programming

Graphics Processing Unit

- **A graphics processing unit** (GPU), is similar CPU
- Designed specifically for performing the complex mathematical and geometric calculations that are necessary for graphics rendering.

Graphics Processing Unit

- A graphics processing unit (GPU) is a computer chip that performs rapid mathematical calculations, primarily for the purpose of rendering images.
- occasionally called **visual processing unit (VPU)**
- GPU is able to render images more quickly than a CPU because of its parallel processing architecture
- Nvidia introduced the first GPU, the GeForce 256, in 1999
- Others include AMD, Intel and ARM.
- In 2012, Nvidia released a virtualized GPU, which offloads graphics processing from the server CPU in a virtual desktop infrastructure.

Graphics Processing Unit

- GPUs are used in
  - Embedded Systems
  - Mobile phones
  - Personal computers
  - Workstations
  - Game consoles

GPU Vs CPU

- A GPU is tailored for highly parallel operation while a CPU executes programs serially.
- For this reason, GPUs have many parallel execution units and higher transistor counts, while CPUs have few execution units and higher clock speeds
- A GPU is for the most part deterministic in its operation
- GPUs have much deeper pipelines (several thousand stages vs 10-20 for CPUs)
- GPUs have significantly faster and more advanced memory interfaces as they need to shift around a lot more data than CPUs

High-end CPU-GPU Comparison

	Xeon 8180M	**Titan V**
Cores	28	5120 (+ 640)
Active threads	2 per core	32 per core
Frequency	2.5 (3.8) GHz	1.2 (1.45) GHz
Peak performance (SP)	4.1 TFlop/s	13.8 TFlop/s
Peak mem. bandwidth	119 GB/s	653 GB/s
Maximum power	205 W	250 W
Launch price	$13,000	$3000

Release dates
- Xeon: Q3'17
- Titan V: Q4'17

What are GPU's Growth?

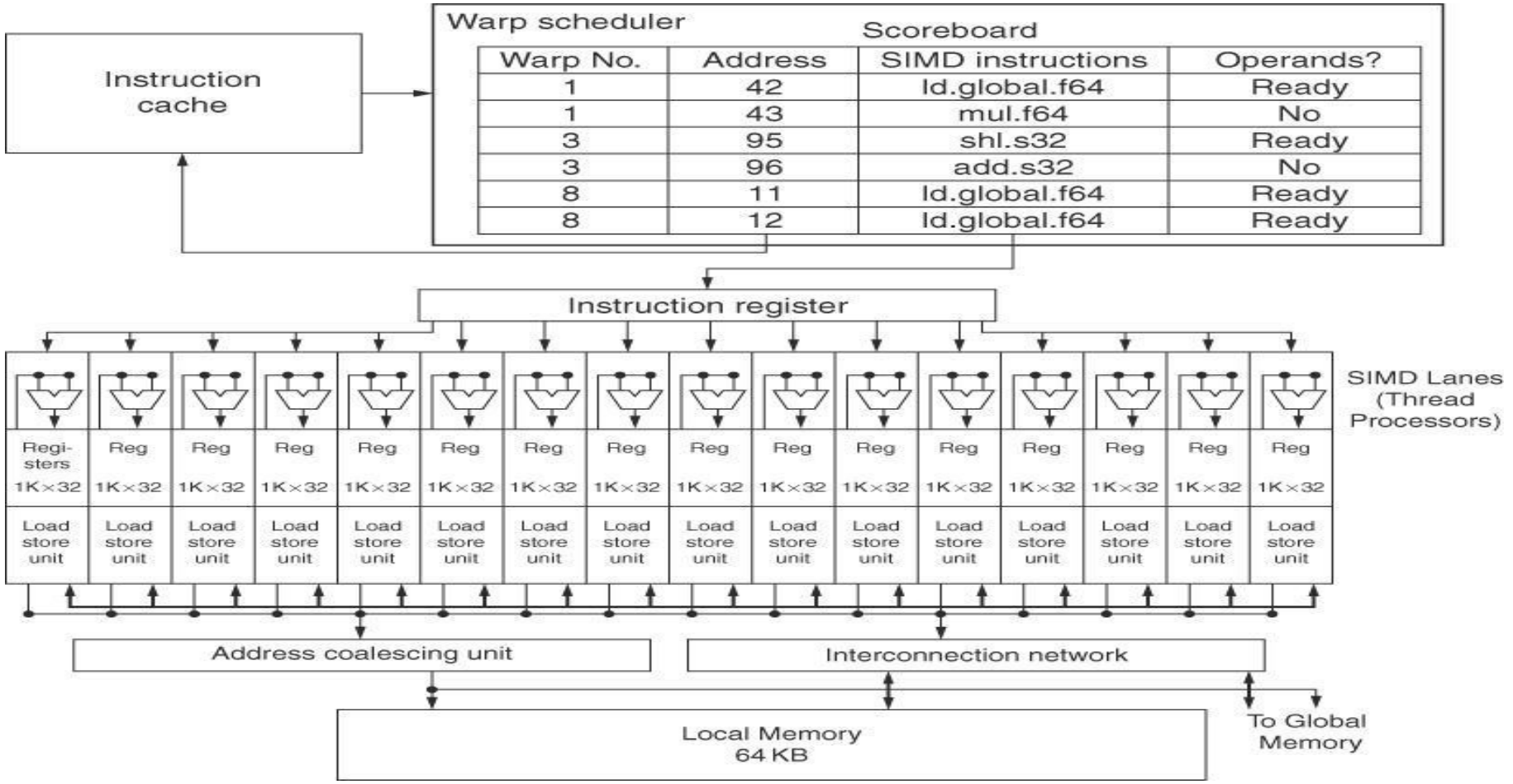
- Entertainment Industry has driven the economy of these chips?
  - people age 15-35 buy $15B in video games / year
- Moore's Law ++
- Simplified design (stream processing)
- Single-chip designs

GPU

- Very Efficient For
  - Fast Parallel Floating Point Processing
  - Single Instruction Multiple Data Operations
  - High Computation per Memory Access

- Not Efficient For
  - Double Precision
  - Logical Operations on Integer Data
  - Branching-Intensive Operations
  - Random Access, Memory-Intensive Operations

NVIDIA GPU – MTSIMD



NVIDIA GPU- MTSIMD

- GPU is a multiprocessor composed of MTSIMD processors.
- It is similar to vector processor but with many parallel FU's that are deeply pipelined.
- MTSIMD is a processor that executes code in the form of thread blocks.
- GPU H/W contains a collection of MTSIMD Processors execute a Grid of Thread Blocks.

NVIDIA GPU- MTSIMD

- GPU H/W has two levels of H/W schedulers

1. Thread Block Scheduler:

  - Thread block scheduler is similar to control unit in Vector processor
  - determine the no of thread blocks for a loop and allocates them to different MTSIMD processors.
  - ensures that thread blocks are assigned to the processors whose local memories have the corresponding data.

NVIDIA GPU- FERMI MTSIMD

2. SIMD Thread Scheduler:

- SIMD Thread scheduler has scoreboard logic
- It keeps track of 48 threads of SIMD instructions
- It tells that which thread of SIMD instructions are ready to run
- It sends those instructions to dispatch unit to be run on MTSIMD processor
- within a SIMD Processor, which schedules when threads of SIMD instructions should run

NVIDIA GPU- MTSIMD

- It has many parallel functional units
- SIMD Processors with separate PCs and are programmed using threads.
- Each MTSIMD Processor is assigned 512 elements of the vectors to work on
- SIMD processors have 32,768 registers
- Like vector processor these registers are logically divided across SIMD lanes.

NVIDIA GPU- MTSIMD

- Each SIMD Thread has 64 vector registers of 32 elements with 32 bit each.
- FERMI has 16 physical lanes each contain 2048 registers
- Thread Blocks would contain 512/32 = 16 SIMD threads.
- Each thread of SIMD instructions in this example compute 32 of the elements of the computation.

NVIDIA GPU- MTSIMD

- GPU applications have so many threads of SIMD instructions that multithreading can
  - hide the latency to DRAM
  - increase utilization of multithreaded SIMD Processors

NVIDA GPU ISA

- PTX(Parallel Thread Execution) provides a stable instruction set for GPUs
- H/W instruction set is hidden from the programmer
- PTX instructions describe the operations on a single CUDA thread
- PTX uses virtual registers
- Translation to machine code is performed in software

NVIDA GPU ISA

- Format of a PTX instruction is

opcode.type d, a, b, c;

– where d is the destination operand; a, b, and c are source operands

- Source operands are 32-bit or 64-bit registers or a constant value.
- Destinations are registers,except for store instructions.

SSN

NVIDA GPU ISA

- the operation type is one of the following:

Type	.type Specifier			
• Untyped bits 8, 16, 32, and 64 bits	. b8,	b16,	. b32,	b64
• Unsigned integer 8, 16, 32, and 64 bits	.U8,	. U16,	U32,	u64
• Signed integer 8, 16, 32, and 64 bits	. S8,	. S16,	. S32,	S64
• Floating Point 16, 32, and 64 bits	.J16, J32, J64			

Conditional Branching

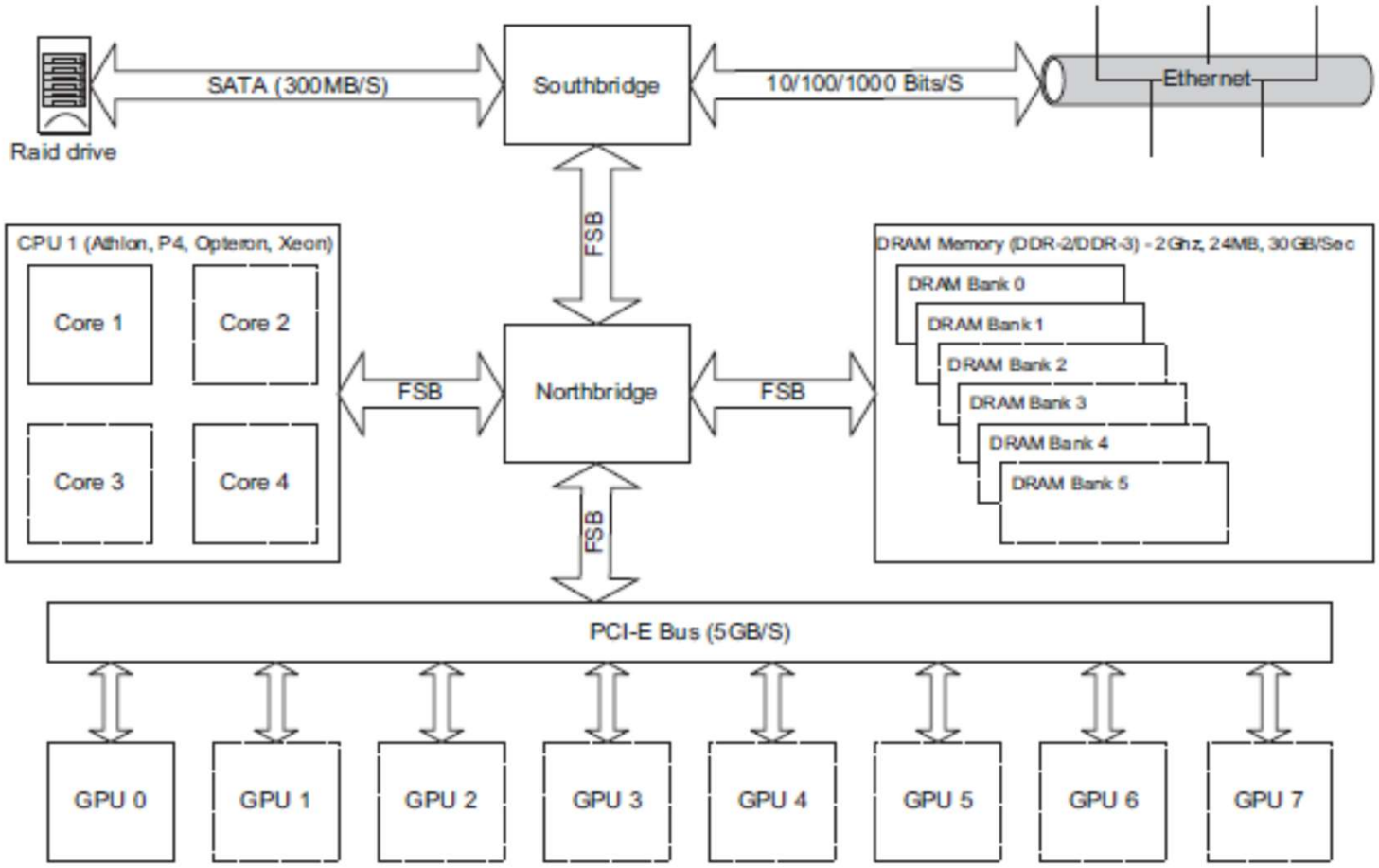
- Like vector architectures, GPU branch hardware uses internal masks
- Also uses
  - Branch synchronization stack
  - Entries consist of masks for each SIMD lane
  - i.e. which threads commit their results

- Per-thread-lane 1-bit predicate register, specified by programmer

Core 2 series Architecture

- All GPU devices are connected to the processor via the PCI-E bus.
- To get data from the processor, we need to go through the Northbridge device over the slow FSB (front-side bus).
- The FSB can run up to 1600 MHz clock rate.
- This is typically one-third of the clock rate of a fast processor.
- Memory is accessed through the Northbridge.
- Peripherals through the Northbridge and Southbridge chipset.

Core 2 series Architecture



Core 2 series Architecture

- The Northbridge deals with all the high-speed components like memory, CPU, PCI-E bus connections, etc.
- The Southbridge chip deals with the slower devices such as hard disks, USB, keyboard, network connections, etc.
- PCI-E-Peripheral Communications Interconnect Express is a bus.
  - it's based on guaranteed bandwidth.
- In the old PCI system each component could use the full bandwidth of the bus, but only one device at a time.
- The more cards you add, the less available bandwidth each card would receive.

Core 2 series Architecture

- PCI-E solved this problem by the introduction of PCI-E lanes.
  - These are high-speed serial links that can be combined together to form X1, X2, X4, X8, or X16 links.
- We have a 5 GB/s full-duplex bus, meaning we get the same upload and download speed, at the same time.
- we can transfer 5 GB/ s to the card, while at the same time receiving 5 GB/s from the card.
- this does not mean we can transfer 10 GB/s to the card if we're not receiving any data (i.e., the bandwidth is not cumulative).

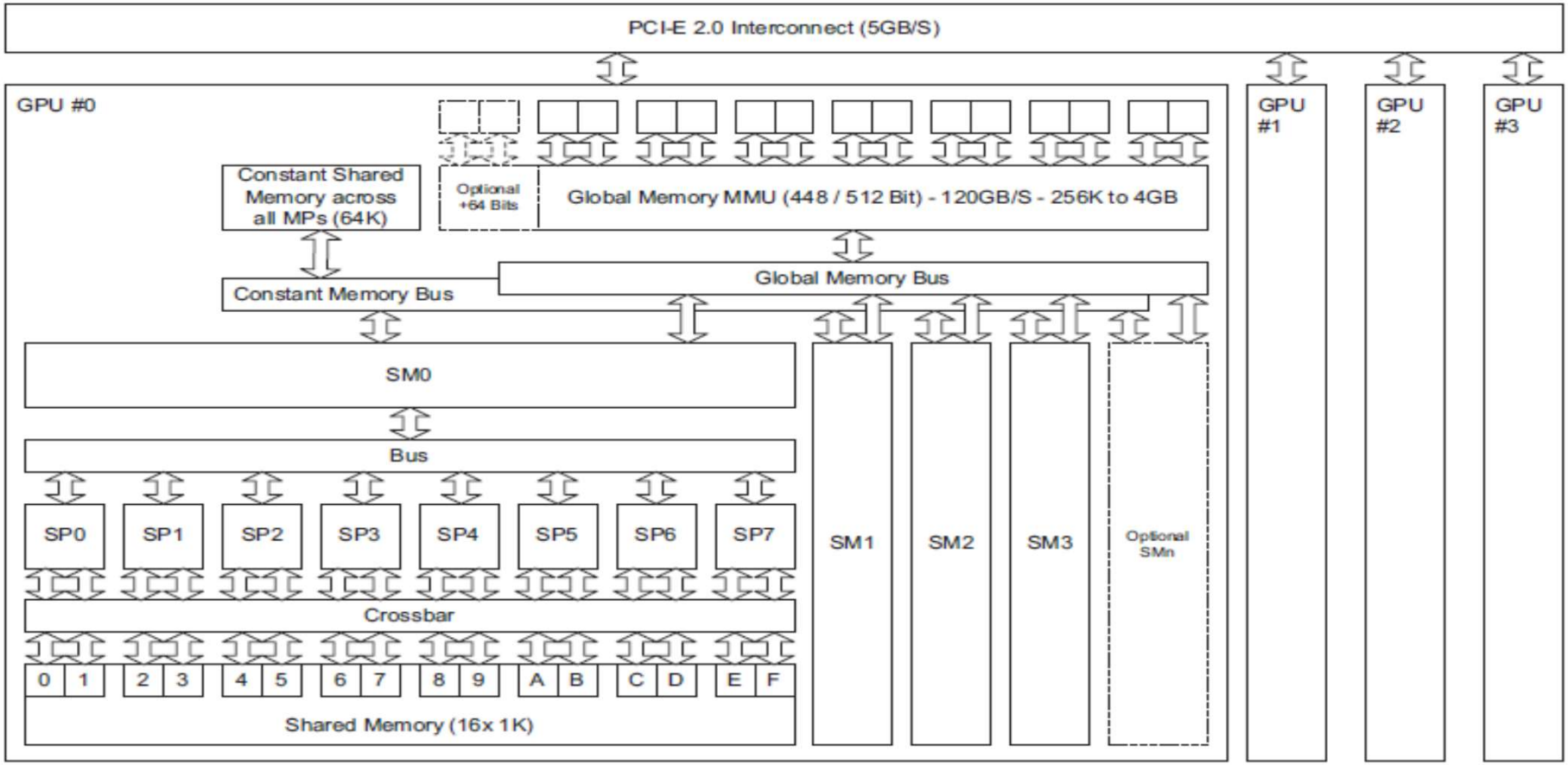
Core 2 series Architecture

- In MPI the latency can be considerable if the Ethernet connections are attached to the Southbridge instead of the PCI-E bus.
- Dedicated high-speed interconnects like InfiniBand or 10 Gigabit Ethernet cards are often used on the PCI-E bus.
- Nehalem architecture brought us QPI (Quick Path Interconnect), which was actually a huge advance over the FSB (Front Side Bus).
- QPI is a high-speed interconnect that can be used to talk to other devices or CPUs.

GPU HARDWARE

- GPU hardware is radically different than CPU hardware.
- The GPU hardware consists of a number of key blocks:
  - Memory (global, constant, shared)
  - Streaming multiprocessors (SMs)
  - Streaming processors (SPs)
- GPU is really an array of SMs, each of which has N cores
  - 8 in G80 and GT200, 32–48 in Fermi, 8 plus in Kepler

GPU HARDWARE



: Block diagram of a GPU (G80/GT200) card.

GPU HARDWARE

- A GPU device consists of one or more SMs.
- Add more SMs to the device and you make the GPU able to process more tasks at the same time, or the same task quicker, if you have enough parallelism in the task.
- NVIDIA hardware will increase in performance by growing a combination of the number of SMs and number of cores per SM.
- There are multiple SPs in each SM. There are 8 SPs shown here; in Fermi this grows to 32–48 SPs and in Kepler to 192.

GPU HARDWARE

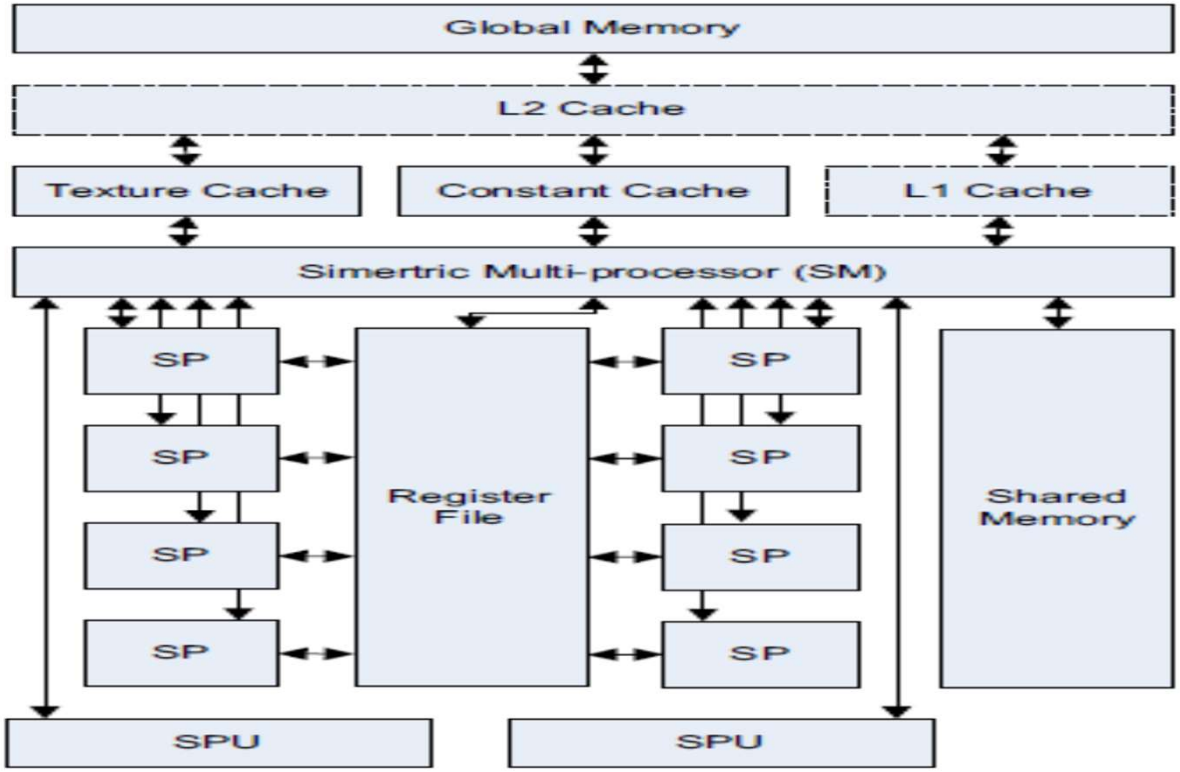


Fig: Inside an SM.

GPU HARDWARE

- Each SM has access to a register file
  - like a chunk of memory that runs at the same speed as the SP units
  - so there is effectively zero wait time on this memory.
- It is used for storing the registers in use within the threads running on an SP.
- There is shared memory block accessible only to the individual SM; this can be used as a program-managed cache.
- Each SM has a separate bus into the texture memory, constant memory, and global memory spaces.
- Texture memory is a special view onto the global memory, which is useful for data where there is interpolation,
  - Ex: 2D or 3D lookup tables.

GPU HARDWARE

- Constant memory is used for read-only data and is cached on all hardware revisions.
- Like texture memory, constant memory is simply a view into the main global memory.
- Global memory is supplied via GDDR (Graphic Double Data Rate) on the graphics card.
- This is a high-performance version of DDR(Double DataRate) memory.
- Memory bus width can be up to 512 bits wide, giving a bandwidth of 5 to 10 times more than found on CPUs, up to 190 GB/s with the Fermi hardware.

GPU HARDWARE

- Each SM also has two or more special-purpose units (SPUs)
- It perform special hardware instructions, such as the high-speed 24-bit sin/cosine/exponent operations.
- Double-precision units are also present on GT200 and Fermi hardware

CUDA COMPUTE LEVELS

- CUDA supports a number of compute levels.
- A full list of the differences between each compute level can be found in the NVIDIA CUDA Programming Guide

CUDA COMPUTE LEVELS

Compute 1.0

- Compute level 1.0 is found on the older graphics cards.
  - Ex: the original 8800 Ultras and many of the 8000 series cards as well as the Tesla C/D/S870s.
- The main features lacking in compute 1.0 cards are those for atomic operations.
- Atomic operations are those where we can guarantee a complete operation without any other thread interrupting.
- The hardware implements a barrier point at the entry of the atomic function and guarantees the completion of the operation (add, sub, min, max, logical and, or, xor, etc.) as one operation.
- Compute 1.0 cards are effectively now obsolete.

CUDA COMPUTE LEVELS

Compute 1.1

- Compute level 1.1 is found in many of the later shipping 9000 series cards, such as the 9800 GTX, which were extremely popular..
- One major change brought in with compute 1.1, is overlapped data transfer and kernel execution.
- The SDK call to cudaGetDeviceProperties() returns the deviceOverlap property, which defines if this functionality is available.
- This allows important optimization called **double buffering**.

CUDA COMPUTE LEVELS

Compute 1.1

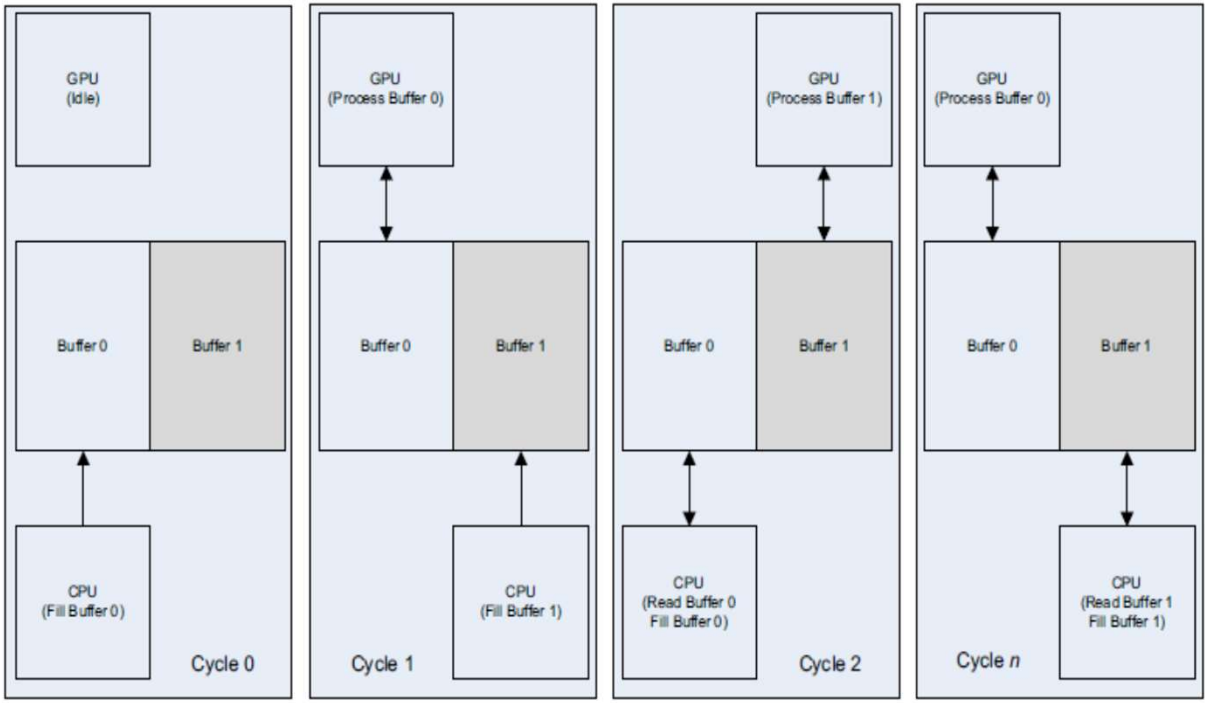


Fig: Double buffering with a single GPU

CUDA COMPUTE LEVELS

Compute 1.1

- Cycle 0: Having allocated two areas of memory in the GPU memory space, the CPU fills the first buffer.
- Cycle 1:
- a)The CPU then invokes a CUDA kernel (a GPU task) on the GPU, which returns immediately to the CPU.
- b)The CPU then fetches the next data packet, from a disk, the network, or wherever.
- c)Meanwhile, the GPU is processing away in the background on the data packet provided.
- When the CPU is ready, it starts filling the other buffer.

CUDA COMPUTE LEVELS

Compute 1.1

- Cycle 2: When the CPU is done filling the buffer, it invokes a kernel to process buffer 1.
- It then checks if the kernel from cycle 1, which was processing buffer 0, has completed.
- If not, it waits until this kernel has finished and then fetches the data from buffer 0 and then loads the next data block into the same buffer.
- During this time the kernel kicked off at the start of the cycle is processing data on the GPU in buffer 1.
- Cycle N:We then repeat cycle 2, alternating between which buffer we read and write to on the CPU with the buffer being processed on the GPU.

CUDA COMPUTE LEVELS

Compute 1.2

- Compute 1.2 devices appeared with the low-end GT200 series hardware. These were the initial GTX260 and GTX280 cards.
- With the GT200 series hardware, NVIDIA approximately doubled the number of CUDA core processors on a single card.
- NVIDIA increased the number of concurrent warps a multiprocessor could execute from 24 to 32.
- Warps are blocks of code that execute within a multiprocessor
- Issues with restrictions on coalesced access to the global memory bank conflicts in the shared memory found in compute 1.0 and compute 1.1 devices were greatly reduced.
- This make the GT200 series hardware far easier to program and it greatly improved the performance.

CUDA COMPUTE LEVELS

Compute 1.3

- The compute 1.3 devices were introduced with the move from GT200 to the GT200 a/b revisions of the hardware. Almost all higher-end cards from this era were compute 1.3 compatible.
- The major change that occurs with compute 1.3 hardware is the introduction of support for limited double-precision calculations.
- GPUs are primarily aimed at graphics and here there is a huge need for fast single-precision calculations, but limited need for double-precision ones.
- Typically, you see an order of magnitude drop in performance using double-precision as opposed to single-precision floating-point operations,.

CUDA COMPUTE LEVELS

Compute 2.0

- Compute 2.0 devices saw the switch to Fermi hardware.
- The original guide for tuning applications for the Fermi architecture can be found on the NVIDIA website at **http://developer.nvidia.com/cuda/**
- Some of the main changes in compute 2.x hardware are as follows:
  - Introduction of 16 K to 48 K of L1 cache memory on each SP.
  - Introduction of a shared L2 cache for all SMs.
  - Support in Tesla-based devices for ECC (Error Correcting Code)-based memory checking and error correction.
  - Support in Tesla-based devices for dual-copy engines.
  - Extension in size of the shared memory from 16 K per SM up to 48 K per SM.
  - For optimum coalescing of data, it must be 128-byte aligned.
  - The number of shared memory banks increased from 16 to 32.

CUDA COMPUTE LEVELS

Compute 2.0

- First, the introduction of the L1 cache. It is the fastest cache type. Compute 1.x hardware has no cache, except for the texture and constant memory caches.
- The introduction of a cache makes it much easier for many programmers to write programs that work well on GPU hardware
- To exploit the cache, the application either needs to have a sequential memory pattern or have at least some data reuse.
- The L2 cache is up to 768 K in size on Fermi and, importantly, is a unified cache.
- It is shared and provides a consistent view for all the SMs. This allows for much faster interblock communication through global atomic operations.

CUDA COMPUTE LEVELS

Compute 2.0

- Support for ECC memory is a must for data centres.
- ECC memory provides for automatic error detection and correction.
- ECC detects and corrects single-bit upset conditions that you may find in large data centers.
- The increase of shared memory banks from 16 to 32 bits.
- This is a major benefit over the previous generations.
- It allows each thread of the current warp (32 threads) to write to exactly one bank of 32 bits in the shared memory without causing a shared bank conflict.

CUDA COMPUTE LEVELS

Compute 2.1

- Compute 2.1 is seen on certain devices aimed specifically at the games market, such as the GTX460 and GTX560
- 48 CUDA cores per SM instead of the usual 32 per SM.
- Eight single-precision, special-function units for transcendental per SM instead of the usual four.
- Dual-warp dispatcher instead of the usual single-warp dispatcher.
- On compute 2.1 hardware, instead of the usual two instruction dispatchers per two clock cycles, we now have four.
- In the hardware, there are three banks of 16 CUDA cores, 48 CUDA cores in total, instead of the usual.

CUDA Programming

CUDA

CUDA -Compute Unified DeviceArchitecture

- is a parallel computing platform and programming model created by NVIDIA
- Implemented by the GPUs
- CUDA gives developers access to the instruction set and memory of the parallel computational elements in CUDA GPUs.
- Using CUDA, GPUs become accessible for computation like CPUs.

CUDA

- functions for the GPU (device) and functions for the system processor (host),
- CUDA uses _device_ or _global_ for GPU and -
  -host- for the CPU.
- CUDA variables declared in the device or global functions are allocated to the GPU Memory

CUDA Philosophy

SIMT philosophy

- Single Instruction Multiple Thread

Computationally intensive

- The time spent on computation significantly exceeds the time spent on transferring data to and from GPU memory

Massively parallel

- The computations can be broken down into hundreds or thousands of independent units of work

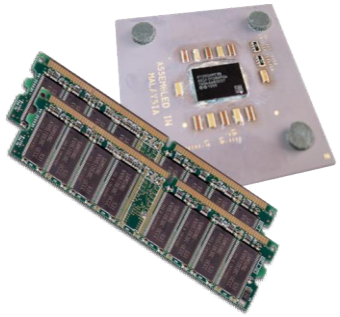
Definitions

- dimGrid - dimensions of the code (in blocks)
- dimBlock- dimensions of a block (in threads)
- *Blockldx*- identifier for blocks
- *threadldx*- identifier for threads per block
- *blockDim*- number of threads per block

Heterogeneous Computing

Host

- CPU and its memory (host memory)



Device

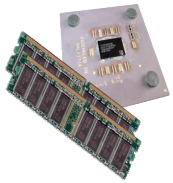
- GPU and its memory (device memory)

SSN

Heterogeneous Computing

```
#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;
}
```

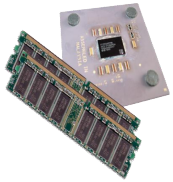
parallel fn



serial code

parallel code

serial code



Heterogeneous Computing

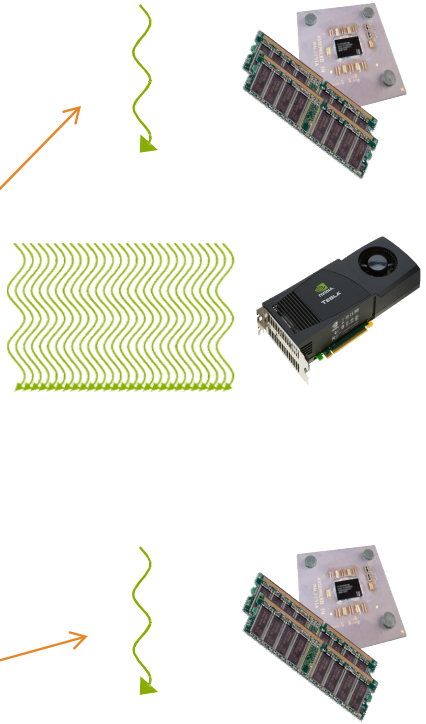
```
#include <iostream>
#include <algorithm>

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;
}
```

parallel fn

serial code

parallel code

serial code



Hello World with CUDA

```
#include <stdio.h>
#include <cuda.h>

__global__ void hwkernel() {
  printf("Hello world!\n");
}

int main() {
  hwkernel<<<1, 1>>>();
}
```

```
$ nvcc hello-world.cu

$./a.out

$
```

Hello World with CUDA

```
#include <stdio.h>
#include <cuda.h>

__global__ void hwkernel() {
printf("Hello world!\n");
}

int main() {
 hwkernel<<<1, 1>>>();
 cudaDeviceSynchronize()
}
```

```
$ nvcc hello-world.cu

$./a.out
Hello world!

$
```

Program returns immediately after launching the kernel. To prevent program to finish before kernel is completed, we call **`cudaDeviceSynchronize().`**

Hello World with CUDA

```
#include <stdio.h>
#include <cuda.h>

_global___void hwkernel() {
printf(“Hello world!\n”);
}

int main() {  hwkernel<<<1,
32>>>();
cudaThreadSynchronize();
}
```

```
$ nvcc hello-world.cu

$./a.out
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
…
…
$
```

How NVCC works?

- Nvcc is a driver program
  - Compiles and links all input files
  - Requires a general-purpose C/C++ host compiler
    - Uses gcc and g++ by default on Linux platforms
  - `nvcc --version`

https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html

SSN

CUDA Compilation Trajectory

- Conceptually, the flow is as follows
  - Input program is preprocessed for device compilation
  - It is compiled to a CUDA binary and/or PTX (Parallel Thread Execution) intermediate code which are encoded in a fatbinary
- Input program is processed for compilation of the host code
  - CUDA-specific C++ constructs are transformed to standard C++ code
  - Synthesized host code and the embedded fatbinary are linked together to generate the executable

Function Declarations in CUDA

	Executed on	Callable from
`__device__float deviceFunc()`	Device	Device
`__global__void kernelFunc()`	Device	Host
`__host__float hostFunc()`	Host	Host

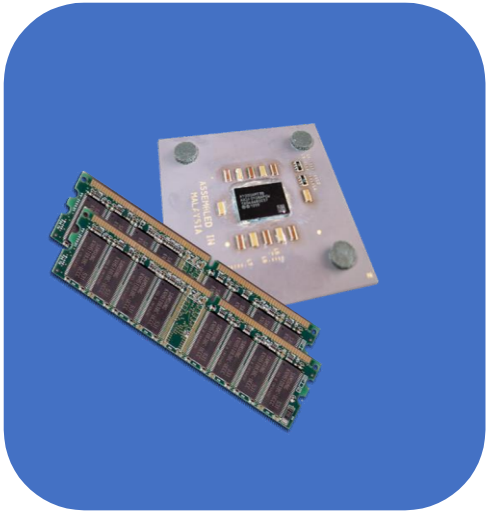
- __global__ define a kernel function, must return void
- __device__ functions can have return values
- __host__ is default, and can be omitted
- Prepending __host__ __device__ causes the system to compile separate host and device versions of the function

Dynamic Parallelism

- It is possible to launch kernels from other kernels
- Calling `__global__` functions from the device is referred to as dynamic parallelism
  - Requires CUDA devices of compute capability 3.5 and CUDA 5.0 or higher

Execution Model

Host
(serial
execution)



Serial code on host

Parallel kernel on device

Serial code on host

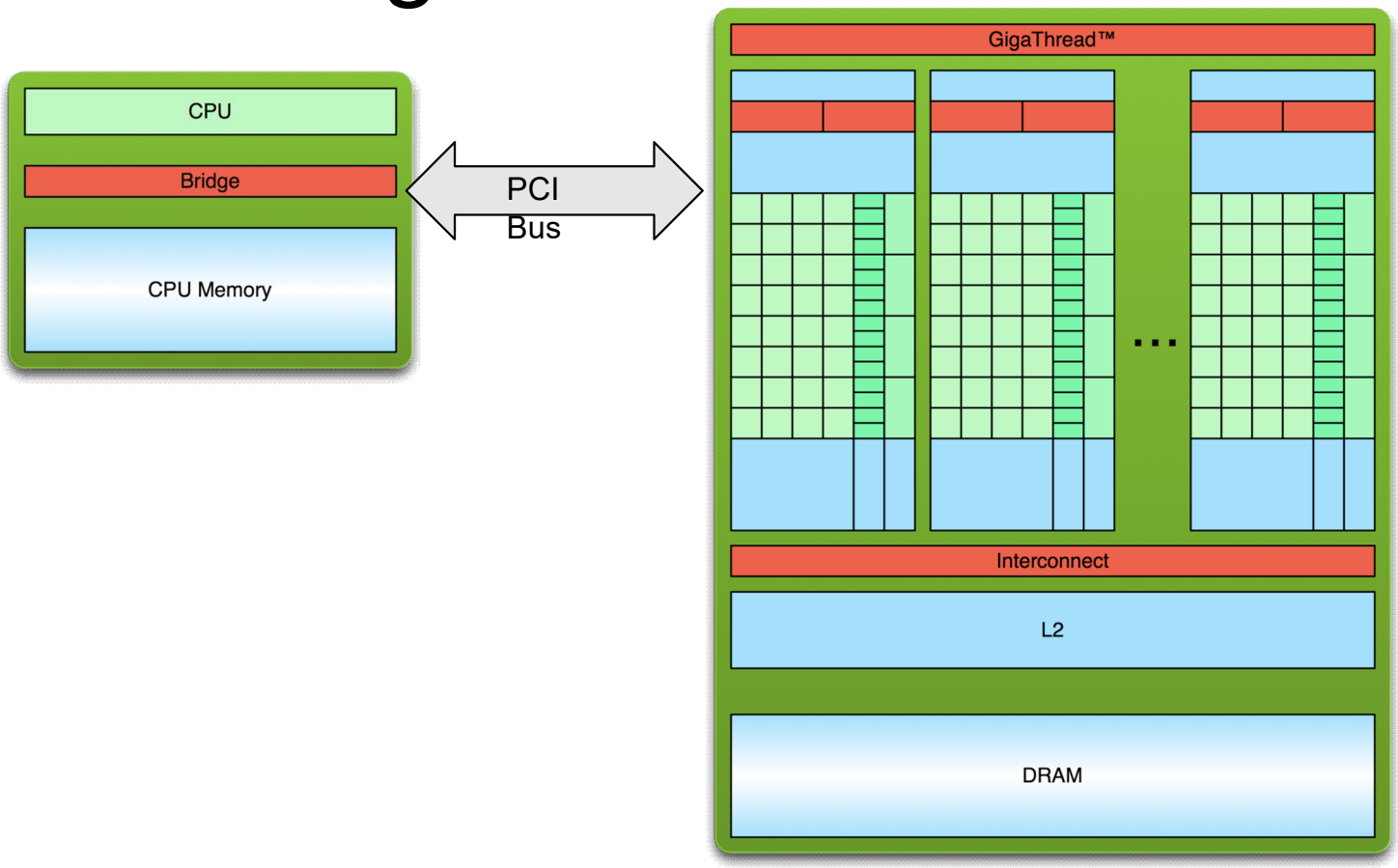
Parallel kernel on

device

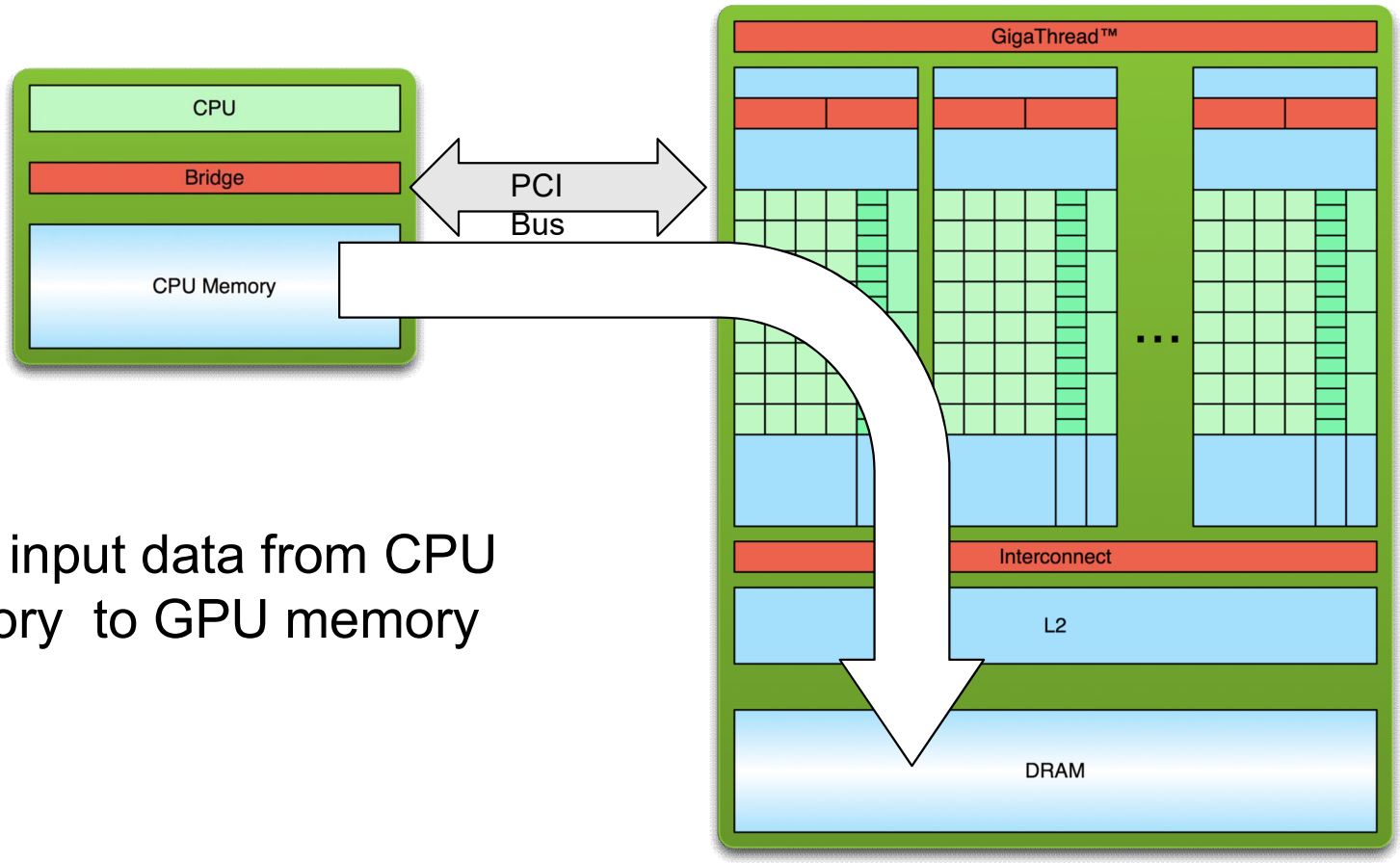
Device
(Parallel
execution)

SSN

Simple Processing Flow

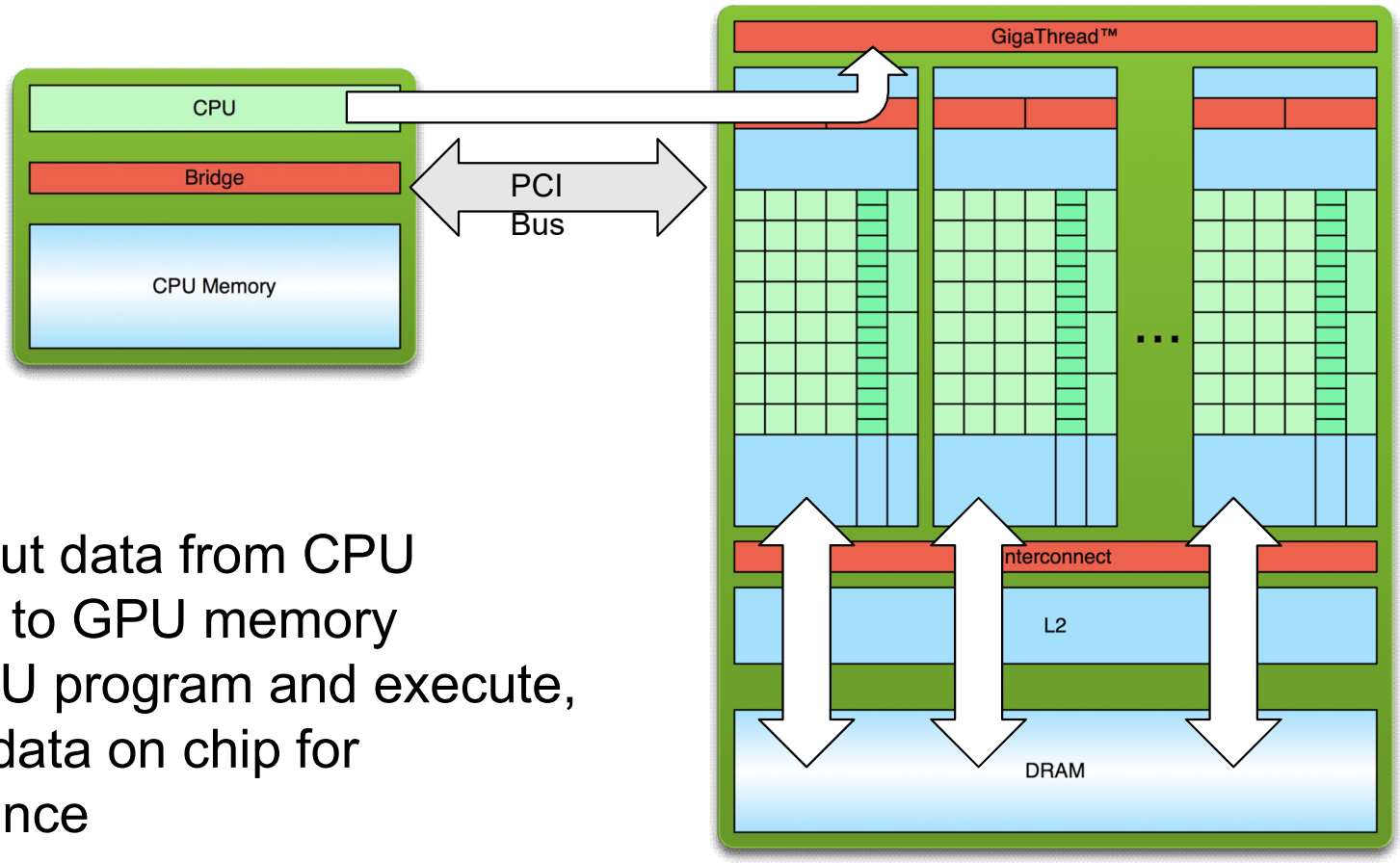


Simple Processing Flow



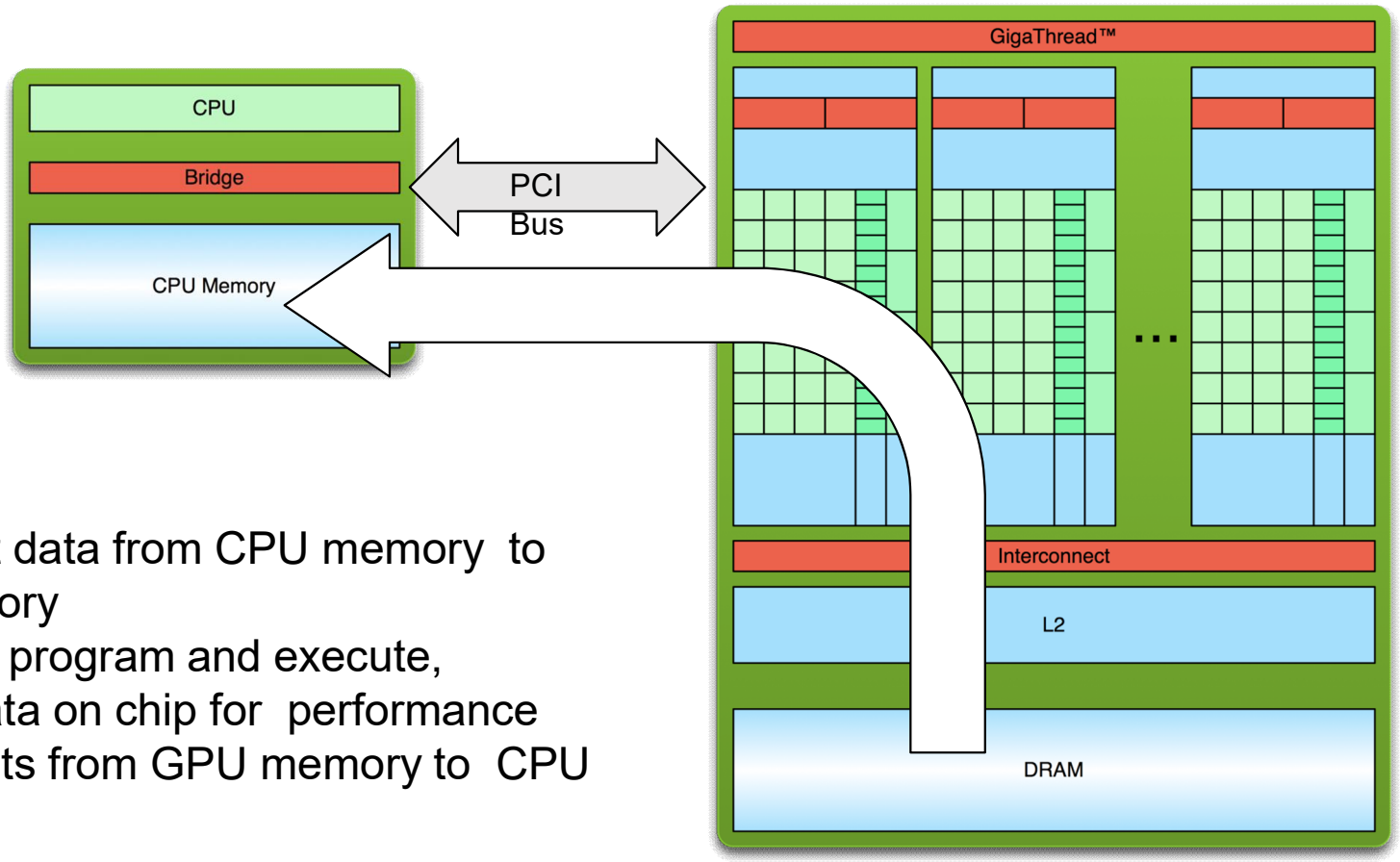
1. Copy input data from CPU memory to GPU memory

Simple Processing Flow



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

Simple 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

Vector Addition Example

```
__global__void VecAdd(float* A, float* B,
        float* C, int N) {
  int i = blockDim.x * blockIdx.x + threadI
dx.x;
  if (i < N)
    C[i] = A[i] + B[i];
}

int main() {
  …
  float*   h_A   =   (float*)malloc(size);
  float*   h_B   =   (float*)malloc(size);
  float* h_C = (float*)malloc(size);
```

```
float* d_A;
cudaMalloc(&d_A, size);
float* d_B;
cudaMalloc(&d_B, size);
float* d_C;
cudaMalloc(&d_C, size);

// Copy vectors from host memory to
// device memory
cudaMemcpy(d_A, h_A, size,
           cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size,
           cudaMemcpyHostToDevice);
```

Vector Addition Example

```
  // Invoke kernel
  int threadsPerBlock = 256;
  int blocksPerGrid = N/threadsPerBlock;
  VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
```

```
// Copy result from device memory to
// host memory  cudaMemcpy(h_C,
  d_C, size,
                cudaMemcpyDeviceToHost);
  …
  cudaFree(d_A);
  cudaFree(d_B);
  cudaFree(d_C);
…
}
```

Typical CUDA Program Flow

1. Load data into CPU memory
   - `fread/rand`
2. Copy data from CPU to GPU memory
   - `cudaMemcpy(..., cudaMemcpyHostToDevice)`
3. Call GPU kernel
   - `yourkernel<<<x, y>>>(...)`
4. Copy results from GPU to CPU memory.
   - `cudaMemcpy(..., cudaMemcpyDeviceToHost)`
5. Use results on CPU

CUDA Extensions for C/C++

- Kernel launch
  - Calling functions on GPU
- Memory management
  - GPU memory allocation, copying data to/from GPU
- Declaration qualifiers
  - `__device_ , _ shared,_ local, _ global_ ,_ host___`
- Special instructions
  - Barriers, fences, etc.
- Keywords
  - `threadIdx, blockIdx, blockDim`

Grids , Blocks, & Threads

Threads

- A thread is the fundamental building block of a parallel program.
- Parallelism in the CPU domain tends to be driven by the desire to run more than one (single-threaded) program on a single CPU. This is the task-level parallelism**(TLP).**
- Programs, which are data intensive, like video encoding, use the data parallelism model and split the task in N parts where N is the number of CPU cores available. Each CPU core calculate one “frame” of data where there are no interdependencies between frames**(DLP)**
- We can split each frame into N segments and allocate each one of the segments to an individual core.
- In the GPU domain, you see exactly these choices when attempting to speed up rendering of 3D worlds in computer games by using more than one GPU. You can send complete, alternate frames to each GPU.

Threads

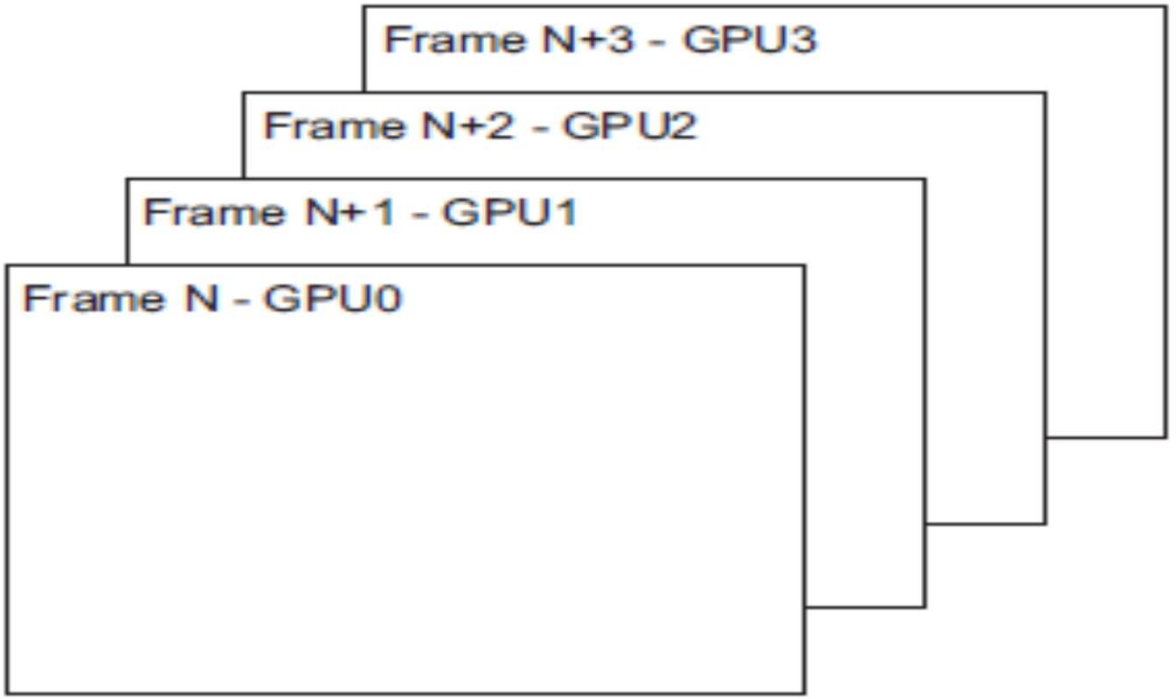


Fig: Alternate frame rendering (AFR) vs. Split Frame Rendering (SFR).

Threads

- **CPU**s support threads, but with a large overhead and thus are considered to be useful for more coarse- grained parallelism problems.
- **CPU**s, unlike **GPU**s, follow the **MIMD** (Multiple Instruction Multiple Data) model in that they support multiple independent instruction streams.
- Ex: a digital photo where you apply an image correction function to increase the brightness.
  - On a GPU you might choose to assign one thread for every pixel in the image.
  - On a quad-core CPU, you would likely assign one-quarter of the image to each CPU core.

Threads

- **CPUs Vs GPUs:**
- GPUs and CPUs are architecturally very different devices.
  - CPUs are designed for running a small number of potentially quite complex tasks.
  - GPUs are designed for running a large number of quite simple tasks.
- The CPU design is aimed at systems that execute a number of discrete and unconnected tasks.
- The GPU design is aimed at problems that can be broken down into thousands of tiny fragments and worked on individually.
- CPUs are very suitable for running operating systems and application software where there are a vast variety of tasks a computer may be performing at any given time

Threads

- **CPUs Vs GPUs:**

- CPUs and GPUs consequently support threads in very different ways.
  - **CPU** has a small number of registers per core that must be used to execute any given task.
  - To achieve this, they rapidly context switch between tasks.
  - Context switching on CPUs is expensive in terms of time, in that the entire register set must be saved to RAM and the next one restored from RAM.
  - **GPUs**, by comparison, also use the same concept of context switching, but instead of having a single set of registers, they have multiple banks of registers.
  - Consequently, a context switch simply involves setting a bank selector to switch in and out the current set of registers, which is several orders of magnitude faster than having to save to RAM.

Threads

- **CPUs Vs GPUs:**
- Both CPUs and GPUs must deal with stall conditions.
  - These are generally caused by I/O operations and memory fetches.
  - The CPU does this by context switching.
  - As the number of threads increases, the percentage of time spent context switching becomes increasingly large and the efficiency starts to rapidly drop off.
  - GPUs are designed to handle stall conditions and expect this to happen with high frequency.
  - The GPU model is a data-parallel one and thus it needs thousands of threads to work efficiently.
  - when it hits a memory fetch operation or has to wait on the result of a calculation, the streaming processors simply switch to another instruction stream and return to the stalled instruction stream sometime later.

Threads

- **CPUs Vs GPUs**
- One of the major differences between CPUs and GPUs is the sheer number of processors on each device.
- CPUs are typically dual- or quad-core devices. That is to say they have a number of execution cores available to run programs on.
- The current Fermi GPUs have 16 SMs, which can be thought of a lot like CPU cores.
- CPUs often run single-thread programs, meaning they calculate just a single data point per core, per iteration.
- GPUs run in parallel by default. Thus, instead of calculating just a single data point per SM, GPUs calculate 32 per SM.
- This gives a 4 times advantage in terms of number of cores (SMs) over a typical quad core CPU, but also a 32 times advantage in terms of data throughput..

Aspect	CPUs	GPUs
Parallelism Model	MIMD (Multiple Instruction Multiple Data)	SIMT (Single Instruction Multiple Thread)
Task Type	Coarse-grained tasks	Fine-grained, data-parallel tasks
Register Handling	Few registers per core, context switching	Multiple banks of registers, efficient context switching
Context Switching	Expensive due to register switching	Efficient due to multiple register banks
Stall Handling	May suffer reduced efficiency with many threads due to context switching	Designed to handle frequent stalls, switches to other tasks efficiently
Number of Processors	Dual- or quad-core typically	Multiple SMs (Streaming Multiprocessors), each with many cores
Single-thread Performance	Runs single-threaded programs	Runs multiple threads in parallel by default
Data Throughput Advantage	Limited by the number of cores, lower data throughput	High data throughput, calculates multiple data points per SM
Efficiency with Threads	Efficiency drops with increasing context switching	Designed to efficiently handle thousands of threads simultaneously
Use Cases	Ideal for general-purpose computing, running diverse tasks	Suited for data-parallel tasks with a large number of computations

Threading on GPUs

- look at a section of code and see what this means from a programming perspective.

```
void some_func(void)
{
int i;
for (i=0;i<128;i++)
{
        a[i] = b[i] * c[i];
}
}.
```

- It stores the result of a multiplication of b and c value for a given index in the result variable a for that same index. The for loop iterates 128 times (indexes 0 to 127).
- In CUDA you could translate this to 128 threads, each of which executes the line

```
a[i] = b[i] * c[i];
```

Threading on GPUs

- There is no dependency between one iteration of the loop .
- we can transform this into a parallel program easily.
- This is called loop parallelization and is very much the basis for one of the more popular parallel language extensions, OpenMP.
- On a quad-core CPU you could also translate this to four blocks, where CPU core 1 handles indexes 0–31,

  core 2 handles indexes 32–63,

  core 3 handles indexes 64–95,

  & core 4 handles indexes 96–127.

Threading on GPUs

- In CUDA, translate this loop by creating a kernel function, which is a function that executes on the GPU.
- In CUDA the CPU handles the serial code execution which is where it excels.
- When you come to a computationally intense section of code the CPU hands it over to the GPU.
- Applications that used a large amount of floating-point math ran many times faster on machines fitted with such coprocessors.
- GPUs are used to accelerate computationally intensive sections of a program

Threading on GPUs

- The GPU kernel function:

```
__global__ void some_kernel_func(int * const a, const int * const b, const int *
const c)
        {
                a[i] = b[i] * c[i];
        }
```

- __global__ prefix added to the C function that tells the compiler to generate GPU code and not CPU code when compiling this function, and to make that GPU code globally visible from within the CPU.

Threading on GPUs

- The CPU and GPU have separate memory spaces, meaning you cannot access CPU parameters in the GPU code and vice versa.
- The global arrays a, b, and c at the CPU level are no longer visible on the GPU level.
- You have to declare memory space on the GPU, copy over the arrays from the CPU, and pass the kernel function pointers to the GPU memory space to both read and write from.
- CUDA provides a special parameter, different for each thread, which defines the thread ID or number.
- You can use this to directly index into the array. This is very similar to MPI, where you get the process rank for each process.
- The thread information is provided in a structure i.e thread_idx .

Threading on GPUs

```
__global__ void some_kernel_func(int * const a, const int * const b, const int * const c)
        {
                const unsigned int thread_idx = threadIdx.x;
                a[thread_idx] =  b[thread_idx] * c[thread_idx];
        }
```

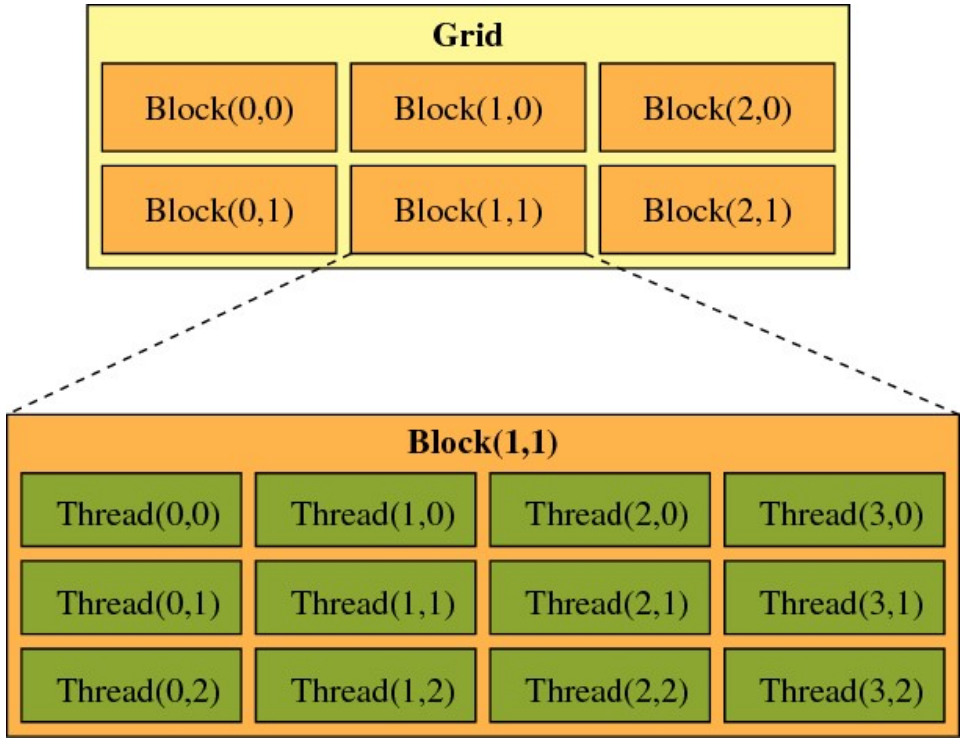
- Each thread does exactly two reads from memory, one multiply and one store operation, and then terminates.
- The code executed by each thread is identical but the data changes.
- This is at the heart of the CUDA and SPMD model.

Threading on GPUs

- Threads are grouped into 32 thread groups and these groups of threads is a **warp** (32 threads) and a **half warp** (16 threads).
- The 128 threads translate into four groups of 32 threads.
- The first set all run together to extract the thread ID and then calculate the address in the arrays and issue a memory fetch request

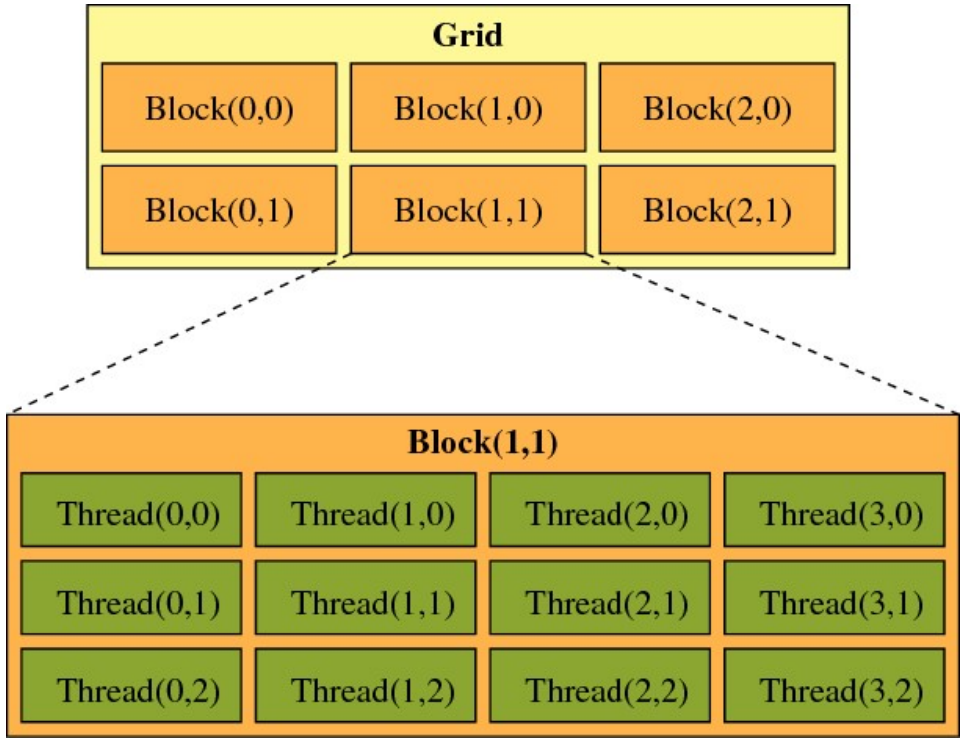
Thread Hierarchy

- A kernel executes in parallel across a set of parallel threads
- All threads that are generated by a kernel launch are collectively called a grid
- Threads are organized in thread blocks, and blocks are organized in to grids



Thread Hierarchy

- A thread block is a set of concurrently executing threads that can cooperate among themselves through barrier synchronization and shared memory
- A grid is an array of thread blocks that execute the same kernel
  - Read inputs to and write results to
    global memory
  - Synchronize between dependent kernel calls



Dimension and Index Variables

Dimension

- `gridDim` specifies the number of blocks in the grid
- `blockDim` specifies the number of threads in each block

Index

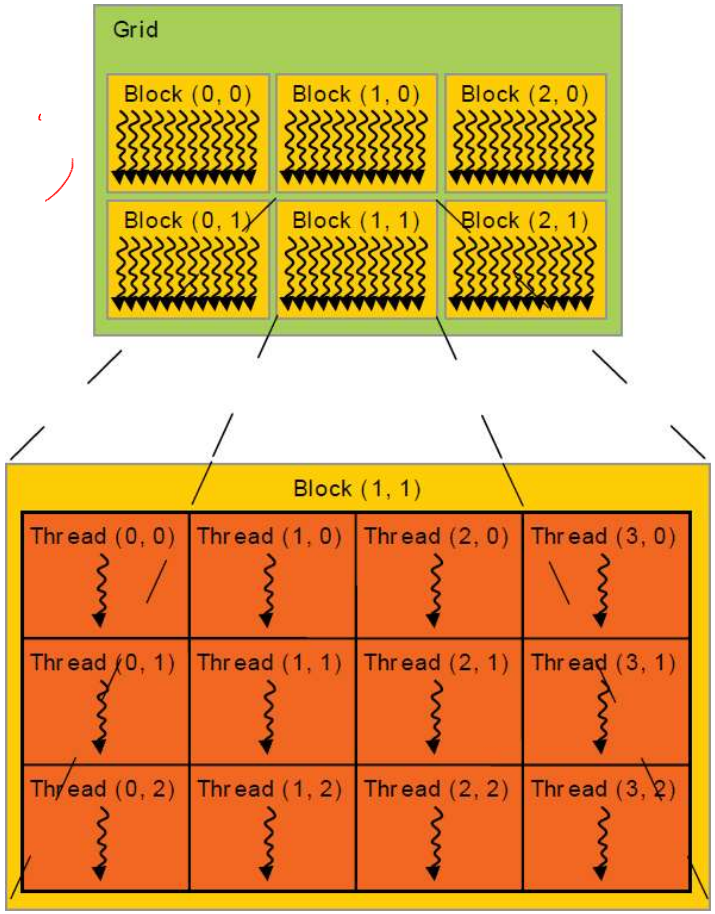
- `blockIdx` gives the index of the block in the grid
- `threadIdx` gives the index of the thread within the block

Thread Hierarchy

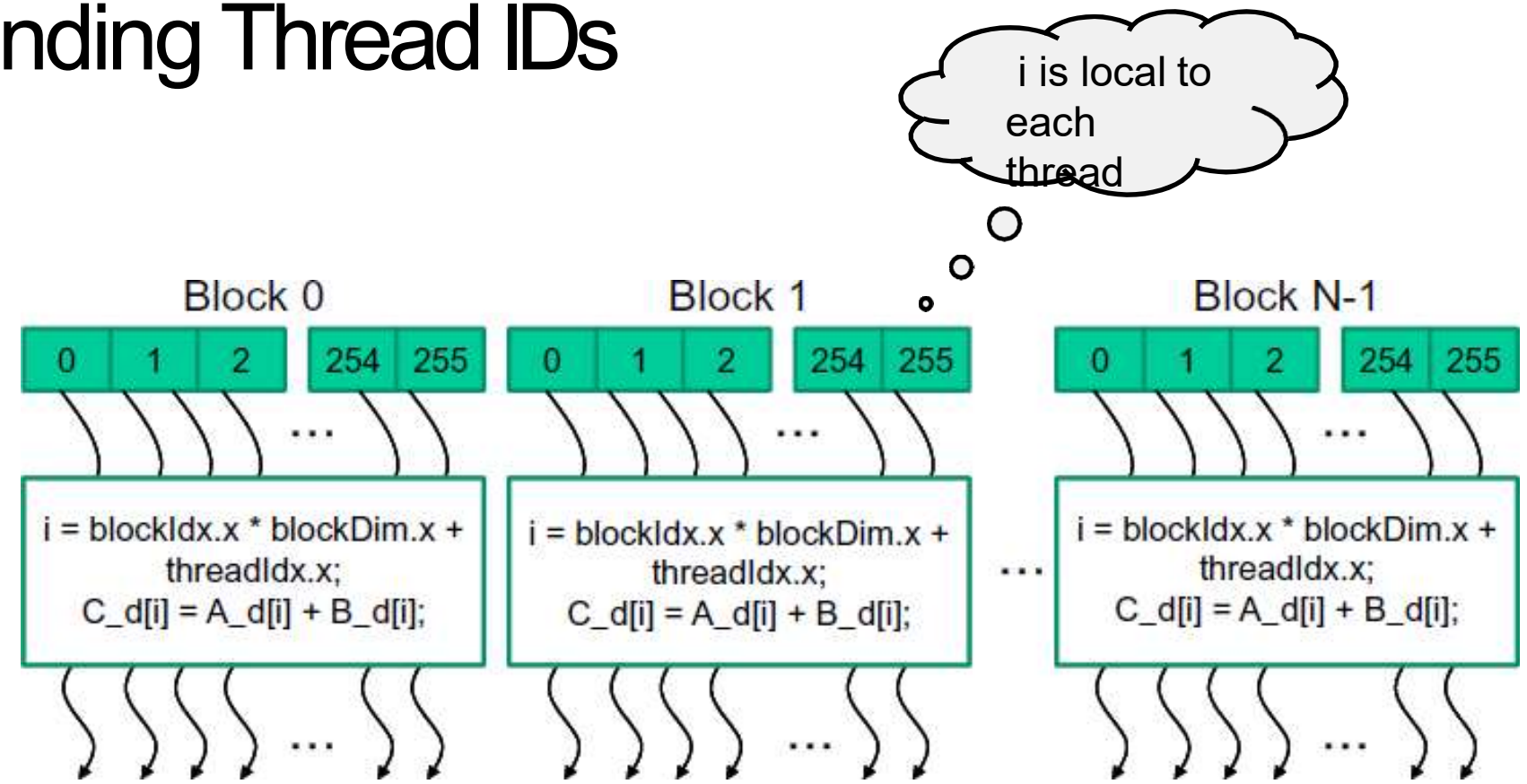
- `threadIdx` is a 3-component vector
  - Thread index can be 1D, 2D, or 3D
  - Thread blocks as a result can be 1D, 2D, or 3D
- How to find out the relation between thread ids and `threadIdx`?
  - 1D: `tid = threadIdx.x`
  - 2D block of size `(Dx, Dy)`: thread ID of a thread of index `(x, y)` is `(x + yDx)`
  - 3D block of size `(Dx, Dy, Dz)`: thread ID of a thread of index `(x, y, z)` is `(x + yDx + zDxDy)`

Thread Hierarchy

- Threads in a block reside on the same core, max 1024 threads in a block
- Thread blocks are organized into 1D, 2D, or 3D grids
  - Also called cooperative thread array
  - Grid dimension is given by `gridDim` variable
- Identify block within a grid with the `blockIdx` variable
  - Block dimension is given by `blockDim` variable



Finding Thread IDs



Threading on GPUs

- Fetches from consecutive threads are grouped together.
- This reduces the overall latency
- As a result of the grouping, the memory fetch returns with the data for a whole group of threads
- This will enable an entire warp.
- These threads are placed in the ready state and become available for the GPU
- Having executed all the warps (groups of 32 threads) the GPU becomes idle waiting for any one of the pending memory accesses to complete.

Threading on GPUs

- CUDA kernels:

- A kernel is just a name for a function that executes on the GPU. To invoke a kernel you use the following syntax:
- kernel_function<<<num_blocks, num_threads>>>(param1, param2, .)
- The parameters : **num_blocks** and **num_threads**. These can be either variables or literal values.
- The **num_block*s*** parameter is no of blocks you launch or ensure you have at least one block of threads.
- The **num_threads** parameter is simply the number of threads you wish to launch into the kernel.

Mapping Blocks and Threads

- A GPU executes one or more kernel grids
- When a CUDA kernel is launched, the thread blocks are enumerated and distributed to SMs
  - Potentially >1 block per SM
- An SM executes one or more thread blocks
  - Each GPU has a limit on the number of blocks that can be assigned to each SM
  - For example, a CUDA device may allow up to eight blocks to be assigned to each SM
  - Multiple thread blocks can execute concurrently on one SM

Mapping Blocks and Threads

- The threads of a block execute concurrently on one SM
  - CUDA cores in the SM execute threads
- A block begins execution only when it has secured all execution resources necessary for all the threads
- As thread blocks terminate, new blocks are launched on the vacated multiprocessors
- Blocks are mostly not supposed to synchronize with each other
  - Allows for simple hardware support for data parallelism

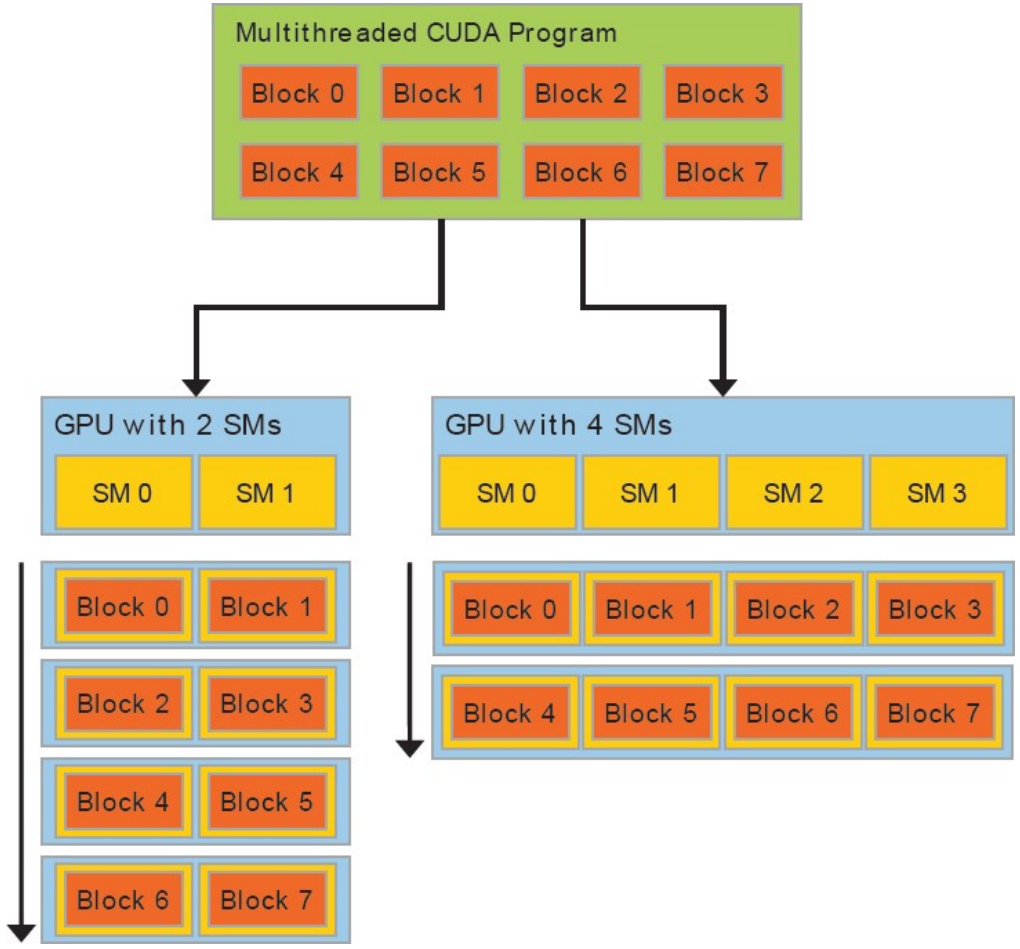
Scheduling Blocks

- Number of threads that can be simultaneously tracked and scheduled is bounded
  - Requires resources for an SM to maintain block and thread indices and their execution status
- Up to 2048 threads can be assigned to each SM on recent CUDA devices
  - For example, 8 blocks of 256 threads, or 4 blocks of 512 threads
- Assume a CUDA device with 28 SMs
  - Each SM can accommodate up to 2048 threads
  - The device can have up to 57344 threads simultaneously residing in the device for execution

Scalability of GPU Architecture

A multithreaded program is partitioned into blocks of threads that execute independently from each other.

A GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors.



Thread Warps

- Conceptually, threads in a block can execute in any order
- Sharing a control unit among compute units reduce hardware complexity, cost, and power consumption
- A set of consecutive threads (currently 32) that execute in SIMD fashion is called a **warp**
  - These are called **wavefront** (with 64 threads) on **AMD**
- **Warps** are scheduling units in an **SM**
  - Part of the implementation in NVIDIA, not the programming model

Thread Warps

- All threads in a warp run in lockstep
  - Warps share an instruction stream
  - Same instruction is fetched for all threads in a warp during the instruction fetch cycle
    - Prior to Volta, warps used a single shared program counter
  - In the execution phase, each thread will either execute the instruction or will execute nothing
  - Individual threads in a warp have their own instruction address counter and register state

Thread Warps

- Warp threads are fully synchronized
  - There is an implicit barrier after each step/instruction

- If 3 blocks are assigned to an SM and each block has 256 threads, how many warps are there in an SM?
  - Each Block is divided into 256/32 = 8 warps
  - There are 8 * 3 = 24 warps

Thread Divergence

- If some threads take the if branch and other threads take the else branch, they cannot operate in lockstep
  - Some threads must wait for the others to execute
  - Renders code at that point to be serial rather than parallel
- Divergence occurs only within a warp
- The programming model does not prevent thread divergence
  - Performance problem at the warp level

BLOCKS

- **Blocks** are groups of threads that are executed together on a single Streaming Multiprocessor (SM).
- They are organized to efficiently utilize the GPU's resources and facilitate parallel processing.
- When you launch a kernel (a function that runs on the GPU) in CUDA, you specify the number of blocks and threads per block.
- The threads within a block can communicate and synchronize with each other through shared memory, making it possible to divide a complex task into smaller units and process them in parallel

BLOCKS

kernel_function<<<num_blocks, num_threads>>>(param1, param2,...)

- If you change this from one to two, you double the number of threads you are asking the GPU to invoke on the hardware. Thus, the same call,

  some_ kernel_ func<<< 2, 128 >>>(a, b, c);

- will call the GPU function named **some_kernel**_func 2 x 128 times, each with a different thread. This, however, complicates the calculation of the ***thread_id x parameter,*** effectively the array index position.

__global__ void some_kernel_func (int * const a, const int * const b, const int * const c)

{

*const unsigned int thread_idx = (blockIdx.x * blockDim.x) + threadIdx.x;*

*a[thread_idx] = b[thread_idx] * c[thread_idx];*

}

BLOCKS

- To calculate the ***thread_idx*** parameter, we have to consider number of blocks.
- 1024 threads per block on the Fermi hardware
- 65,536 blocks would translate into around 64 million threads.
- At 1024 threads, you only get one thread block per SM.
- you need some 65,536 SMs in a single GPU

BLOCKS

- With 64 million threads, assuming one thread per array element, you can process up to 64 million elements.
- Assuming each element is a single-precision floating-point number, requiring 4 bytes of data, need around 256 million bytes, or 256 MB, of data storage space.
- Almost all GPU cards support at least this amount of memory space.
- working with threads and blocks alone you can achieve quite a large amount of parallelism.

BLOCKS

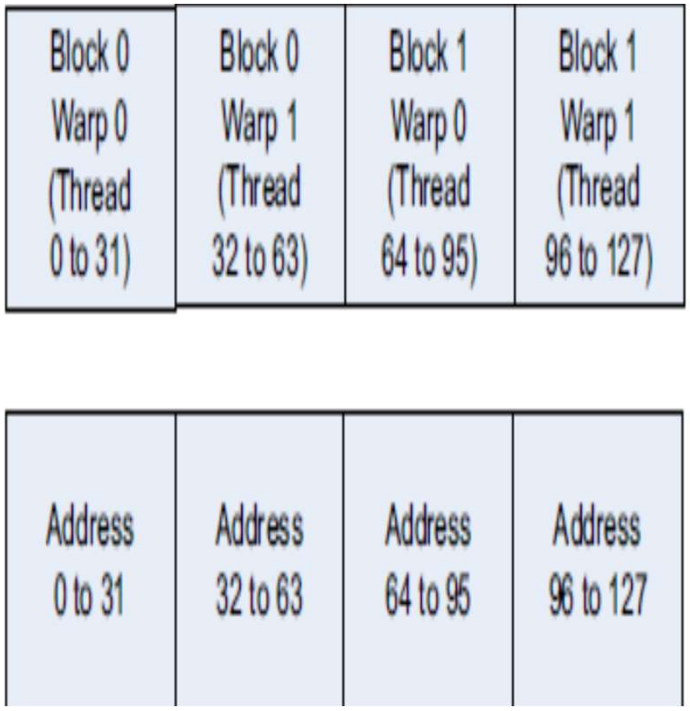


Fig : Block mapping to address

Block arrangement

- A kernel program to print the block, thread, warp, and thread index to the screen.
- Unless you have at least version 3.2 of the SDK, the printf statement is not supported in kernels.
- So we'll ship the data back to the CPU and print it to the console window.
- The kernel program is thus as follows:

Block arrangement

- __global__ void what_is_my_id(unsigned int * const block, unsigned int * const thread, unsigned int * const warp, unsigned int * const calc_thread)

```
        {
/* Thread id is block index * block size +  thread offset into the block */
        const unsigned int thread_idx =(blockIdx.x * blockDim.x) + threadIdx.x;
        block[thread_idx] =blockIdx.x;
        thread[thread_idx] = threadIdx.x;

 /* Calculate warp using built in variable warpSize */
        warp[thread_idx] = threadIdx.x / warpSize;
        calc_thread[thread_idx] = thread_idx;
        }
```

Block arrangement

- On the CPU you have to run a section of code, to allocate memory for the arrays on the GPU and then transfer the arrays back from the GPU and display them on the CPU.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
__global__ void what_is_my_id(unsigned int * const block,
unsigned int * const thread,
unsigned int * const warp,
unsigned int * const calc_thread)
{
```

Block arrangement

```
/* Thread id is block index * block size +thread offset into the block */
        const unsigned int thread_idx = (blockIdx.x * blockDim.x) + threadIdx.x;
        block[thread_idx] = blockIdx.x;
        thread[thread_idx] = threadIdx.x;

/* Calculate warp using built in variable warpSize */
        warp[thread_idx] = threadIdx.x / warpSize;
        calc_thread[thread_idx] = thread_idx;
        }
```

Block arrangement

```
        #define ARRAY_SIZE 128
        #define ARRAY_SIZE_IN_BYTES (sizeof(unsigned int) * (ARRAY_SIZE))
/* Declare statically four arrays of ARRAY_SIZE each */
        unsigned int cpu_block[ARRAY_SIZE];
        unsigned int cpu_thread[ARRAY_SIZE];
        unsigned int cpu_warp[ARRAY_SIZE];
        unsigned int cpu_calc_thread[ARRAY_SIZE];
        int main(void)
        {
/* Total thread count = 2 * 64 = 128 */
        const unsigned int num_blocks = 2;
        const unsigned int num_threads = 64;
        char ch;
```

Block arrangement

```
/* Declare pointers for GPU based parameters */
        unsigned int * gpu_block;
        unsigned int * gpu_thread;
        unsigned int * gpu_warp;
        unsigned int * gpu_calc_thread;
/* Declare loop counter for use later */
        unsigned int i;
/* Allocate four arrays on the GPU */
        cudaMalloc((void **)&gpu_block, ARRAY_SIZE_IN_BYTES);
        cudaMalloc((void **)&gpu_thread, ARRAY_SIZE_IN_BYTES);
        cudaMalloc((void **)&gpu_warp, ARRAY_SIZE_IN_BYTES);
        cudaMalloc((void **)&gpu_calc_thread, ARRAY_SIZE_IN_BYTES);
```

Block arrangement

```
/* Execute our kernel */
        what_is_my_id<<<num_blocks, num_threads>>>(gpu_block, gpu_thread,
gpu_warp,gpu_calc_thread);


/* Copy back the gpu results to the CPU  /
        cudaMemcpy(cpu_block, gpu_block, ARRAY_SIZE_IN_BYTES,
        cudaMemcpyDeviceToHost);
        cudaMemcpy(cpu_thread, gpu_thread, ARRAY_SIZE_IN_BYTES,
        cudaMemcpyDeviceToHost);
        cudaMemcpy(cpu_warp, gpu_warp, ARRAY_SIZE_IN_BYTES,
        cudaMemcpyDeviceToHost);
        cudaMemcpy(cpu_calc_thread, gpu_calc_thread, ARRAY_SIZE_IN_BYTES,
        cudaMemcpyDeviceToHost);
```

Block arrangement

```
/* Free the arrays on the GPU as now we're done with them */
        cudaFree(gpu_block);
        cudaFree(gpu_thread);
        cudaFree(gpu_warp);
        cudaFree(gpu_calc_thread);

/* Iterate through the arrays and print */
        for (i=0; i < ARRAY_SIZE; i++)
        {
        printf("Calculated Thread: %3u - Block: %2u - Warp %2u - Thread %3u\n",
        cpu_calc_thread[i], cpu_block[i], cpu_warp[i], cpu_thread[i]);
        }
        ch =getch();
        }
```

Block arrangement

- The output of the previous program is as follows:

Calculated Thread: 0 - Block: 0 - Warp 0 - Thread 0
Calculated Thread: 1 - Block: 0 - Warp 0 - Thread 1
Calculated Thread: 2 - Block: 0 - Warp 0 - Thread 2
Calculated Thread: 3 - Block: 0 - Warp 0 - Thread 3
Calculated Thread: 4 - Block: 0 - Warp 0 - Thread 4

Calculated Thread: 30 - Block: 0 - Warp 0 - Thread 30
Calculated Thread: 31 - Block: 0 - Warp 0 - Thread 31
Calculated Thread: 32 - Block: 0 - Warp 1 - Thread 32
Calculated Thread: 33 - Block: 0 - Warp 1 - Thread 33
Calculated Thread: 34 - Block: 0 - Warp 1 - Thread 34

Calculated Thread: 62 - Block: 0 - Warp 1 - Thread 62
Calculated Thread: 63 - Block: 0 - Warp 1 - Thread 63
Calculated Thread: 64 - Block: 1 - Warp 0 - Thread 0
Calculated Thread: 65 - Block: 1 - Warp 0 - Thread 1
Calculated Thread: 66 - Block: 1 - Warp 0 - Thread 2
Calculated Thread: 67 - Block: 1 - Warp 0 - Thread 3

Grids

- A **grid** is simply a set of blocks arranged in X & a Y axis, in a 2D mapping.
- The final Y mapping gives you Y * X * T possibilities for a thread index.
- The no. of threads in a block be a multiple of the **warp size**, which is currently defined as 32.
- you can schedule a full warp on the hardware, if you don’t do this, then the remaining part of the warp goes unused.
- you have to introduce a condition to ensure you don’t process elements off the end of the X axis.

Grids

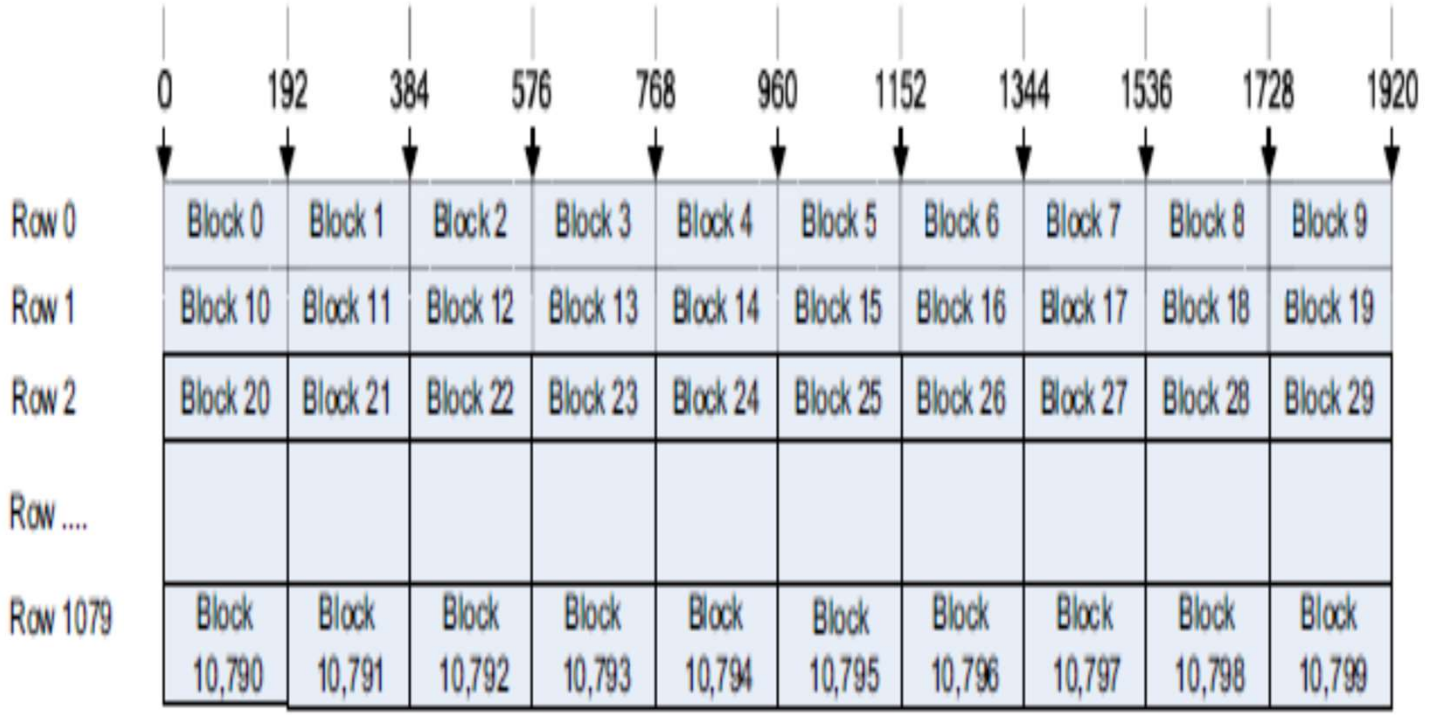


Fig: Block allocation to Rows

Grids

- A thread size that is a multiple of the X axis and the warp size makes it easier.
- Along the top on the X-axis, you have the thread index.
- The row index forms the Y-axis.
- The height of the row is exactly one pixel.
- You have 1080 rows of 10 blocks, you have 1080x10=10,800 blocks.

Grids

Stride and offset

- The **width of the array** is referred to as the **stride** of the memory access.
- The **offset** is the column value being accessed, starting at the left, which is always element 0.
- CUDA is designed to allow for data decomposition into parallel threads and blocks.
- It allows you to define 1D, 2D, or 3D indexes (Y x X xT) when referring to the parallel structure of the program.

Grids

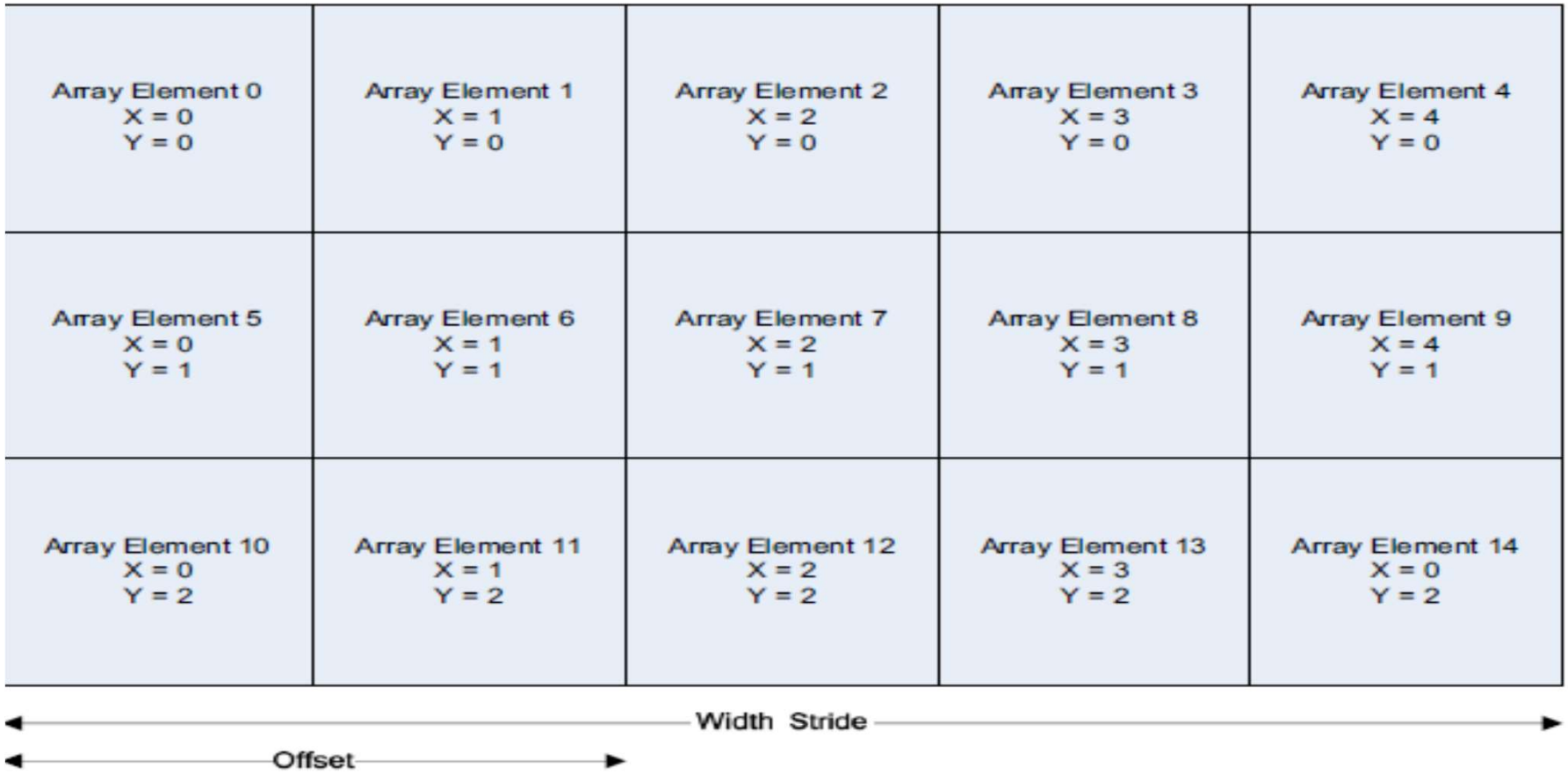


Fig : Array mapping to elements

Grids

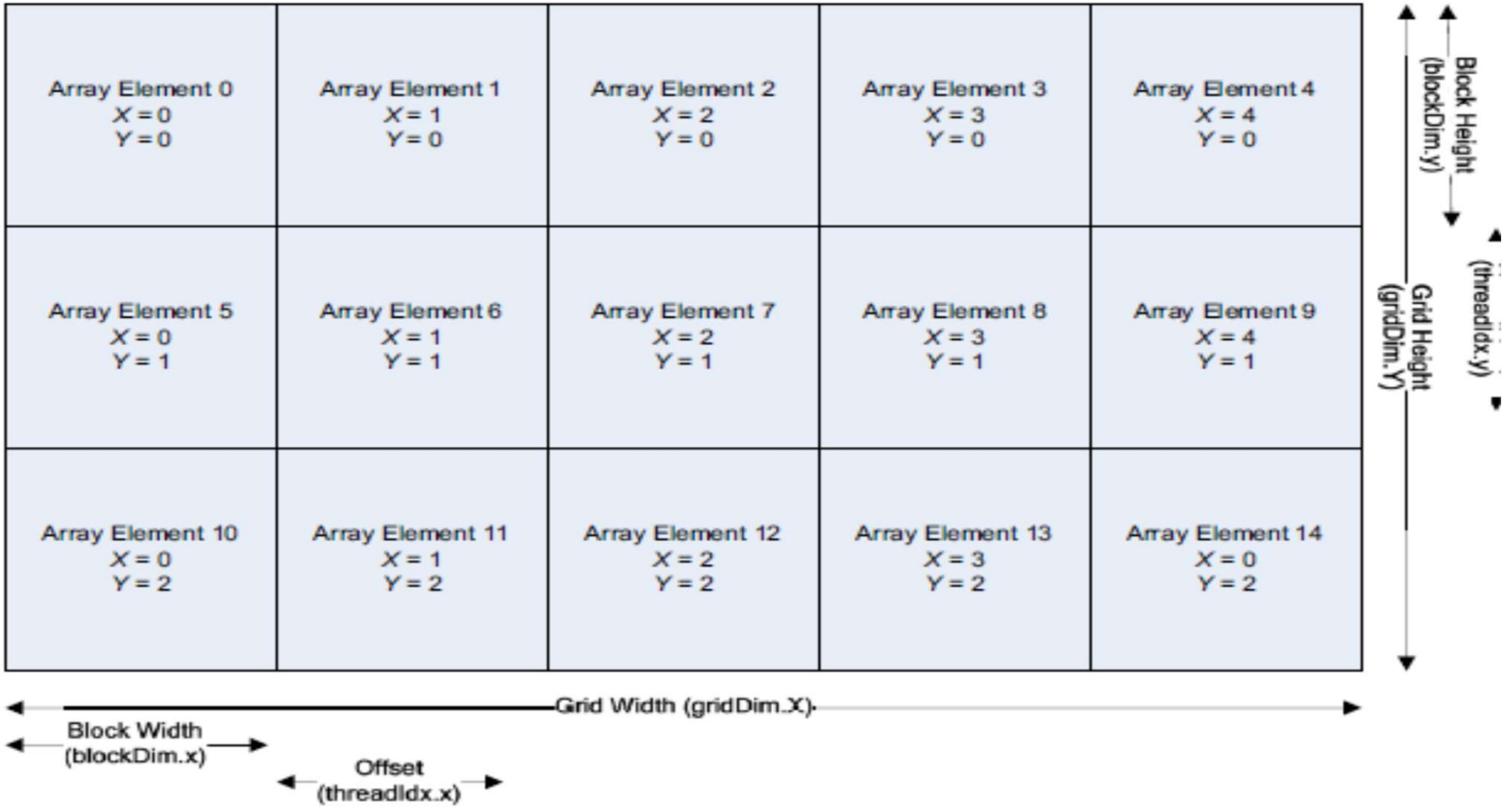
X and Y thread indexes

- A 2D array in terms of blocks means you get two thread indexes.
- you will be accessing the data in a 2D way:

```
const unsigned int idx = (blockIdx.x * blockDim.x) + threadIdx.x;
const unsigned int idy = (blockIdx.y * blockDim.y) + threadIdx.y;
some_array[idy][idx] += 1.0;
```

- blockDim.x and blockDim.y, specifies the dimension on the X and Y axis.
- You can schedule them as stripes across the array, or as squares within the array fig below

Grids



- Fig : Grid, block, and thread dimensions.

Grids

- You can see a number of new parameters, which are:

  gridDim.x–The size in blocks of the X dimension of the grid.

  gridDim.y–The size in blocks of the Y dimension of the grid.

  blockDim.x–The size in threads of the X dimension of a single block.

  blockDim.y–The size in threads of the Y dimension of a single block.

  threadIdx.x–The offset within a block of the X thread index.

  threadIdx.y–The offset within a block of the Y thread index.

WARPS

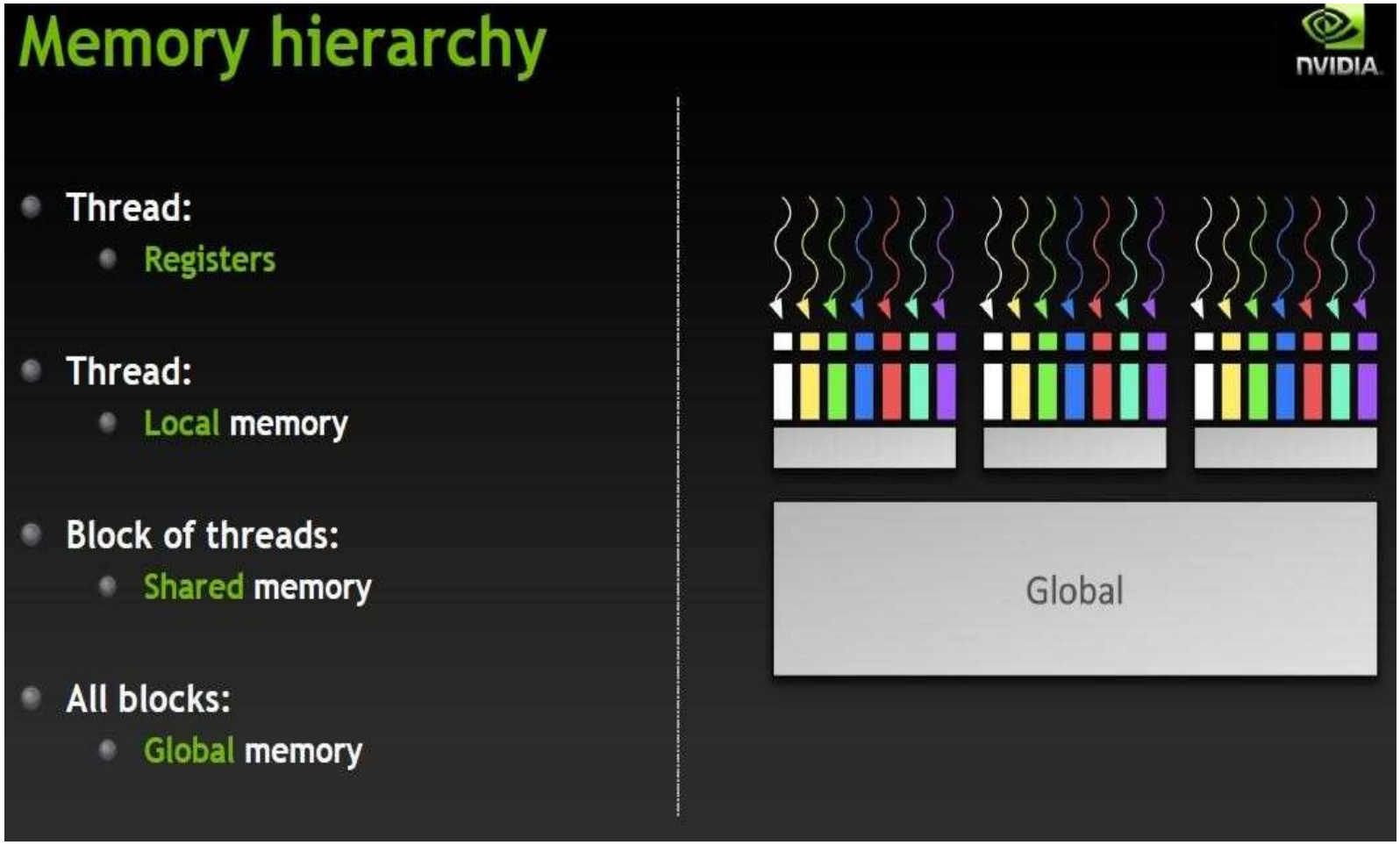
- Warps are the basic unit of execution on the GPU.
- The GPU is effectively a collection of SIMD vector processors.
- Each group of threads, or warps, is executed together.
- In the ideal case, only one fetch from memory for the current instruction and a broadcast of that instruction to the entire set of SPs in the warp.
- This is much more efficient than the CPU model, which fetches independent execution streams to support task-level parallelism.
- In the CPU model, for every core you are running an independent task, you can conceptually divide the memory.

WARPS

- With GPU programming, It's vector architecture and expects you to write code that runs on thousands of threads.
- This approach allows you to check things, such as whether memory copying to/from the GPU is working correctly, before introducing parallelism into the application.
- Warps on the GPU are currently 32 elements, although nVidia reserves the right to change this in the future.

Memory Handling with C U D A

NVIDIA GPU Memory Structures



NVIDIA GPUs: Terminology

- Memory hardware
- Global Memory
  - DRAM available to all threads (SIMD processors in GPU)
- Local Memory
  - Private to the thread
- Shared Memory
  - Accessible to all threads of a Streaming Processor
- Thread Processor Registers

CACHES

- A cache is a high-speed memory that is physically close to the processor core.
- Caches are expensive.
- The maximum speed of a cache is proportional to the size of the cache.
- L1 cache is the fastest, but is limited in size to usually around 16 K, 32 K, or 64 K. It is usually allocated to a single CPU core.
- The L2 cache is slower, but much larger, typically 256 K to 512 K.
- The L3 cache may or may not be present and is often several megabytes in size.
- The L2 and/or L3 cache may be shared between processor cores or maintained as separate caches linked directly to given processor cores.
- Generally, at least the L3 cache is a shared cache between processor cores on a conventional CPU.
- This allows for fast intercore communication via this shared memory within the device

CACHES

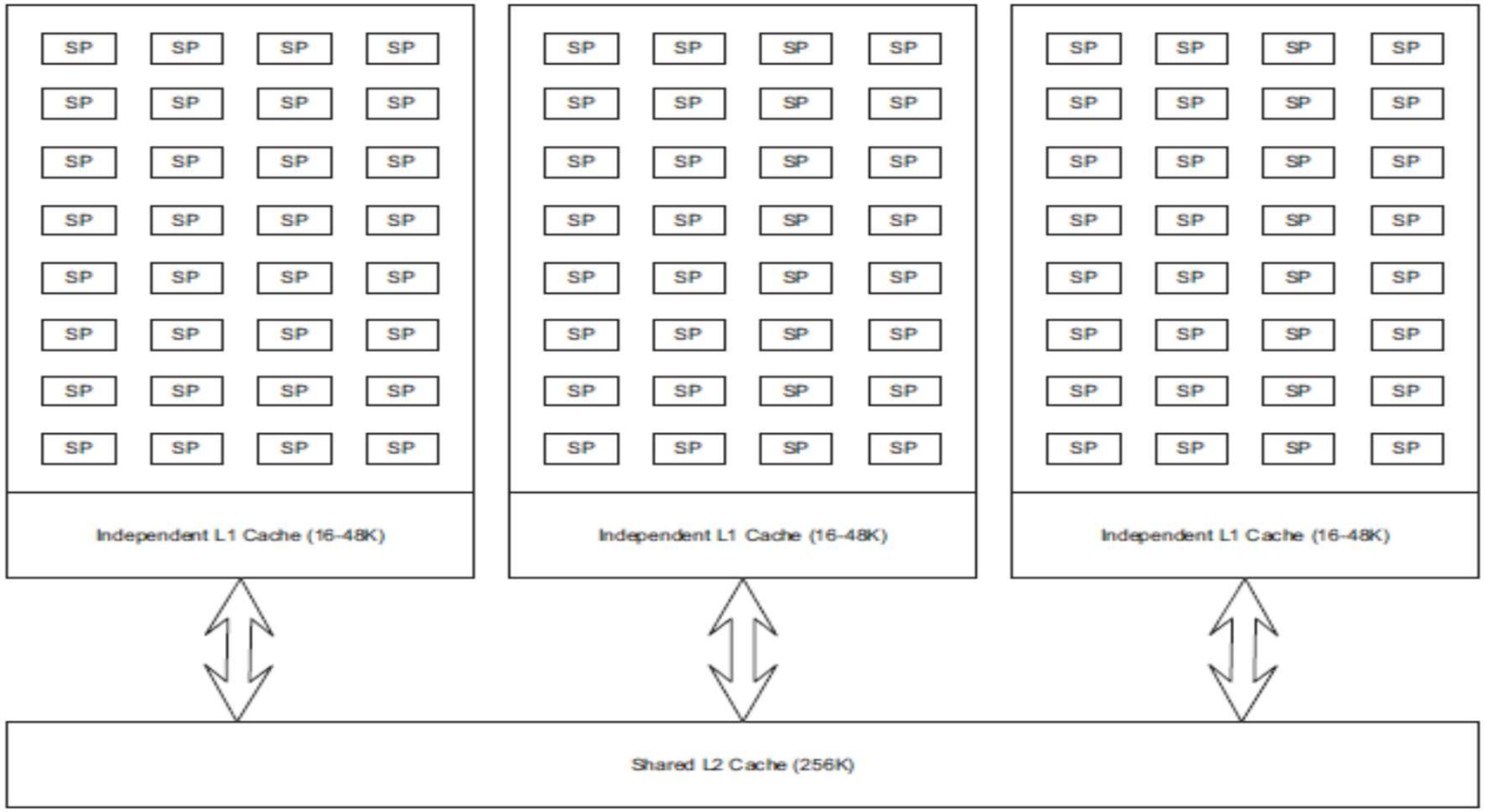


Fig: SM L1/L2 data path.

CACHES

- In Fermi GPU we introduce the concept of a nonprogrammer managed data cache.
- The architecture additionally has, per SM, an L1 cache that is both programmer managed and hardware managed.
- It also has a shared L2 cache across all SMs

CACHES

- **Types of data storage**
- On a GPU, we have a number of levels of areas where you can place data, each defined by its potential bandwidth and latency, as shown in Table below.

CACHES

Table 6.1 Access Time by Memory Type

Storage Type	Registers	Shared Memory	Texture Memory	Constant Memory	Global Memory
Bandwidth	~8 TB/s	~1.5 TB/s	~200 MB/s	~200 MB/s	~200 MB/s
Latency	1 cycle	1 to 32 cycles	~400 to 600	~400 to 600	~400 to 600

REGISTER USAGE

- The GPU, has thousands of registers per SM (streaming multiprocessor).
- An SM can be thought of like a multithreaded CPU core.
- On a typical CPU we have two, four, six, or eight cores.
- On a GPU we have N SM cores.
- On a Fermi GF100 series, there are 16 SMs on the top-end device. The GT200 series has up to 32 SMs per device
- A typical CPU will support one or two hardware threads per core.
- A GPU by contrast has between 8 and 192 SPs per core, meaning each SM can at any time be executing this number of concurrent hardware threads.

REGISTER USAGE

- On GPUs, application threads are pipelined, context switched, and dispatched to multiple SMs.
- The number of active threads across all SMs in a GPU device is usually in the tens of thousands range.

REGISTER USAGE

- One major difference we see between CPU and GPU architectures is how CPUs and GPUs map registers:
  - The CPU runs lots of threads by using register renaming and the stack.
  - To run a new task the CPU needs to do a context switch, which involves storing the state of all registers onto the stack (the system memory) and then restoring the state from the last run of the new thread.
  - This can take several hundred CPU cycles.
  - If you load too many threads onto a CPU it will spend all of the time simply swapping out and in registers as it context switches.
  - The effective throughput of useful work rapidly drops off as soon as you load too many threads onto a CPU.

REGISTER USAGE

- The GPU by contrast is the exact opposite.
  - It uses threads to hide memory fetch and instruction execution latency
  - The GPU does not use register renaming, but instead dedicates real registers to each and every thread.
  - Thus, when a context switch is required, it has near zero overhead.
  - On a context switch the selector (or pointer) to the current register set is updated to point to the register set of the next warp that will execute.

REGISTER USAGE

- We use registers to avoid usage of the slower memory types.
- For example, suppose we had a loop that set each bit in turn, depending on the value of some Boolean variable.

```
for (i=0; i<31; i++)
{
packed_result | = (pack_array[i] << i);
}
```

- Here we are reading array element ***i*** from an array of elements to pack into an integer, packed_ result.
- We're left shifting the Boolean by the necessary number of bits and then using a ***bitwise or*** operation with the previous result.

SSN

REGISTER USAGE

- If the parameter ***packed _result*** exists in memory, you have 32 memory read and writes.
- We can place the parameter ***packed_result*** in a local variable
- The compiler would place into a register.
- As we accumulate into the register instead of in main memory, and later write only the result to main memory, we save 31 of the 32 memory reads and writes.

REGISTER USAGE

- Assume 500 cycles for one global memory read or write operation.
- For every value you'd need to read, apply the or operation, and write the result back.
- You have 32 X read + 32 X write = 64X500 cycles =32,000 cycles.
- The register version would eliminate 31 read and 32 write operations, replacing the 500-cycle operations with single-cycle operations.

(1X memory read) +(1 X memory write)+(31x register read) + (31X register write)

or

(1x500)+(1X500) +(31 X1)+ (31X 1)= 1062 cycles versus 32, 000 cycles.

- This is a huge reduction in the number of cycles.
- We have a 31 times improvement to perform a relatively common operation in certain problem domains

SHARED MEMORY

- Shared memory is effectively a user-controlled L1 cache.
- The L1 cache and shared memory share a 64 K memory segment per SM.
- In Kepler this can be configured in 16 K blocks in favor of the L1 or shared Memory.
- In Fermi the choice is 16K or 48K in favor of the L1 or shared memory.
- The shared memory has in the order of 1.5 TB/s bandwidth with extremely low latency.
- This is hugely superior to the up to 190GB/s available from global memory, but around one-fifth of the speed of registers.

SHARED MEMORY

- Shared memory is a bank-switched architecture.
- On Fermi it is 32 banks wide, and on G200 and G80 hardware it is 16 banks wide.
- Each bank of data is 4 bytes in size, enough for a single-precision floating-point data item or a standard 32-bit integer value.
- Kepler also introduces a special 64 bit wide mode .
- There is no need for a one-to-one sequential access, just that every thread accesses a separate bank in the shared memory.
- There is a crossbar switch connecting any single bank to any single thread.
- This is very useful when you need to swap the words.

SHARED MEMORY

- Every thread in a warp reads the same bank address.
- This triggers a broadcast mechanism to all threads within the warp.
- Usually thread zero writes the value to communicate a common value with the other threads in the warp (fig below).
- If we have any other pattern, we end up with bank conflicts of varying degrees.
- This means you stall the other threads in the warp that idle while the threads accessing the shared memory address queue up one after another.
- One important aspect of this is that it is not hidden by a switch to another warp, so we do in fact stall the SM.

Radix sort

- It has a fixed number of iterations and a consistent execution flow.
- It works by sorting based on the least significant bit and then working up to the most significant bit.
- With a 32-bit integer, using a single radix bit, you will have 32 iterations of the sort, no matter how large the dataset.
- example : { 122, 10, 2, 1, 2, 22, 12, 9 }
- The binary representation of each of these would be

  122 = 01111010
  10 = 00001010
  2 = 00000010
  22 = 00010010
  12 = 00001100
  9 = 00001001

Radix sort

- In the first pass, all elements with a 0 in the LSB would form the first list.
- Those with a 1 as the LSB would form the second list. Thus, the two lists are

  0 = { 122, 10, 2, 22, 12 } & 1 = { 9 }
- The two lists are appended in this order, becoming

  { 122, 10, 2, 22, 12, 9 }
- The process is then repeated for bit one, generating the next two lists based on the ordering of the previous cycle:

  0 = { 12, 9 } & 1 = { 122, 10, 2, 22 }
- The combined list is then

  { 12, 9, 122, 10, 2, 22 }
- Scanning the list by bit two, we generate

  0 = { 9, 122, 10, 2, 22 } & 1 = { 12 }

  = { 9, 122, 10, 2, 22, 12 }

Radix sort

- The program continues until it has processed all 32 bits of the list in 32 passes.
- To build the list you need **N + 2N** memory cells.
- one for the source data, one of the **0 list**, and one of the **1 list**.

Radix sort: Serial code

```
__host__ void cpu_sort(u32 * const data, const u32 num_elements)
  {
      static u32 cpu_tmp_0[NUM_ELEM];
      static u32 cpu_tmp_1[NUM_ELEM];
      for (u32 bit=0;bit<32;bit++)
      {
      u32 base_cnt_0 = 0;
      u32 base_cnt_1 = 0;
```

Radix sort: Serial code

```
for (u32 i=0; i<num_elements; i++)
        {
        const u32 d= data[i];
        const u32 bit_mask = (1 << bit);
        if ( (d & bit_mask) > 0 )
        {
        cpu_tmp_1[base_cnt_1] = d;
        base_cnt_1++;
        }
```

Radix sort: Serial code

```
        else
        {
        cpu_tmp_0[base_cnt_0] = d;
        base_cnt_0++;
        }
        }
// Copy data back to source - first the zero list
        for (u32 i=0; i<base_cnt_0; i++)
        {
        data[i]= cpu_tmp_0[i];
        }
```

Radix sort: Serial code

```
// Copy data back to source - then the one list
        for (u32 i=0; i<base_cnt_1; i++)
        {
        data[base_cnt_0+i] = cpu_tmp_1[i];
        }
}
}
```

Radix sort: Serial code

- The code works by being passed two values, a pointer to the data to sort and the number of elements in the dataset.
- It overwrites the unsorted data so the returned set is sorted.
- The outer loop iterates over all 32 bits in a 32-bit integer word and the inner loop iterates over all elements in the list.
- The algorithm requires 32N iterations in which the entire dataset will be read and written 32 times.
- Within the inner loop the data is split into two lists,
  - the 0 list and the 1 list depending on which bit of the word is being processed.
  - The data is then reconstructed from the two lists, the 0 list always being written before the 1 list

Radix sort: Serial code

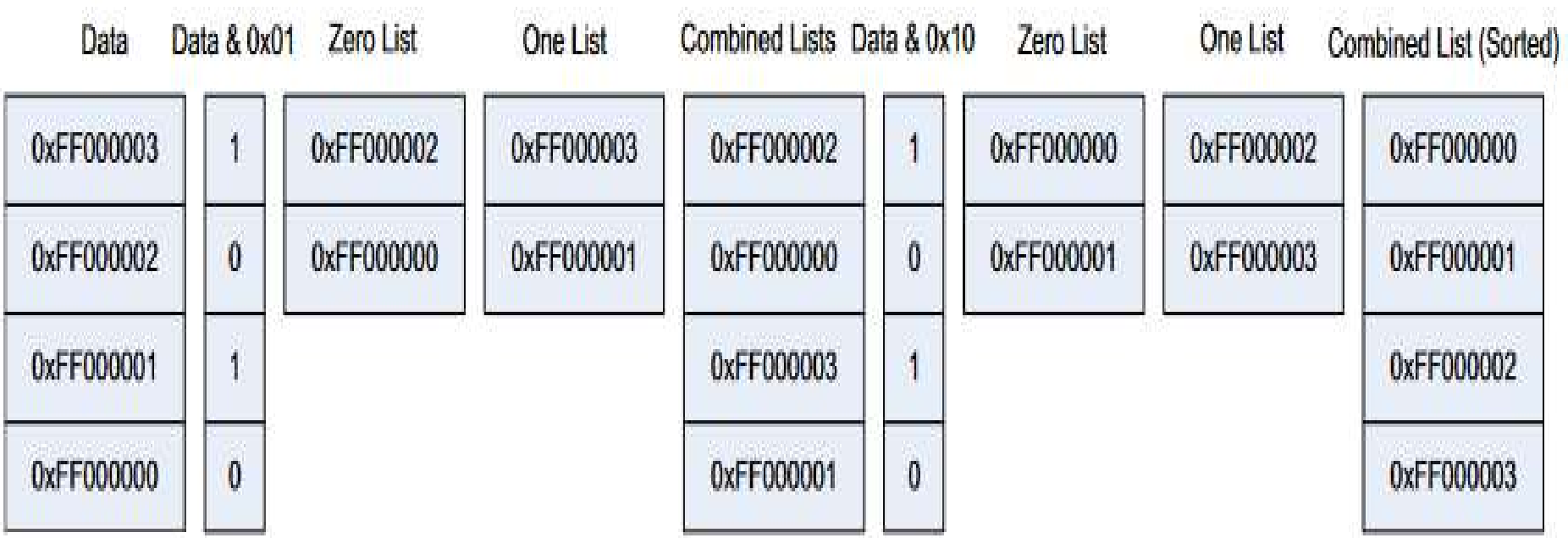


Fig: Radix Sort

Radix Sort: GPU Code

- The GPU version is a little more complex, in that we need to take care of multiple threads.

```
__device__ void radix_sort (u32 * const sort_tmp, const u32 num_lists,
const u32 num_elements, const u32 tid, u32 * const sort_tmp_0, u32 * const sort_tmp_1)
{
// Sort into num_list, lists
// Apply radix sort on 32 bits of data
        for (u32 bit=0;bit<32;bit++)
        {
        u32 base_cnt_0 = 0;
        u32 base_cnt_1 = 0;
        for (u32 i=0; i<num_elements; i+=num_lists)
        {
        const u32 elem = sort_tmp[i+tid];
        const u32 bit_mask = (1 << bit);
```

Radix Sort: GPU Code

```
        if ( (elem & bit_mask) > 0 )
        {
        sort_tmp_1[base_cnt_1+tid] = elem;
        base_cnt_1+=num_lists;
        }
        else
        {
        sort_tmp_0[base_cnt_0+tid] = elem;
        base_cnt_0+=num_lists;
        }
        }
// Copy data back to source - first the zero list
```

Radix Sort: GPU Code

```
        for (u32 i=0; i<base_cnt_0; i+=num_lists)
        {
        sort_tmp[i+tid] = sort_tmp_0[i+tid];
        }

// Copy data back to source - then the one list
        for (u32 i=0; i<base_cnt_1; i+=num_lists)
        {
        sort_tmp[base_cnt_0+i+tid] = sort_tmp_1[i+tid];
        }
        }
        __syncthreads();
}
```

Radix Sort: GPU Code

- The GPU kernel is written here as a device function, a function only capable of being called within GPU Kernel.
- The inner loop has changed and instead of incrementing by one, the program increments by num_lists a value passed into the function.
- This radix sort will produce is num_lists of independent sorted lists using num_lists threads.
- Since the SM in the GPU can run 32 threads at the same speed as just one thread and it has 32 shared memory banks, you might imagine the ideal value for num_lists would be 32.

Radix Sort: GPU Code

Table 6.6 Parallel Radix Sort Results (ms)

Device/Threads	1	2	4	8	16	32	64	128	256
GTX470	39.4	20.8	10.9	5.74	2.91	1.55	0.83	0.48	0.3
9800GT	67	35.5	18.6	9.53	4.88	2.66	1.44	0.82	0.56
GTX260	82.4	43.5	22.7	11.7	5.99	3.24	1.77	1.02	0.66
GTX460	31.9	16.9	8.83	4.56	2.38	1.27	0.69	0.4	0.26

Fig: Parallel Radix sort results

SSN

Radix Sort: Optimized GPU Code

- Do not need separate 0 and 1 lists.
- The 0 list can be created from reusing the space in the original list.
- This not only allows you to discard the 1 list, but also removes a copy back to the source list.
- This saves a lot of unnecessary work.

Radix Sort: Optimized GPU Code

- __device__ void radix_sort2 (u32 * const sort_tmp, const u32 num_lists,
  const u32 num_elements, const u32 tid, u32 * const sort_tmp_1)
  {

```
// Sort into num_list, lists
// Apply radix sort on 32 bits of data
    for (u32 bit=0;bit<32;bit++)
    {
    const u32 bit_mask= (1 << bit);
    u32 base_cnt_0 = 0;
    u32 base_cnt_1 = 0;
    for (u32 i=0; i<num_elements; i+=num_lists)
    {
    const u32 elem = sort_tmp[i+tid];
    if ( (elem & bit_mask) > 0 )
```

Radix Sort: GPU Code

```
{
sort_tmp_1[base_cnt_1+tid] = elem;
base_cnt_1+=num_lists;
}
else
{
Sort_tmp[base_cnt_0+tid] =elem;
base_cnt_0+=num_lists;
}
}
```

Radix Sort: Optimized GPU Code

```
// Copy data back to source from the one's list
        for (u32 i=0; i<base_cnt_1; i+=num_lists)
        {
        sort_tmp[base_cnt_0+i+tid] = sort_tmp_1[i+tid];
        }
        }
        __syncthreads();
}
```

Radix Sort: Optimized GPU Code

Table 6.7 Optimized Radix Sort Results (ms)

Device/Threads	1	2	4	8	16	32	64	128	256
GTX470	26.51	14.35	7.65	3.96	2.05	1.09	0.61	0.36	0.24
9800GT	42.8	23.22	12.37	6.41	3.3	1.78	0.98	0.63	0.4
GTX260	52.54	28.46	15.14	7.81	4.01	2.17	1.2	0.7	0.46
GTX460	21.62	11.81	6.34	3.24	1.69	0.91	0.51	0.31	0.21

Fig:Optimized Radix Sort Results

Merging Lists

- Merging lists using CUDA involves leveraging the parallel processing capabilities of NVIDIA GPUs to efficiently combine two or more lists into a single sorted list.
- CUDA is a parallel computing platform and programming model that allows you to use the power of GPUs for general-purpose computing tasks.

Merging Lists

Divide Data: Divide the input lists into smaller chunks that can be processed independently by different threads on the GPU. Each thread will handle a portion of the data.

Allocate GPU Memory: Allocate memory on the GPU to hold the input and output lists. Copy the input lists from the CPU to the GPU memory.

Sorting: Depending on the merging algorithm you choose, you might need to sort the input chunks before merging. This can be done using sorting algorithms like merge sort or parallel sorting networks.

Merging Lists

Merging: Implement a parallel merging algorithm to combine the sorted chunks. One common approach is to use a parallel merge algorithm similar to merge sort's merge step.

Copy Results: Copy the merged and sorted list back from the GPU memory to the CPU memory.

Clean Up: Release the allocated GPU memory.

Merging Lists

```
_global__ void merge_kernel(int* input1, int* input2, int* output, int size1,
int size2) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (tid < size1 + size2)
 {
        if (tid < size1) {
            output[tid] = input1[tid];
        } else {
            output[tid] = input2[tid - size1];
        }
    }
}
```

Merging Lists

```
int main() {
    // Initialize input lists on CPU
    int size1 = // size of the first list
    int size2 = // size of the second list
    int* host_input1 = // allocate and populate
    int* host_input2 = // allocate and populate

    // Allocate GPU memory
    int* device_input1;
    int* device_input2;
    int* device_output;
    cudaMalloc(&device_input1, size1 * sizeof(int));
    cudaMalloc(&device_input2, size2 * sizeof(int));
    cudaMalloc(&device_output, (size1 + size2) * sizeof(int));
```

Merging Lists

```
// Copy input data to GPU
    cudaMemcpy(device_input1, host_input1, size1 * sizeof(int),
cudaMemcpyHostToDevice);
    cudaMemcpy(device_input2, host_input2, size2 * sizeof(int),
cudaMemcpyHostToDevice);

    // Configure kernel launch parameters
    int block_size = 256;
    int grid_size = (size1 + size2 + block_size - 1) / block_size;

    // Launch the merge kernel
    merge_kernel<<<grid_size, block_size>>>(device_input1,
device_input2, device_output, size1, size2);
```

Merging Lists

```
// Copy results back to CPU
    int* host_output = new int[size1 + size2];
    cudaMemcpy(host_output, device_output, (size1 + size2) * sizeof(int),
cudaMemcpyDeviceToHost);

    // Clean up
    cudaFree(device_input1);
    cudaFree(device_input2);
    cudaFree(device_output);
    delete[] host_output;

    return 0;
}
```

Parallel reduction

- common optimization technique in parallel computing, often used to efficiently compute the sum, minimum, maximum, or other associative operations of an array of values.
- The basic idea behind parallel reduction is to divide the input array into smaller chunks, process those chunks in parallel, and then combine the results to obtain the final result.
- This is done iteratively until you have a single value that represents the reduction result.

Parallel reduction

- Divide Input Data: Divide the input array into blocks of threads. Each thread block will be responsible for reducing a portion of the input data.
- Load Data: Each thread loads its respective data into shared memory, which is faster to access compared to global memory. Shared memory is a small, low-latency memory space shared among threads within a thread block.
- Perform Parallel Reduction: Perform a parallel reduction within each thread block using an algorithm like tree-based reduction or warp-based reduction. These algorithms take advantage of the parallelism offered by the GPU's architecture to efficiently reduce the data.

Parallel reduction

- Combine Block Results: After each thread block completes its reduction, one thread in each block writes the block's reduced result to a separate global memory array.

- Perform Final Reduction: Finally, you perform another reduction step on the global memory array that contains the results from each block. This can be done using the same parallel reduction technique, recursively, until you obtain a single value.

Parallel reduction

```
__global__ void parallelSum(float* input, float* output, int N) {
    __shared__ float sharedData[THREADS_PER_BLOCK];

    int tid = threadIdx.x;
    int blockId = blockIdx.x;
    int globalId = blockDim.x * blockId + tid;
```

Parallel reduction

```
if (globalId < N) {
    sharedData[tid] = input[globalId];
  } else {
    sharedData[tid] = 0.0f;
  }
  __syncthreads();

  // Perform parallel reduction within the block
```

Parallel reduction

```
for (int stride = blockDim.x / 2; stride > 0; stride >>= 1)
 {
       if (tid < stride) {
          sharedData[tid] += sharedData[tid + stride];
       }
       __syncthreads();
   }

   // Write the block's reduced result to global memory
   if (tid == 0) {
       output[blockId] = sharedData[0];
   }
}
```

GLOBAL MEMORY

- GPU global memory is global because it's writable from both the GPU & CPU.
- It can be accessed from any device on the PCI-E bus.
- GPU cards can transfer data to and from one another, directly, without needing the CPU.
- The memory from the GPU is accessible to the CPU in one of three ways:
  - Explicitly with a blocking transfer.
  - Explicitly with a nonblocking transfer.
  - Implicitly using zero memory copy.

GLOBAL MEMORY

- The usual model of execution involves:

  the CPU transfers a block of data to the GPU,

  the GPU kernel processing it,

  and then the CPU initiating a transfer of the data back to the host memory.
- A slightly more advanced model of this is where we use streams to overlap transfers and kernels to ensure the GPU is always kept busy(fig)
- The same way that we can pipeline kernels, as is shown in the memory accesses are pipelined.

GLOBAL MEMORY

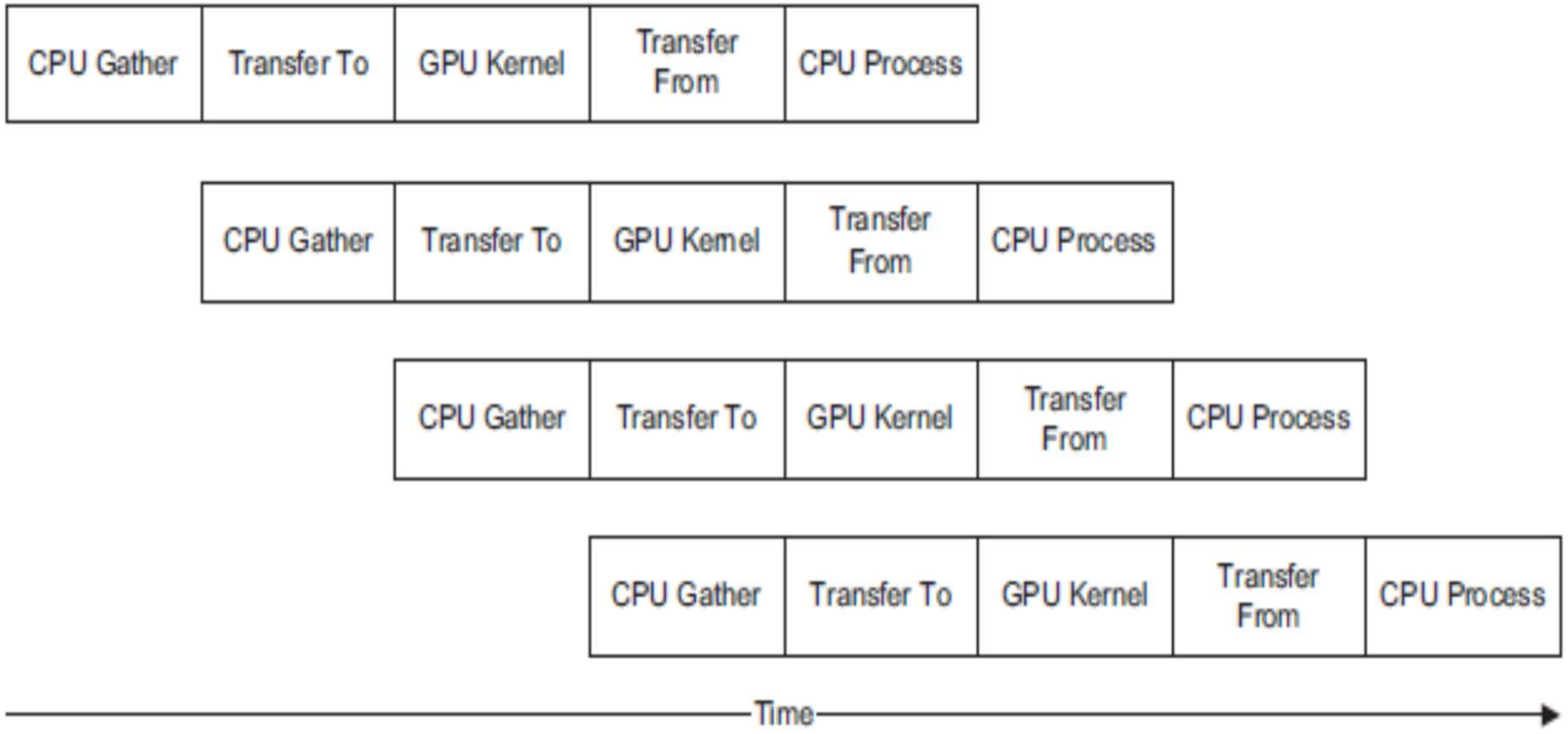


Fig : Overlapping kernel and memory transfers

GLOBAL MEMORY

- Coalescable pattern is where all the threads access a contiguous and aligned memory block.
- Here we have shown Addr as the logical address offset from the base location, assuming we are accessing byte-based data.
- TID represents the thread number
- Assuming we're accessing a single precision float or integer value, each thread will be accessing 4 bytes of memory.
- Memory is coalesced on a warp basis meaning we get 32 X 4 = 128 byte access to memory.
- Coalescing sizes supported are 32, 64, and 128 bytes, meaning warp accesses to byte, 16- and 32- bit data will always be coalesced if the access is a sequential pattern and aligned to a 32-byte boundary.

GLOBAL MEMORY

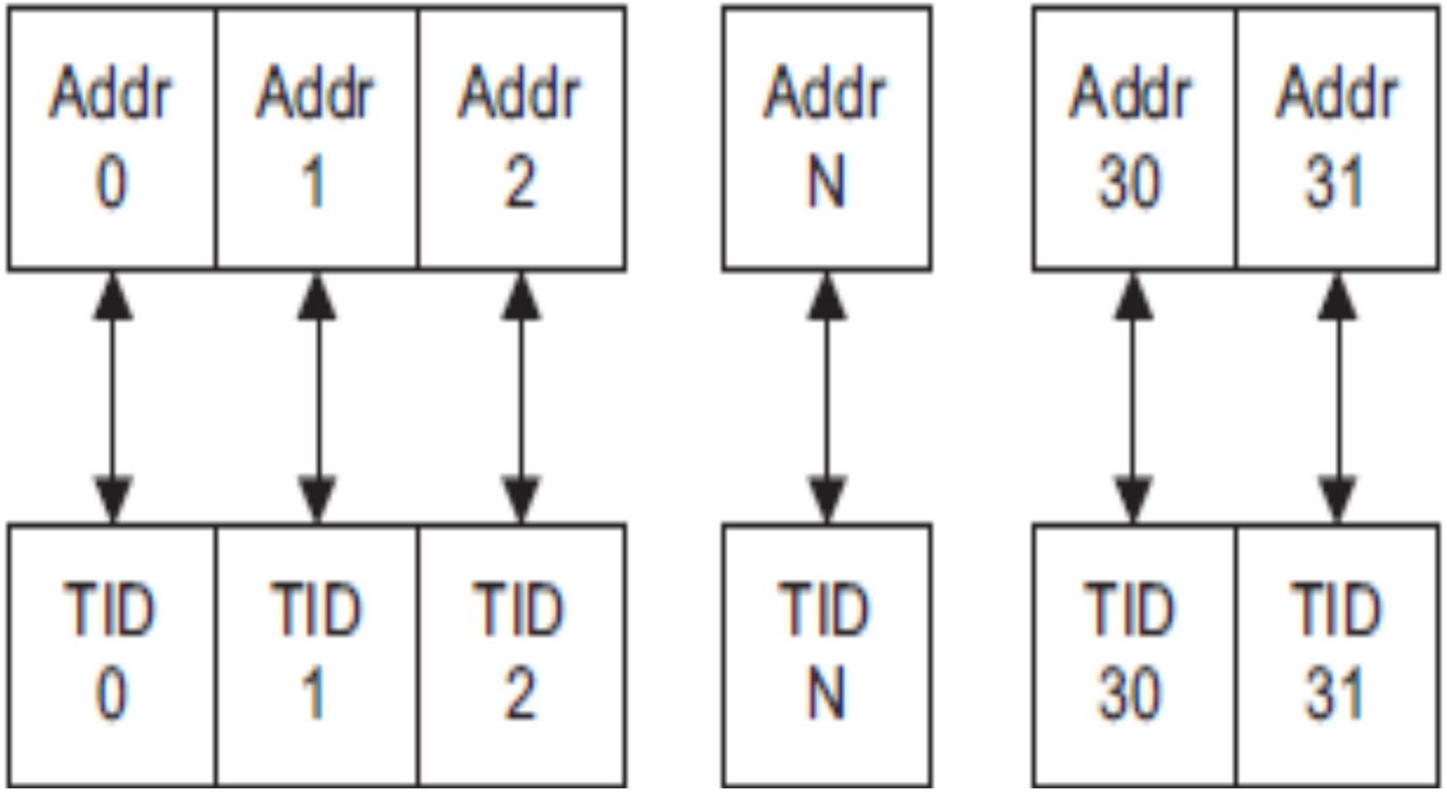


Fig: Addresses accessed by thread ID

GLOBAL MEMORY

- The alignment is achieved by using a special malloc instruction, replacing the standard ***cudaMalloc*** with ***cudaMallocPitch,*** which has the following syntax:

*extern __host__ cudaError_t CUDARTAPI cudaMallocPitch(void **devPtr, size_t *pitch, size_t width, size_t height);*

- This translates to cudaMallocPitch (pointer to device memory pointer, pointer to pitch, desired width of the row in bytes, height of the array in bytes).
- If you have an array of 100 rows of 60 float elements, using the conventional cudaMalloc, you would allocate 100 x 60x sizeof(float) bytes, or 100 x 60 x 4 = 24,000 bytes.

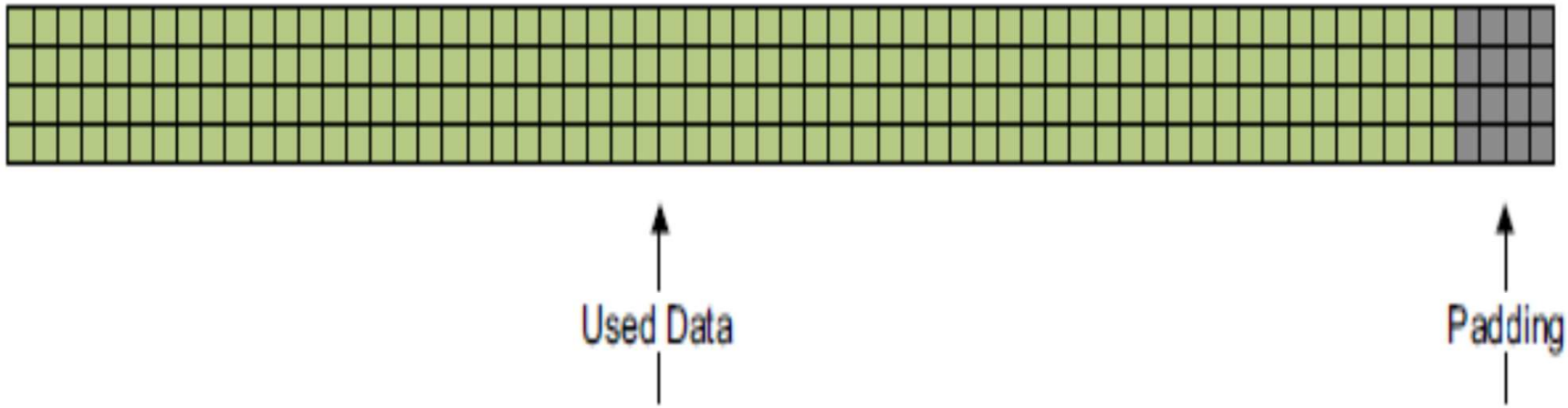
GLOBAL MEMORY

- Accessing arrayindex [1][0] (i.e., row one, element zero) would result in noncoalesced access. This is because the length of a single row of 60 elements would be 240 bytes, which is of course not a power of two.
- Using the ***cudaMallocPitch*** function the size of each row is padded by an

amount necessary for the alignment requirements of the given device.I

- In our example, it would in most cases be extended to 64 elements per row, or 256 bytes.
- The pitch the device actually uses is returned in the pitch parameters passed to cudaMallocPitch.

GLOBAL MEMORY



- Fig: Padding Achieved using CudaMallocPitch

GLOBAL MEMORY

- **High Bandwidth**: GPU global memory typically has a much higher bandwidth compared to system RAM, allowing for faster data access by the GPU cores.
- **Parallel Access**: It is designed to be accessed in parallel by multiple GPU cores simultaneously. This is crucial for efficient parallel processing, which is the primary strength of GPUs.
- **Large Capacity**:Modern GPUs have substantial global memory capacities, often several gigabytes or even terabytes, depending on the model and type of GPU.
- **Data Transfer:** Data is transferred between the CPU's main memory and the GPU's global memory when processing tasks are offloaded to the GPU. This transfer can be a potential bottleneck, and minimizing data transfer is often a consideration in optimizing GPU applications.

GLOBAL MEMORY

- **Heterogeneous Memory Architecture (HMA):** Some GPUs may have heterogeneous memory architectures that incorporate various types of memory, such as High Bandwidth Memory (HBM), Graphics Double Data Rate (GDDR) memory, and traditional GDDR or DDR RAM. These different memory types can offer varying levels of performance and energy efficiency.

- **Explicit Memory Management:** In many GPU programming models like CUDA (Compute Unified Device Architecture) and OpenCL, developers need to explicitly manage data movement between CPU and GPU memory. This explicit control allows developers to optimize data transfers and memory usage for specific tasks.

GLOBAL MEMORY: Searching

- The search we have two options: a binary search.
- A binary search takes advantage of the fact we have a sorted list of samples from the previous step.
- It works by dividing the list into two halves and asking whether the value it seeks is in the top or bottom half of the dataset.
- It then divides the list again and again until such time as it finds the value.
- The worst case sort time for a binary search is log2(N).

Parallel Searching

```
#include <iostream>
#include <cstdio>

// Kernel function for binary search
__global__ void binarySearchKernel(int *arr, int target, int left, int right, int *result) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
```

Parallel Searching

```
while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) {
            atomicExch(result, mid); // Store the result in a thread-safe manner
            return;
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
}
```

Parallel Searching

```
int main() {
    const int arraySize = 1024;
    const int target = 42;

    int *hostArray, *deviceArray, *deviceResult;

    // Allocate memory on host and device
    hostArray = new int[arraySize];
    cudaMalloc(&deviceArray, arraySize * sizeof(int));
    cudaMalloc(&deviceResult, sizeof(int));
```

Parallel Searching

```
    // Initialize the sorted array on the host
    for (int i = 0; i < arraySize; ++i) {
        hostArray[i] = i * 2;
    }
// Copy data from host to device
    cudaMemcpy(deviceArray, hostArray, arraySize * sizeof(int),
cudaMemcpyHostToDevice);


    // Set up grid and block dimensions
    int threadsPerBlock = 256;
    int blocksPerGrid = (arraySize + threadsPerBlock - 1) /
threadsPerBlock;
```

Parallel Searching

```
 // Launch kernel
    binarySearchKernel<<<blocksPerGrid, threadsPerBlock>>>(deviceArray, target, 0, arraySize - 1, deviceResult);
// Copy result from device to host
    int result;
    cudaMemcpy(&result, deviceResult, sizeof(int), cudaMemcpyDeviceToHost);
if (result != -1) {
        std::cout << "Element " << target << " found at index " << result << std::endl;
    } else {
        std::cout << "Element " << target << " not found in the array." << std::endl;
    }
```

Parallel Searching

```
    // Clean up
    delete[] hostArray;
    cudaFree(deviceArray);
    cudaFree(deviceResult);
    return 0;
}
```

TEXTURE MEMORY

- **Texture Memory**, also known as **Graphics Card Memory** or **Video Memory.**
- It is a specialized type of memory found in modern graphics processing units (GPUs).
- It plays a crucial role in rendering graphics and images on a computer screen, especially in video games and other graphics-intensive applications.

TEXTURE MEMORY

- When a GPU renders graphics, it needs to store textures, frame buffers, shaders, and other data required for visual rendering are stored in GPU texture memory.
- It is significantly faster than the system RAM (main memory) of the computer.
- Accessing data from GPU texture memory is much quicker, allowing for faster rendering and improved performance in graphics-intensive tasks.

TEXTURE MEMORY

- The texture memory is divided into several components such as:
- **Frame Buffer**: This is used to store the image that is currently being displayed on the screen. The size of the frame buffer is typically related to the resolution of the display.
- **Textures**: Textures are 2D or 3D images used to cover the surfaces of 3D objects in a scene. They provide details, colors, and patterns to enhance visual realism.
- **Shaders**: These are small programs that run on the GPU to control the rendering pipeline and manipulate the data stored in texture memory to create visual effects.

CONSTANT MEMORY

- Constant memory is a form of virtual addressing of global memory.
- There is no special reserved constant memory block.
- Constant memory has two special properties :
  - it is cached
  - it supports broadcasting a single value to all the elements within a warp.
- Constant memory, as its name suggests, is for read-only memory.
- It is either declared at compile time or defined at runtime as read only by host
- To declare a section of memory as constant at compile time, you simply use the __constant__
- keyword. For example:
- ___constant__ float my_array[1024] = { 0.0F, 1.0F, 1.34F, . };

CONSTANT MEMORY

- constant memory" refers to a type of memory available on NVIDIA GPUs that offers special properties for read-only data.
- Limited Size: Constant memory is limited in size and typically ranges from tens of kilobytes to a few hundred kilobytes, depending on the specific GPU architecture.
- High Bandwidth: Constant memory has high bandwidth compared to regular global memory on the GPU.
- Initialization: The constant memory is initialized from the host (CPU) before launching the GPU kernel.

Summary

The following Topics were discussed:

- GPU processor Architecture
- CUDA Hardware overview
  - Threads
  - Blocks
  - Warps
  - Grids
- Memory Handling with CUDA
  - Global Memory
  - Constant Memory
  - Texture memory
  - Shared memory
- CUDA Programming
  - Sorting
  - Vector addition
  - Searching

Test Your Understanding

- Write a note on CUDA hardware overview.
- Write a CUDA program to sort a given list using Radix sort
- Write a CUDA program to search an element in a given list.

Assignment

- Write a CUDA program to sort an array of numbers.
- Write a CUDA program to search an element in a given list of elements