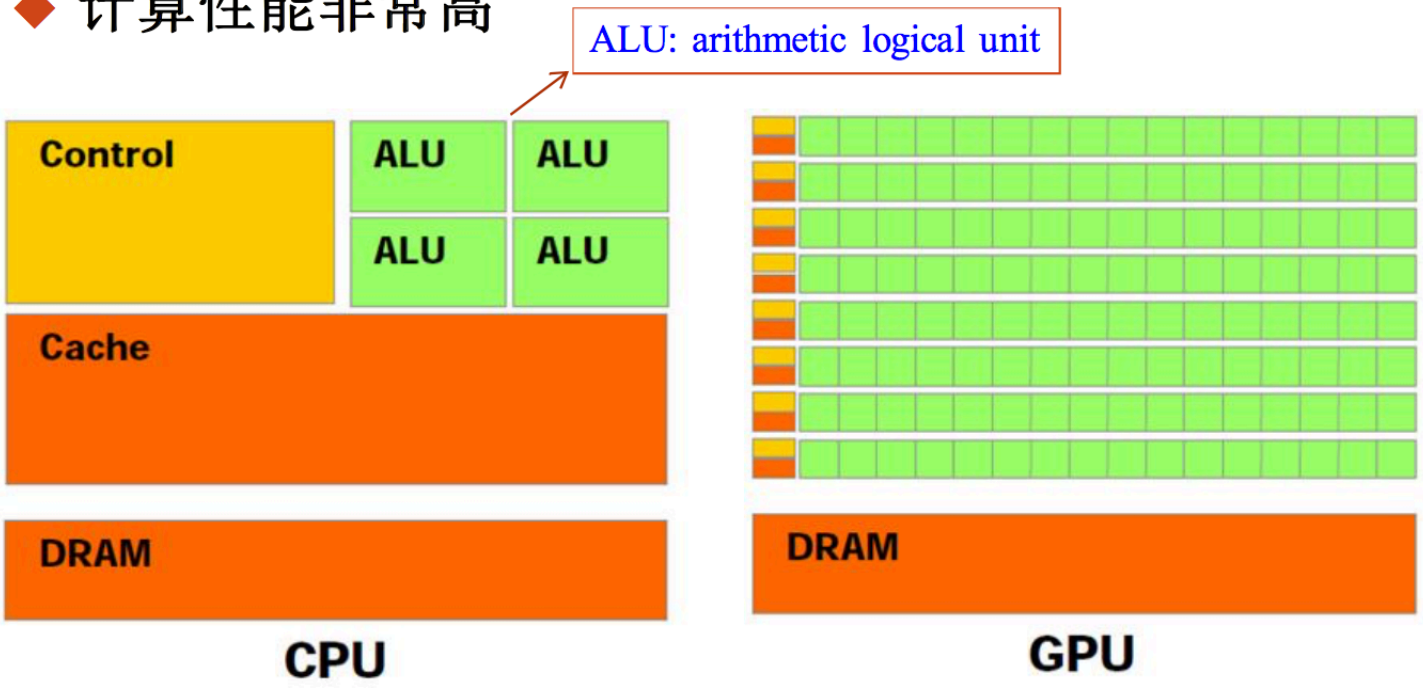
Advanced Big Data Analytics

Chapter 11 -- CUDA C/C++ BASICS

GPU vs CPU

- GPUs contain much larger number of dedicated ALUs then CPUs.
- GPUs also contain extensive support of Stream Processing paradigm. It is related to SIMD (Single Instruction Multiple Data) processing.
- Each processing unit on GPU contains local memory that improves data manipulation and reduces fetch time.

◆ 计算性能非常高



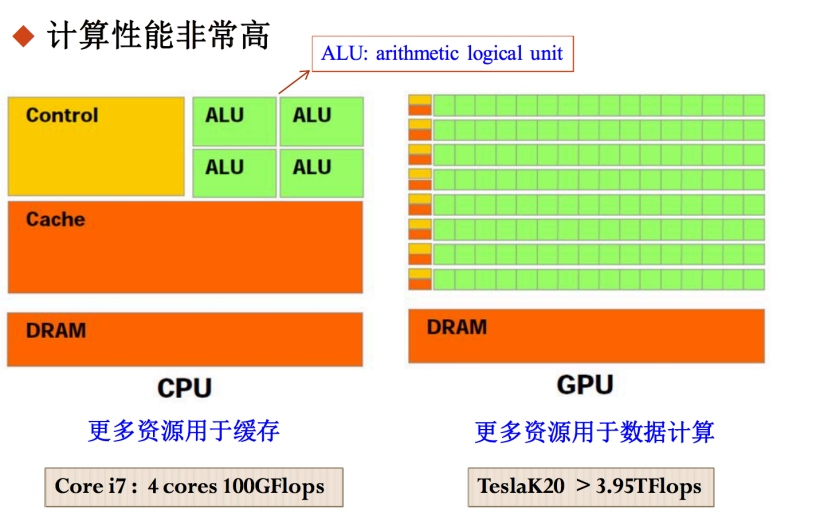
更多资源用于缓存

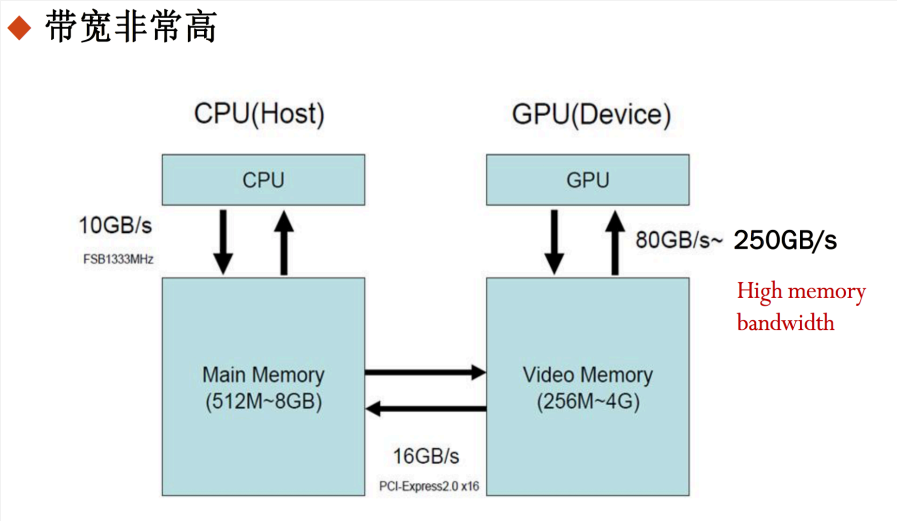
更多资源用于数据计算

Core i7： 4 cores 100GFlops

TeslaK20 > 3.95TFlops

GPGPU





Pros of GPU:

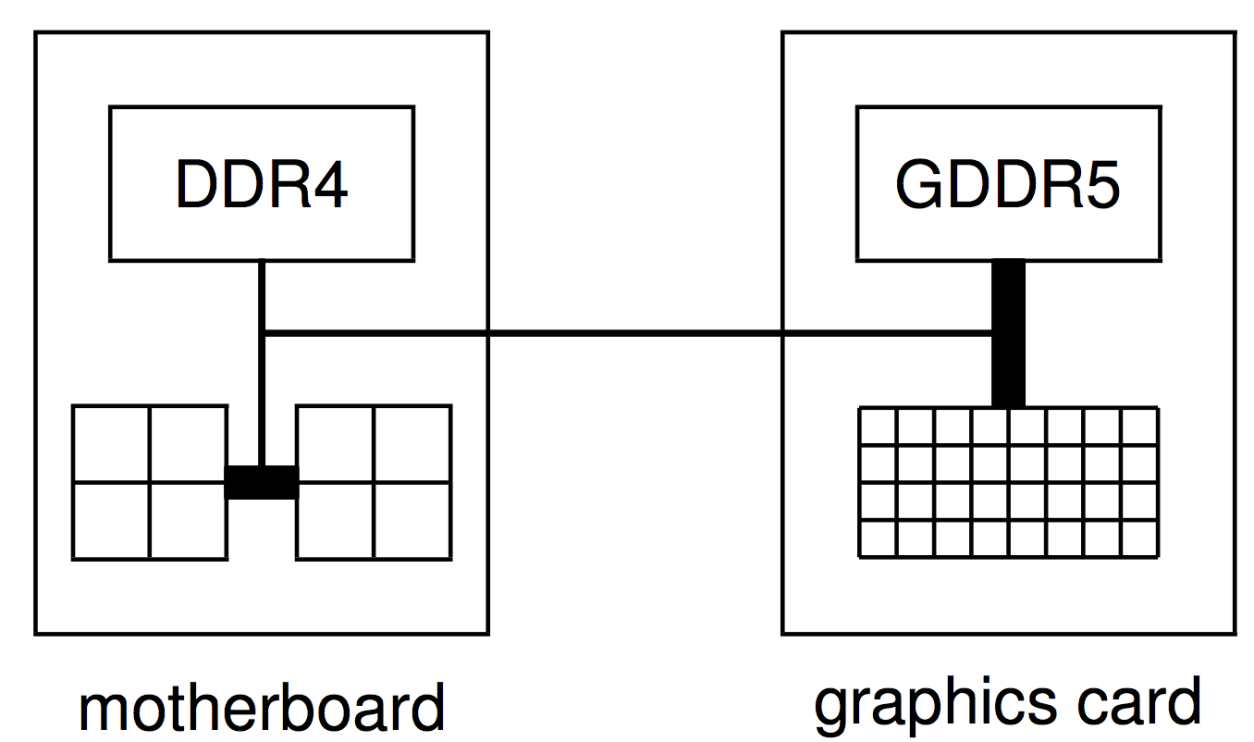
- 更大的内存带宽内存带宽;
- 更大量的执行单元;
- 对比CPU, 显卡的价格较为低廉

Cons of GPU:

- 对于不能高度并行化的工作，帮助不大
- 不具有分支预测
- GPGPU程序模型仍不成熟，无公认标准

Hardware view of GPU

At the top-level, a PCIe graphics card with a many-core GPU and high-speed graphics “device” memory sits inside a standard PC/server with one or two multicore CPUs:



Hardware view of NVIDIA GPU

Currently, 3 generations of hardware cards in use:

- Kepler:
  first released in 2012
  includes HPC cards with excellent double precision, e.g. K40s and K80s
- Maxwell:
  first released in 2014
  only gaming cards, not HPC, so poor DP
- Pascal:
  just released in 2016
  The new Pascal generation has cards for both gaming/VR and HPC

Hardware view of NVIDIA GPUs

The new Pascal generation has cards for both gaming/VR and HPC

Consumer graphics cards (GeForce):

- GTX 1060: 1280 cores, 6GB (£240)
- GTX 1070: 1920 cores, 8GB (£400)
- GTX 1080: 2560 cores, 8GB (£600)
- GTX Titan X: 3584 cores, 12GB (£1200)

HPC (Tesla):

- P100: 3584 cores, 16GB HBM2 (£4-8k?)

Hardware view of Pascal GPUs

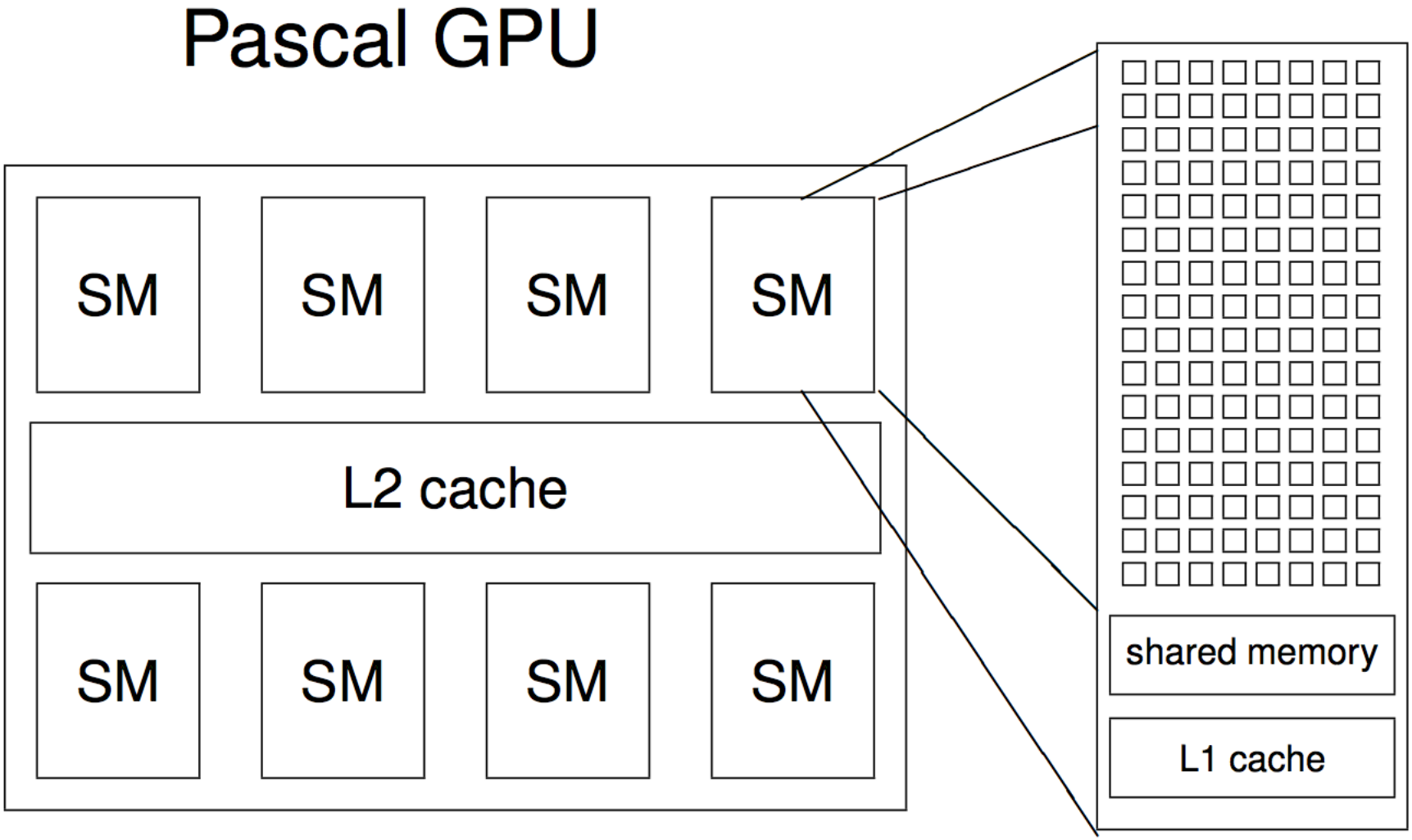
building block is a “streaming multiprocessor” (SM):

- 128 cores (64 in P100) and 64k registers
- 96KB (64KB in P100) of shared memory
- 48KB (24KB in P100) L1 cache
- 8-16KB (?) cache for constants
- up to 2K threads per SM

Different chips have different numbers of these SMs:

product	SMs	bandwidth	memory	power
GTX 1060	10	192 GB/s	6 GB	120W
GTX 1070	16	256 GB/s	8 GB	150W
GTX 1080	20	320 GB/s	8 GB	180W
GTX Titan X	28	480 GB/s	12 GB	250W
P100	56	720 GB/s	16 GB HBM2	300W

Hardware view of NVIDIA GPUs



Hardware view of Maxwell GPUs

There are multiple products in the Maxwell generation, but none with good double precision performance for HPC

Consumer graphics cards (GeForce):

- GTX 960: 1024 cores, 2GB (£170 $\rightarrow$ £155)
- GTX 980: 2048 cores, 4GB (£450 $\rightarrow$ £400)
- (old) GTX Titan X: 3076 cores, 12GB (£1000)

Multithreading

Key hardware feature is that the cores in a SM are SIMT (Single Instruction Multiple Threads) cores: a.k.a, SIMD

- groups of 32 cores execute the same instructions simultaneously, but with different data
- similar to vector computing on CRAY supercomputers
- 32 threads all doing the same thing at the same time
- natural for graphics processing and much scientific computing
- SIMT is also a natural choice for many-core chips to simplify each core

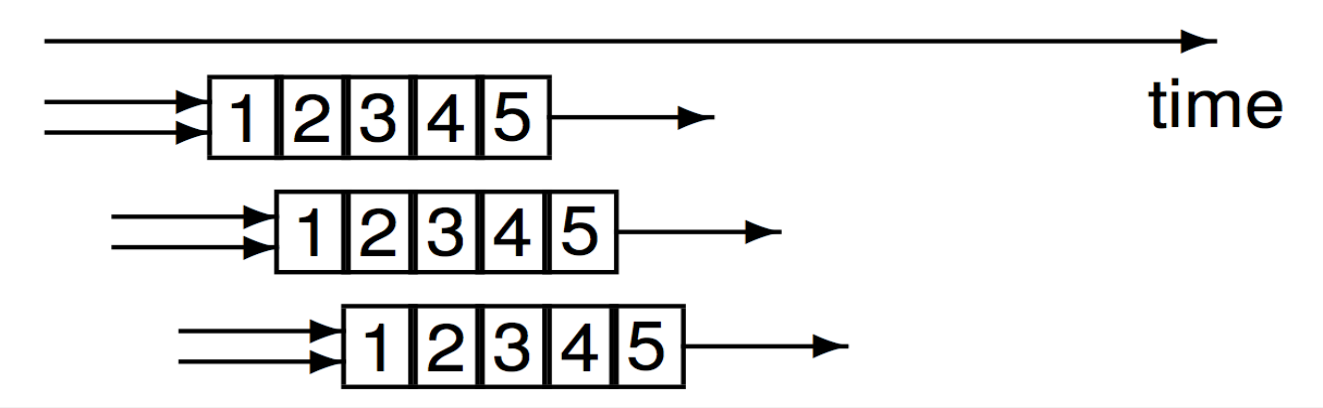
Multithreading

Lots of active threads is the key to high performance:

- no “context switching”; each thread has its own registers, which limits the number of active threads
- threads on each SM execute in groups of 32 called “warps” – execution alternates between “active” warps, with warps becoming temporarily “inactive” when waiting for data

Multithreading

originally, each thread completed one operation before the next started to avoid complexity of pipeline overlaps



however, NVIDIA have now relaxed this, so each thread can have multiple independent instructions overlapping

memory access from device memory has a delay of 200-400 cycles; with 40 active warps this is equivalent to 5-10 operations, so enough to hide the latency?

Software view

At the top level, we have a master process which runs on the CPU and performs the following steps:

1.initialises card
2. allocates memory in host and on device
3. copies data from host to device memory
4. launches multiple instances of execution “kernel” on device
5. copies data from device memory to host
6. repeats 3-5 as needed
7. de-allocates all memory and terminates

Software view

At a lower level, within the GPU

- each instance of the execution kernel executes on a SM
- if the number of instances exceeds the number of SMs, then more than one will run at a time on each SM if there are enough registers and shared memory, and the others will wait in a queue and execute later
- all threads within one instance can access local shared memory but can’t see what the other instances are doing (even if they are on the same SM)
- there are no guarantees on the order in which the instances execute

What is CUDA?

- CUDA Architecture
  - Expose GPU parallelism for general-purpose computing
  - GPUs allow creation of very large number of concurrently executed threads at very low system resource cost
- CUDA C/C++
  - Based on industry-standard C/C++
  - Small set of extensions to enable heterogeneous programming
  - Straightforward APIs to manage devices, memory etc.
  - CUDA compiler uses variation of C with future support of C++
  - CUDA was released on February 15, 2007 for PC and Beta version for MacOS X on August 19, 2008.

Why CUDA?

- CUDA provides ability to use high-level languages such as C to develop application that can take advantage of high level of performance and scalability that GPUs architecture offer.
- GPUs allow creation of very large number of concurrently executed threads at very low system resource cost.
- CUDA also exposes fast shared memory (16KB) that can be shared between threads.
- Full support for integer and bitwise operations.
- Compiled code will run directly on GPU.

CUDA limitations

- No support of recursive function. Any recursive function must be converted into loops.
- Many deviations from Floating Point Standard (IEEE 754).
- No texture rendering.
- Bus bandwidth and latency between GPU and CPU is a bottleneck for many applications.
- Threads should only be run in groups of 32 and up for best performance.
- Only supported on NVidia GPUs

执行模式

1. 内存存取 latency 的问题：
   - CPU 通常使用 cache 来减少存取主内存的次数，以避免内存 latency 影响到执行效率。显示芯片则多半没有 cache（或很小），而利用并行化执行的方式来隐藏内存的latency。即，当第一个thread需要等待内存读取结果时，则开始执行第二个thread，依此类推）；
2. 分支指令的问题：CPU 通常利用分支预测等方式来减少分支指令造成的pipeline bubble。显示芯片则多半使用类似处理内存 latency 的方式。不过，通常显示芯片处理分支的效率会比较差。

因此，最适合利用 CUDA 处理的问题，是可以大量并行化的问题，才能有效隐藏内存的 latency，并有效利用显示芯片上的大量执行单元。使用 CUDA 时，同时有上千个 thread 在执行是很正常的。因此，如果不能大量并行化的问题，使用 CUDA 就没办法达到最好的效率了。

```
fu@fu-Precision-Tower-7910:/usr/local/cuda-8.0$ ls -lh
total 60K
-rw-r--r--  1 root root  365 Sep 15  2016 LICENSE
-rw-r--r--  1 root root  365 Sep 15  2016 README
drwxr-xr-x  3 root root 4.0K Oct 23  2016 bin
drwxr-xr-x  5 root root 4.0K Oct 23  2016 doc
drwxr-xr-x  5 root root 4.0K Oct 23  2016 extras
lrwxrwxrwx  1 root root   28 Sep 15  2016 include -> targets/x86_64-linux/include
lrwxrwxrwx  1 root root   24 Sep 15  2016 lib64 -> targets/x86_64-linux/lib
drwxr-xr-x  8 root root 4.0K Oct 23  2016 libnsight
drwxr-xr-x  7 root root 4.0K Oct 23  2016 libnvvp
drwxr-xr-x  3 root root 4.0K Oct 23  2016 nvml
drwxr-xr-x  7 root root 4.0K Oct 23  2016 nvvm
drwxr-xr-x 11 root root 4.0K Oct 23  2016 samples
drwxr-xr-x  3 root root 4.0K Oct 23  2016 share
drwxr-xr-x  2 root root 4.0K Oct 23  2016 src
drwxr-xr-x  3 root root 4.0K Oct 23  2016 targets
drwxr-xr-x  2 root root 4.0K Oct 23  2016 tools
-rw-r--r--  1 root root   20 Sep 15  2016 version.txt
fu@fu-Precision-Tower-7910:/usr/local/cuda-8.0$
```

- bin -- 工具程序及动态链接库
- doc -- 文件
- include -- header 文档
- lib -- 链接库档案
- open64 -- 基于 Open64 的 CUDA compiler
- src -- 一些原始码

安装程序也会设定一些环境变量，包括：

- CUDA_BIN_PATH -- 工具程序的目录，默认为 C:\CUDA\bin
- CUDA_INC_PATH -- header 文件的目录，默认为 C:\CUDA\inc
- CUDA_LIB_PATH -- 链接库文件的目录，默认为 C:\CUDA\lib

CUDA Components

Installing CUDA on a system, there are 3 components:

- driver
  - low-level software that controls the graphics card
- toolkit
  - nvcc CUDA compiler
  - Nsight IDE plugin for Eclipse or Visual Studio
  - profiling and debugging tools
  - several libraries
- SDK
  - lots of demonstration examples
  - some error-checking utilities
  - not officially supported by NVIDIA
  - almost no documentation

GCC

gcc(选项)(参数)

编译成可执行文件
gcc –c –I /usr/dev/mysql/include test.c –o test.o

最后我们把所有目标文件链接成可执行文件:
gcc –L /usr/dev/mysql/lib –lmysqlclient test.o –o test

Linux下的库文件分为两大类分别是动态链接库（通常以.so结尾）和静态链接库（通常以.a结尾），二者的区别仅在于程序执行时所需的代码是在运行时动态加载的，还是在编译时静态加载的

GCC在链接时优先使用动态链接库，只有当动态链接库不存在时才考虑使用静态链接库，如果需要的话可以在编译时加上-static选项，强制使用静态链接库
gcc –L /usr/dev/mysql/lib –static –lmysqlclient test.o –o test

CUDA programming

Already explained that a CUDA program has two pieces:

- host code on the CPU which interfaces to the GPU
- kernel code which runs on the GPU

At the host level, there is a choice of 2 APIs (Application Programming Interfaces):

- runtime

  simpler, more convenient
- driver

  much more verbose, more flexible (e.g. allows run-time compilation), closer to OpenCL

We will only use the runtime API in this course

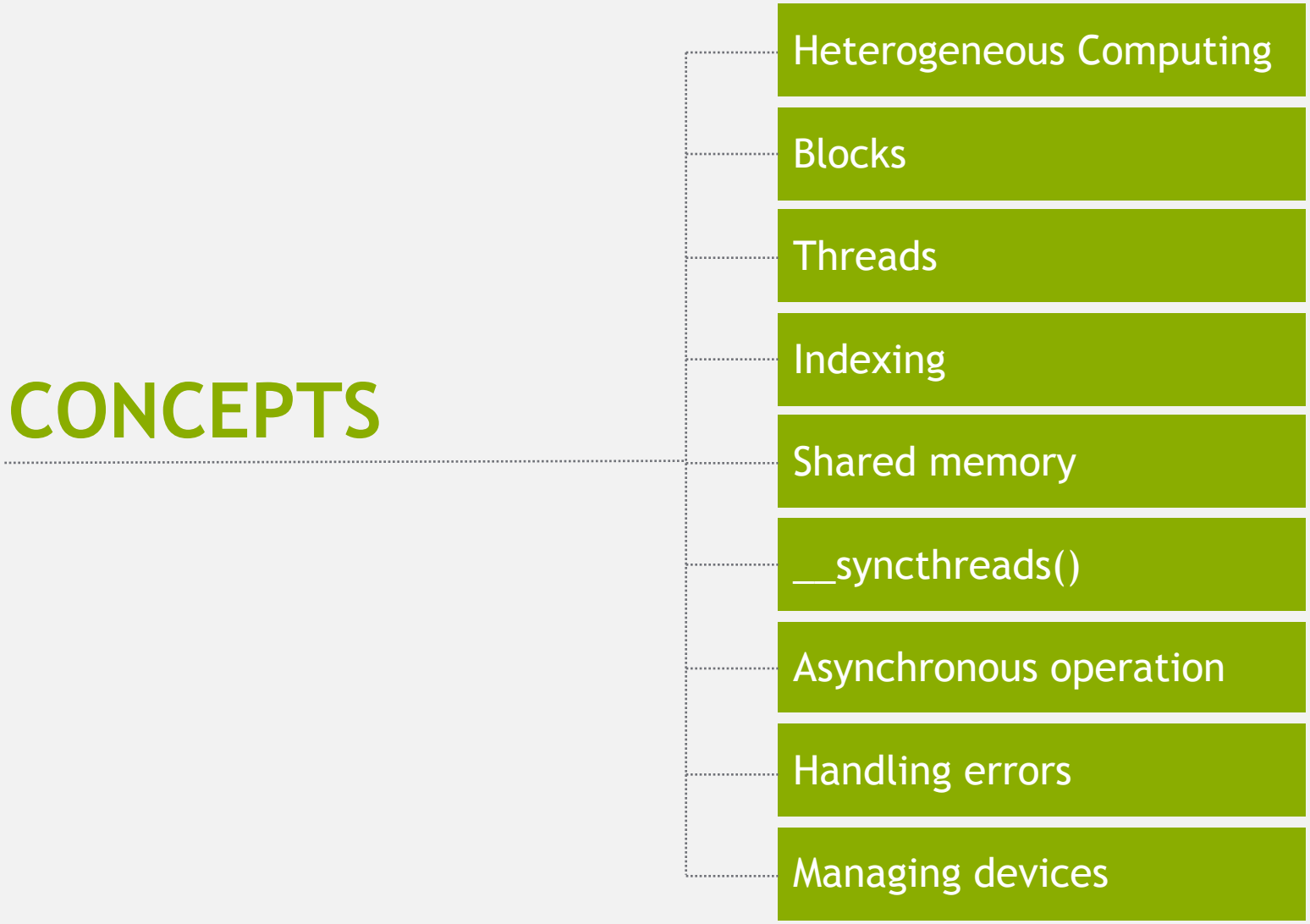
Introduction to CUDA C/C++

- What will you learn in this session?
  - Start from “Hello World!”
  - Write and launch CUDA C/C++ kernels
  - Manage GPU memory
  - Manage communication and synchronization

Prerequisites

- You (probably) need experience with C or C++
- You don’t need GPU experience
- You don’t need parallel programming experience
- You don’t need graphics experience

CONCEPTS


Heterogeneous Computing
Blocks
Threads
Indexing
Shared memory
__syncthreads()
Asynchronous operation
Handling errors
Managing devices

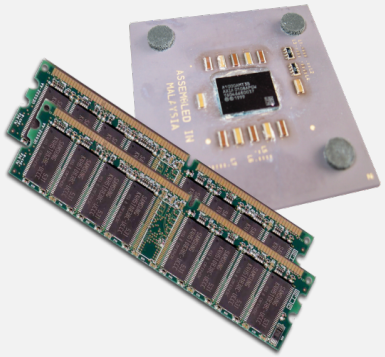
HELLO WORLD!

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

Heterogeneous Computing

- Terminology:
  - *Host* The CPU and its memory (host memory)
  - *Device* The GPU and its memory (device memory)



Host

Device

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;
}
```

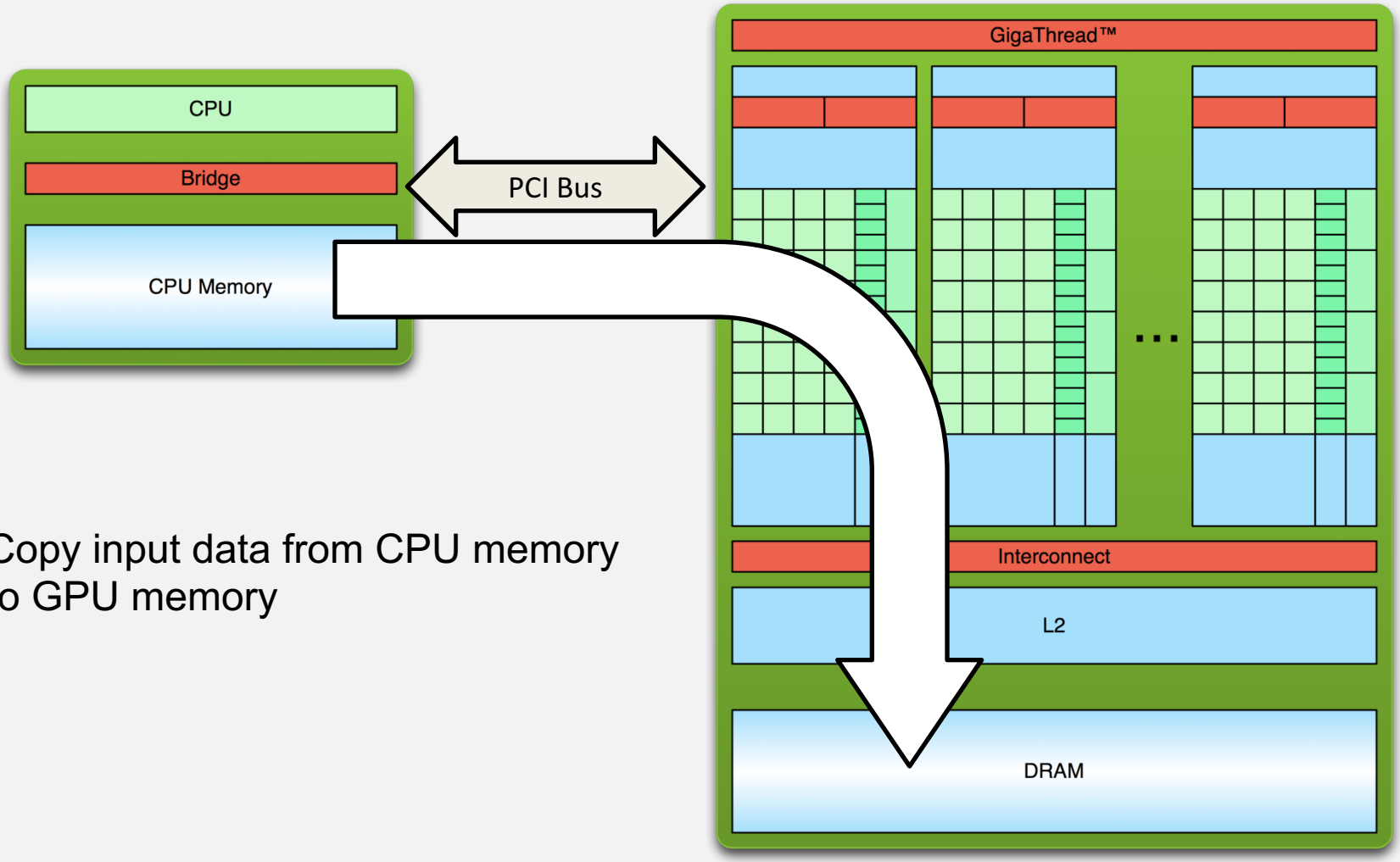
parallel fn

serial code

parallel code

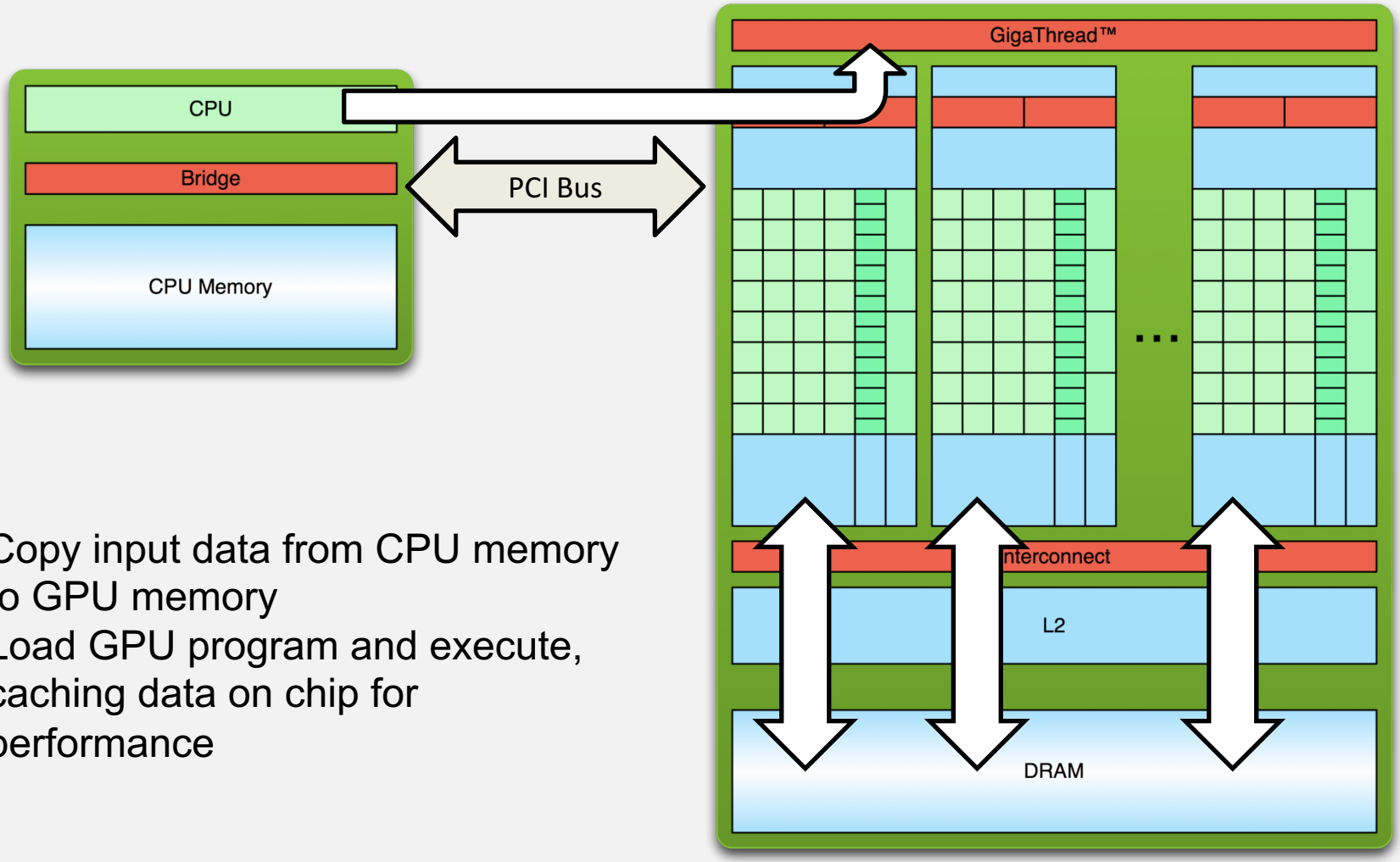
serial code

Simple Processing Flow



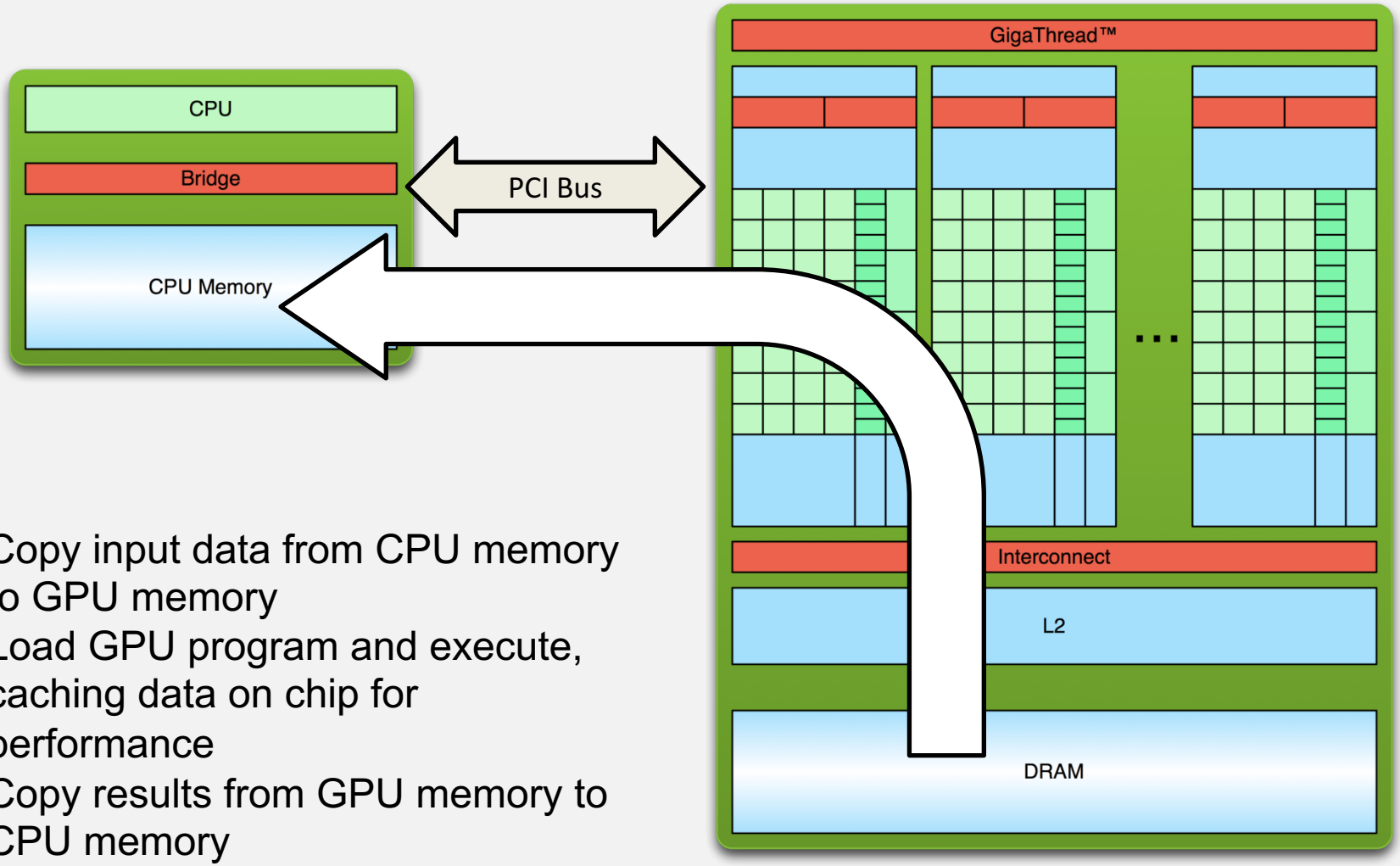
1. Copy input data from CPU memory to GPU memory

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

Hello World!

```
int main(void) {
        printf("Hello World!\n");
        return 0;
}
```

- Standard C that runs on the host
- NVIDIA compiler (nvcc) can be used to compile programs with no *device* code

Output:

```
$ nvcc hello_world.cu
$ a.out
Hello World!
$
```

Hello World! with Device Code

```
__global__ void mykernel(void) {
}

int main(void) {
      mykernel<<<1,1>>>();
      printf("Hello World!\n");
      return 0;
}
```

- Two new syntactic elements…

函式名称<<<block 数目, thread 数目, shared memory 大小>>>(参数...);

Hello World! with Device Code

```
__global__ void mykernel(void) {
}
```

- CUDA C/C++ keyword `__global__` indicates a function that:
  - Runs on the device
  - Is called from host code

- `nvcc` separates source code into host and device components
  - Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
  - Host functions (e.g. `main()`) processed by standard host compiler
    - **`gcc`, `cl.exe`**

Hello World! with Device Code

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

- Triple angle brackets mark a call from *host* code to *device* code
  - Also called a “kernel launch”
  - We’ll return to the parameters (1,1) in a moment

- That’s all that is required to execute a function on the GPU!

CUDA programming

In its simplest form it looks like:
kernel_routine<<<gridDim, blockDim>>>(args);

- gridDim is the number of instances of the kernel (the “grid” size), i.e., number of blocks
- blockDim is the number of threads within each instance (the “block” size) , i.e. , number of threads.
- args is a limited number of arguments, usually mainly pointers to arrays in graphics memory, and some constants which get copied by value

The more general form allows gridDim and blockDim to be 2D or 3D to simplify application programs

函式名称<<<block 数目, thread 数目, shared memory 大小>>>(参数...);

CUDA programming

At the lower level, when one instance of the kernel is started on a SM it is executed by a number of threads, each of which knows about:

- some variables passed as arguments
- pointers to arrays in device memory (also arguments)
- global constants in device memory
- shared memory and private registers/local variables
- some special variables:
  - *gridDim* size (or dimensions) of grid of blocks
  - *blockDim* size (or dimensions) of each block
  - *blockIdx* index (or 2D/3D indices) of block
  - *threadIdx* index (or 2D/3D indices) of thread
  - *warpSize* always 32 so far, but could change

Hello World! with Device Code

```
__global__ void mykernel(void){
}

int main(void) {
        mykernel<<<1,1>>>();
        printf("Hello World!\n");
        return 0;
}
```

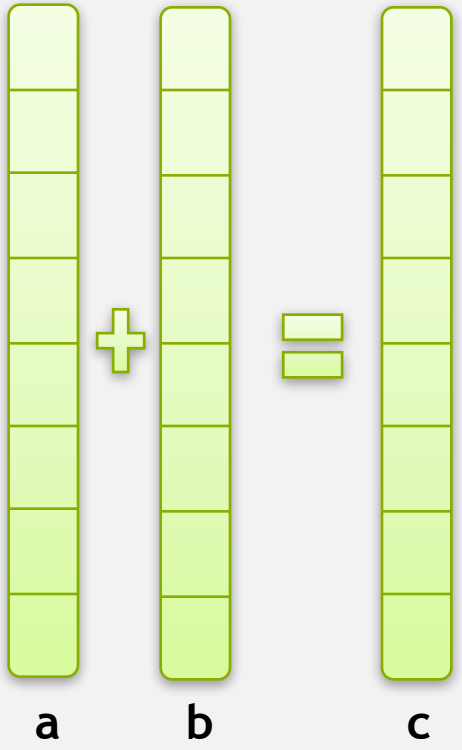
Output:

```
$ nvcc
hello.cu
$ a.out
Hello World!
$
```

- `mykernel()` does nothing, somewhat anticlimactic!

Parallel Programming in CUDA C/C++

- But wait… GPU computing is about massive parallelism!
- We need a more interesting example…
- We'll start by adding two integers and build up to vector addition



Addition on the Device

- A simple kernel to add two integers

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

- As before `__global__` is a CUDA C/C++ keyword meaning
  - `add()` will execute on the device
  - `add()` will be called from the host

Addition on the Device

- Note that we use pointers for the variables

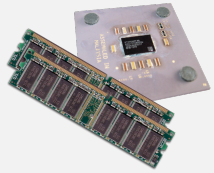
```
__global__ void add(int *a, int *b, int *c) {
        *c = *a + *b;
}
```

- `add()` runs on the device, so `a`, `b` and `c` must point to device memory

- We need to allocate memory on the GPU

Memory Management

- Host and device memory are separate entities
  - *Device* pointers point to GPU memory
    - May be passed to/from host code
    - May *not* be dereferenced in host code
  - *Host* pointers point to CPU memory
    - May be passed to/from device code
    - May *not* be dereferenced in device code



- Simple CUDA API for handling device memory
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Similar to the C equivalents `malloc()`, `free()`, `memcpy()`

Addition on the Device: `add()`

- Returning to our `add()` kernel

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

- Let's take a look at main()…

Addition on the Device: `main()`

```
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);

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

Addition on the Device: `main()`

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

    // 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;
}
```

Functions in details

cudaError_t cudaMemcpy (void * *dst,* const void * *src,* size_t *count,* enum **cudaMemcpyKind** *kind*)

Parameters:
dst - Destination memory address
src - Source memory address
count - Size in bytes to copy
kind - Type of transfer

Others:
cudaMemcpy2D, **cudaMemcpyToArray**, **cudaMemcpy2DToArray**, **cudaMemcpyFromArray**, **cudaMemcpy2DFromArray**, **cudaMemcpyArrayToArray**, **cudaMemcpy2DArrayToArray**, **cudaMemcpyToSymbol**, **cudaMemcpyFromSymbol**, **cudaMemcpyAsync**, **cudaMemcpy2DAsync**, **cudaMemcpyToArrayAsync**, **cudaMemcpy2DToArrayAsync**, **cudaMemcpyFromArrayAsync**, **cudaMemcpy2DFromArrayAsync**, **cudaMemcpyToSymbolAsync**, **cudaMemcpyFromSymbolAsync**

RUNNING IN PARALLEL

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

Hardware view of Pascal GPUs

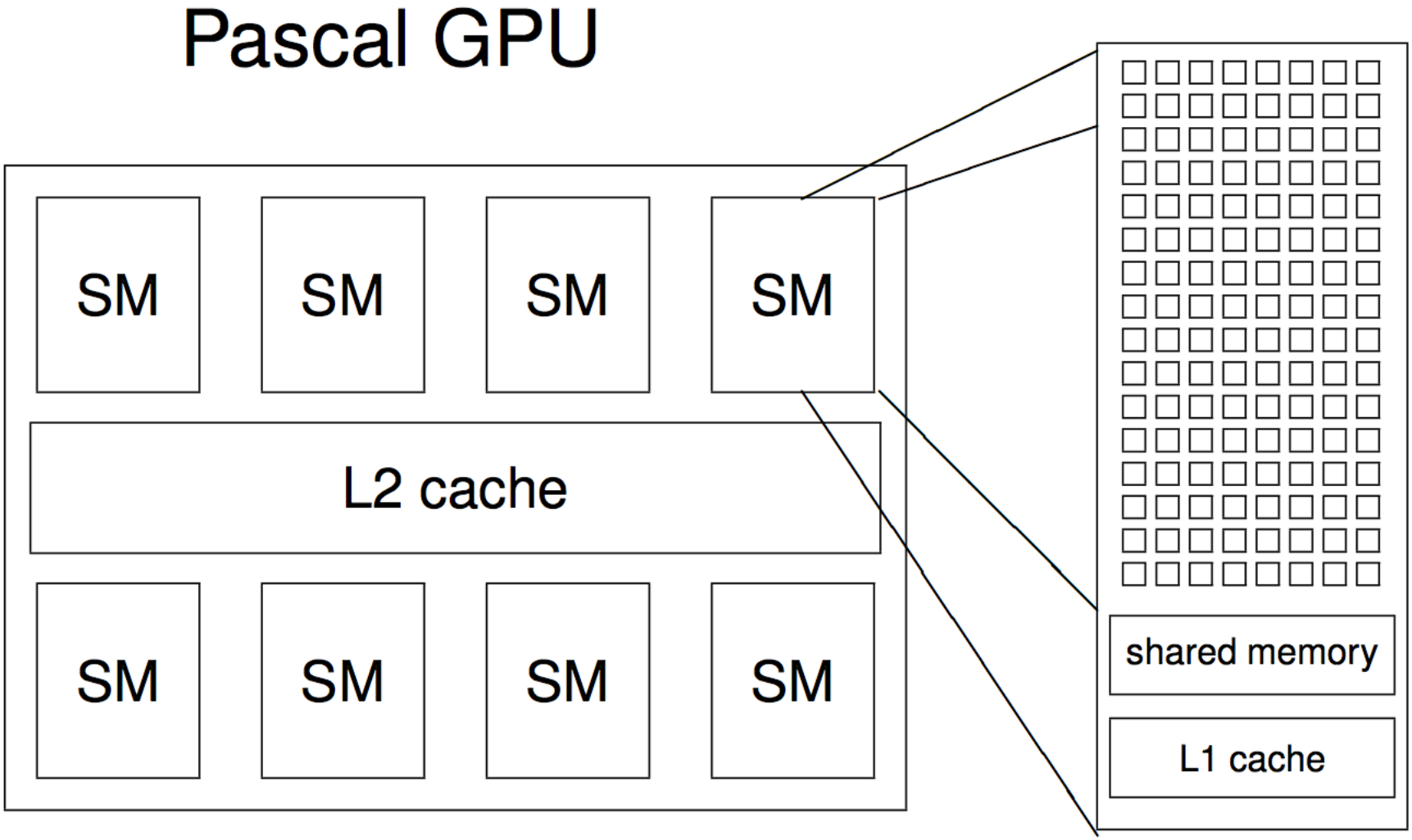
building block is a “streaming multiprocessor” (SM):

- 128 cores (64 in P100) and 64k registers
- 96KB (64KB in P100) of shared memory
- 48KB (24KB in P100) L1 cache
- 8-16KB (?) cache for constants
- up to 2K threads per SM

Different chips have different numbers of these SMs:

product	SMs	bandwidth	memory	power
GTX 1060	10	192 GB/s	6 GB	120W
GTX 1070	16	256 GB/s	8 GB	150W
GTX 1080	20	320 GB/s	8 GB	180W
GTX Titan X	28	480 GB/s	12 GB	250W
P100	56	720 GB/s	16 GB HBM2	300W

Hardware view of NVIDIA GPUs



Moving to Parallel

- GPU computing is about massive parallelism
  - So how do we run code in parallel on the device?

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

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

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

Vector Addition on the Device

- With `add()` running in parallel we can do vector addition

- Terminology: 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`

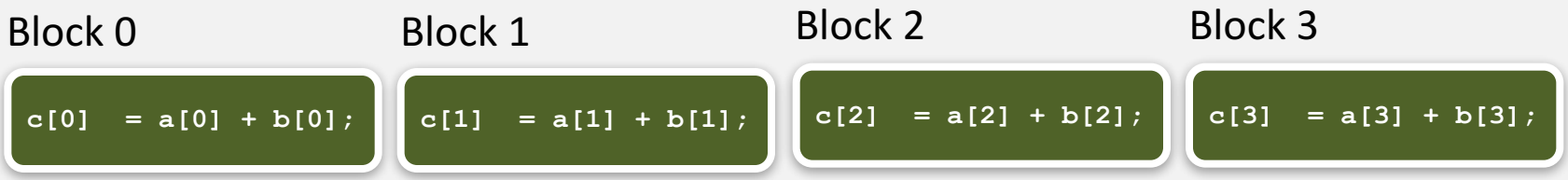
```
__global__ void add(int *a, int *b, int *c) {
        c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- By using `blockIdx.x` to index into the array, each block handles a different index

Vector Addition on the Device

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

- On the device, each block can execute in parallel:



Vector Addition on the Device: `add()`

- Returning to our parallelized `add()` kernel

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

- Let's take a look at main()…

Vector Addition on the Device: `main()`

```
#define N 512
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 = N * sizeof(int);

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

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Vector Addition on the Device: `main()`

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

    // Launch add() kernel on GPU with N blocks
    add<<<N,1>>>(d_a, d_b, d_c);

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

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Review (1 of 2)

- Difference between *host* and *device*
  - *Host* CPU
  - *Device* GPU

- Using `__global__` to declare a function as device code
  - Executes on the device
  - Called from the host

- Passing parameters from host code to a device function

Review (2 of 2)

- Basic device memory management
  - `cudaMalloc()`
  - `cudaMemcpy()`
  - `cudaFree()`

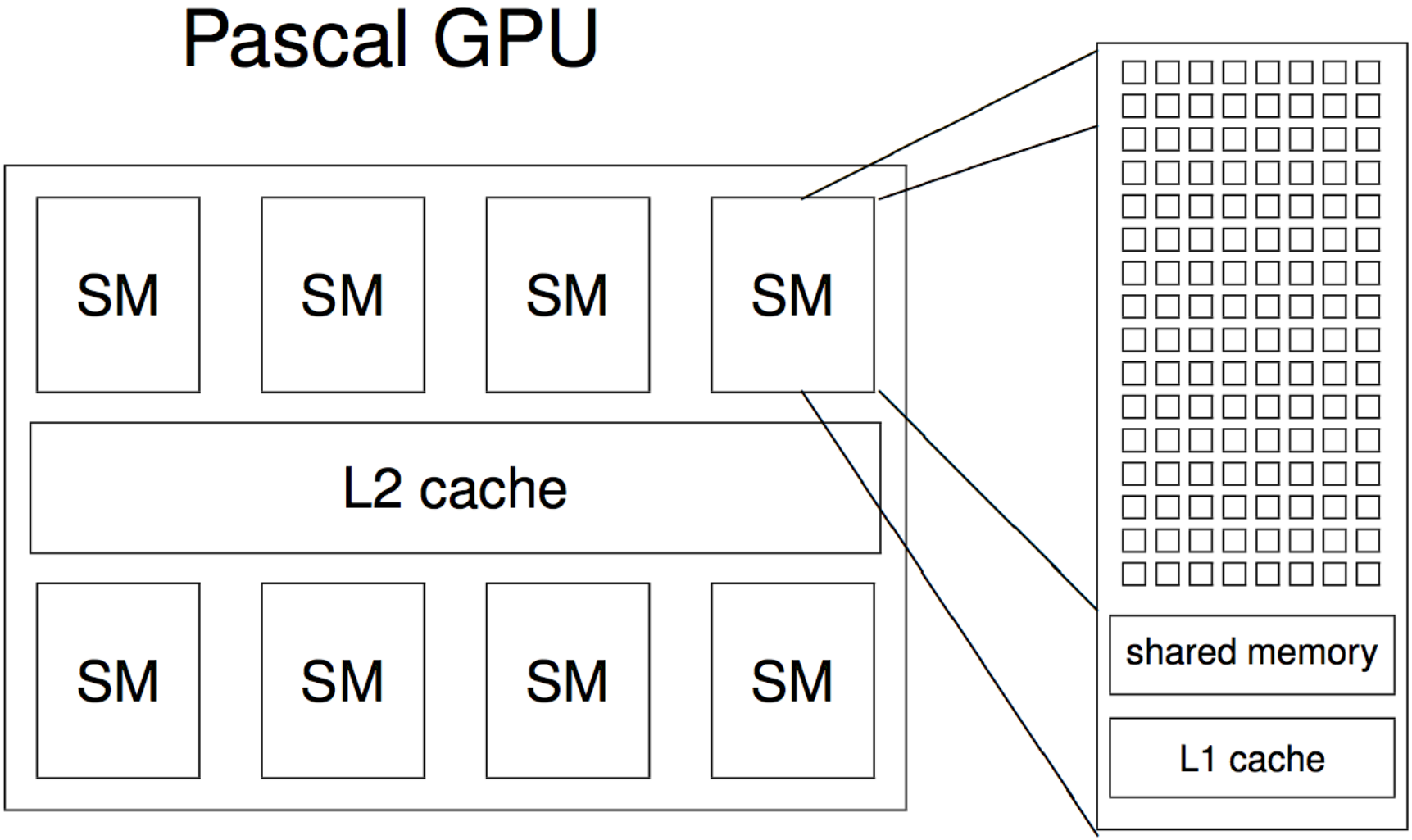
- Launching parallel kernels
  - Launch `N` copies of `add()` with `add<<<N,1>>>(…);`
  - Use `blockIdx.x` to access block index

INTRODUCING THREADS

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

Hardware view of NVIDIA GPUs



CUDA Threads

- Terminology: a block can be split into parallel threads

- Let's change `add()` to use parallel *threads* instead of parallel *blocks*

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

- We use `threadIdx.x` instead of `blockIdx.x`

- Need to make one change in `main()`...

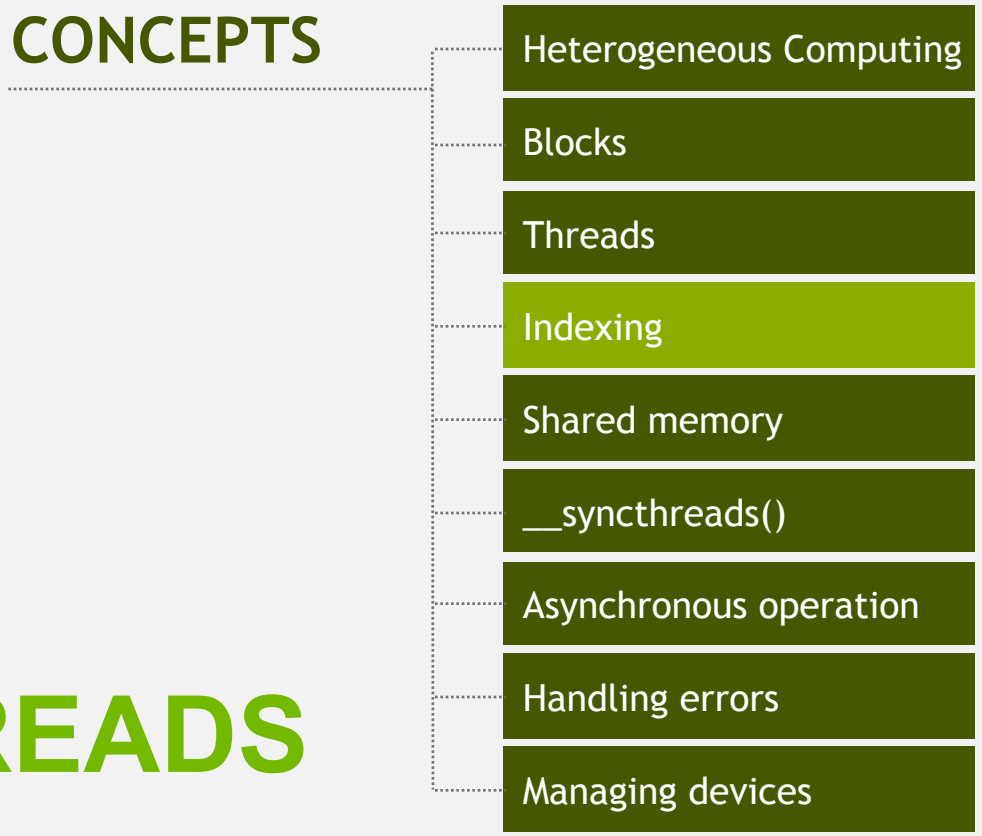
Vector Addition Using Threads: `main()`

```
#define N 512
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 = N * sizeof(int);

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

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

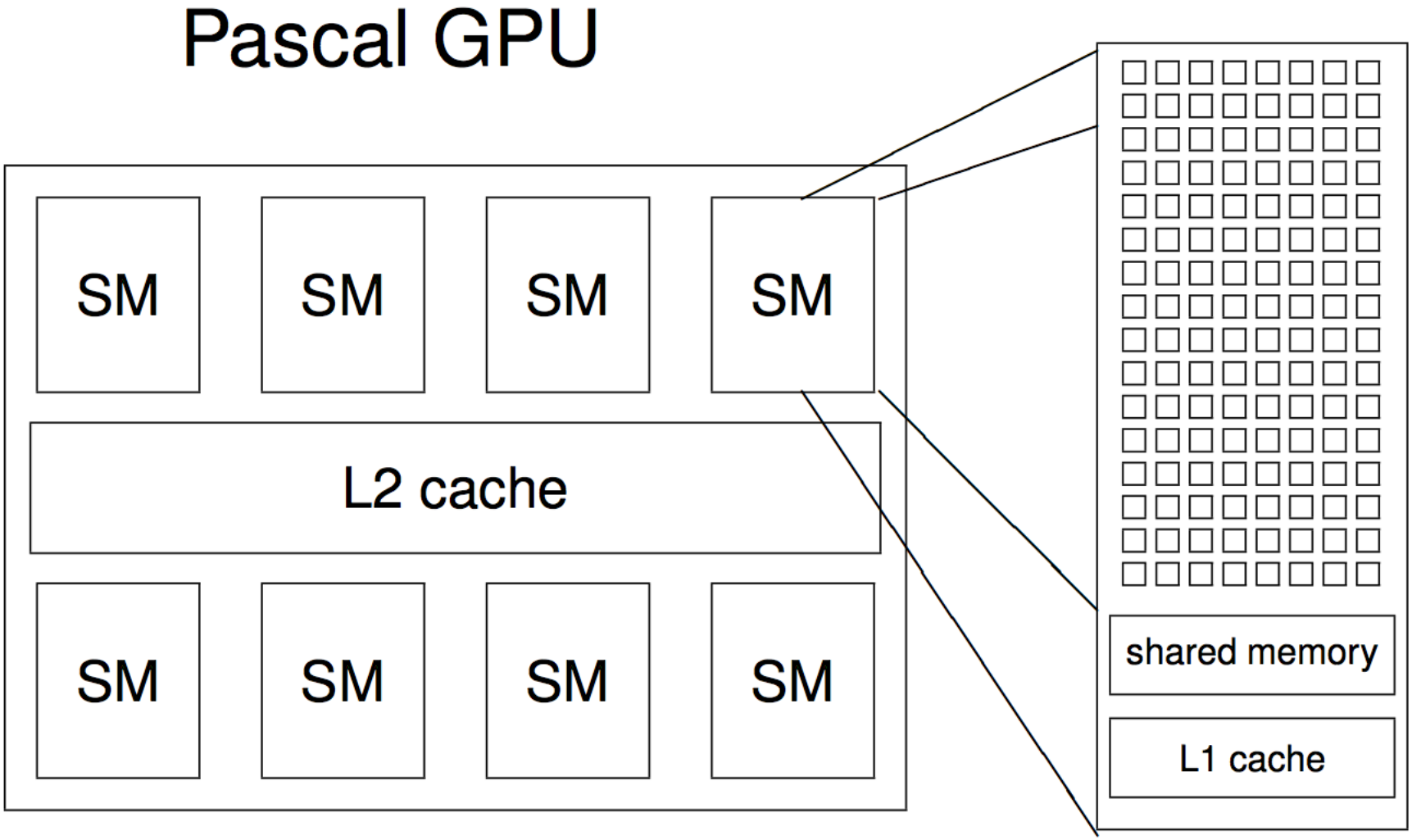
Vector Addition Using Threads: `main()`

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

    // Launch add() kernel on GPU with N threads
    add<<<1,N>>>(d_a, d_b, d_c);

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

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

COMBINING THREADS AND BLOCKS



Hardware view of NVIDIA GPUs

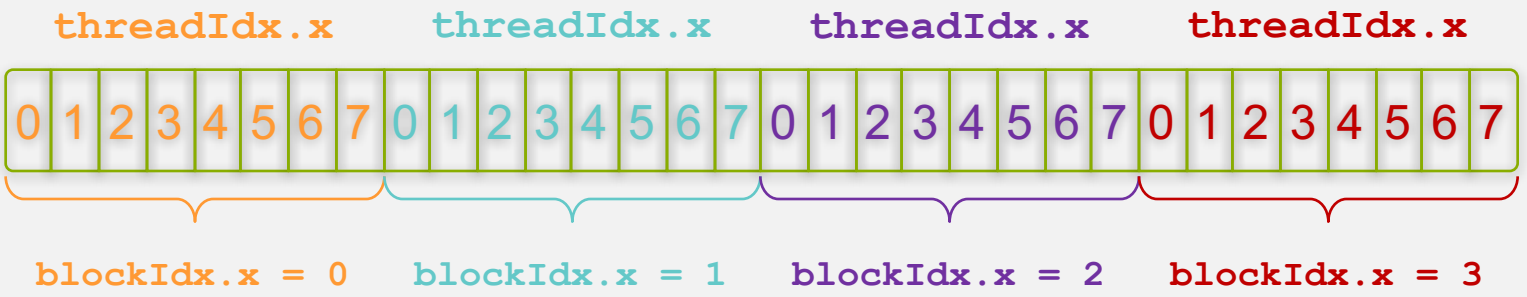


Combining Blocks and Threads

- We've seen parallel vector addition using:
  - Many blocks with one thread each (blockIdx.x)
  - One block with many threads (threadIdx.x)
- Let's adapt vector addition to use both blocks and threads
- Why? We'll come to that…
- First let's discuss data indexing…

Indexing Arrays with Blocks and Threads

- No longer as simple as using `blockIdx.x` and `threadIdx.x`
  - Consider indexing an array with one element per thread (8 threads/block)

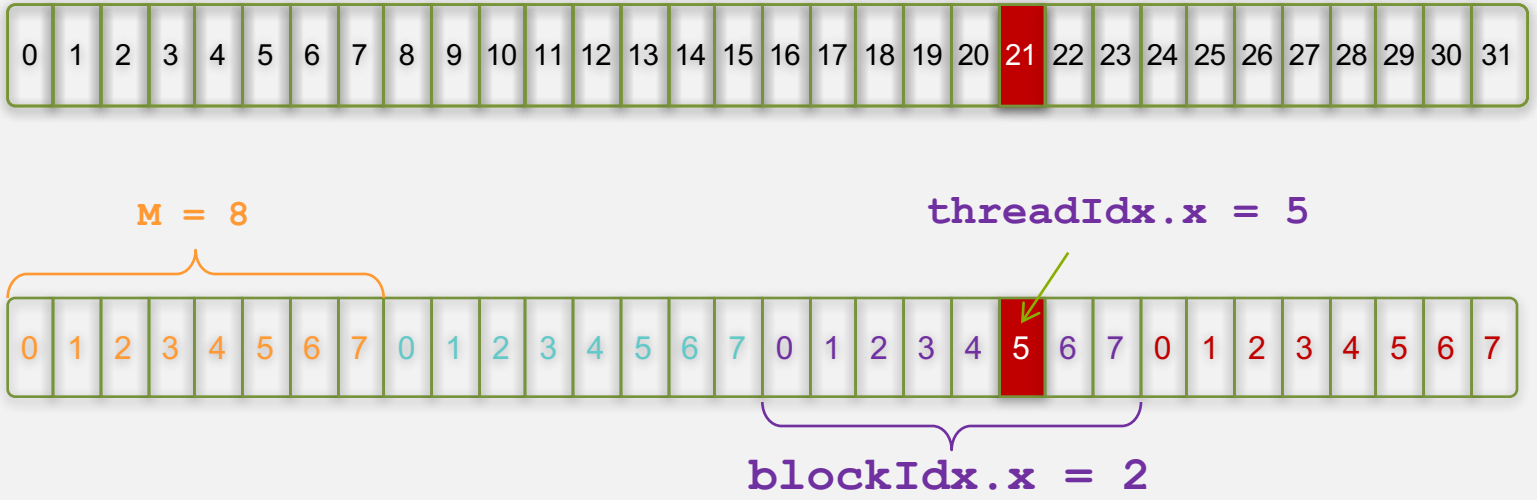


- With M threads/block a unique index for each thread is given by:

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

Indexing Arrays: Example

- Which thread will operate on the red element?




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

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];
}
```

- What changes need to be made in `main()`?

Addition with Blocks and Threads: `main()`

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
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 = N * sizeof(int);

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

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Addition with Blocks and Threads: `main()`

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

    // Launch add() kernel on GPU
    add<<<N/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

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

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Handling Arbitrary Vector Sizes

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

- 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];
}
```

- Update the kernel launch:

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

Review

- Launching parallel kernels
  - Launch `N` copies of `add()` with `add<<<N/M,M>>>(…);`
  - Use `blockIdx.x` to access block index
  - Use `threadIdx.x` to access thread index within block

- Allocate elements to threads:

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

Why Bother with Threads?

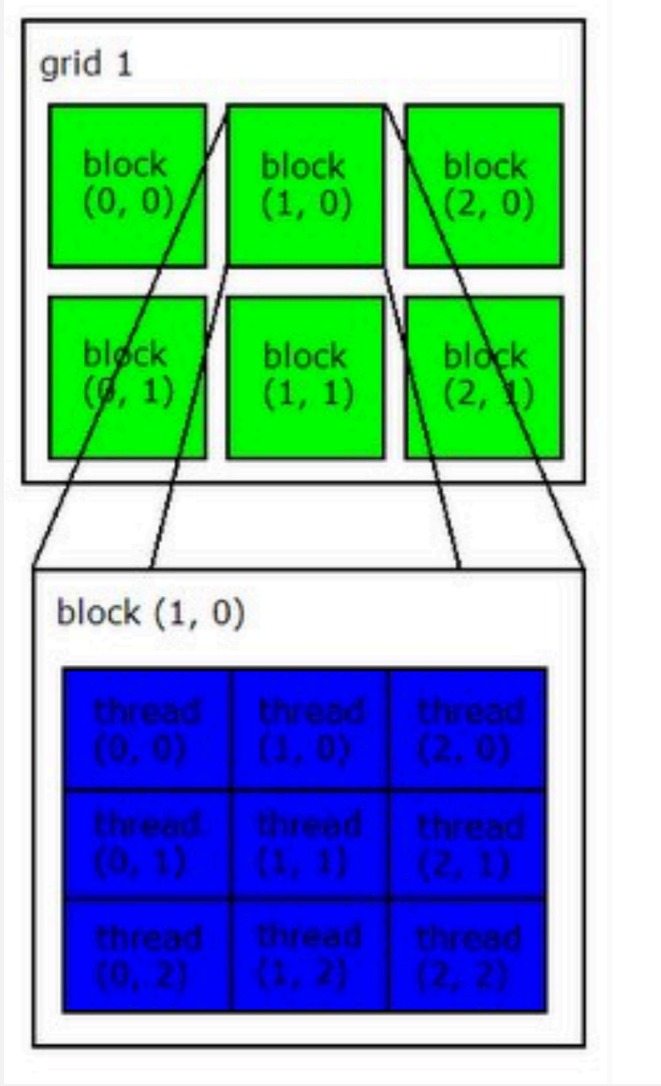
- Threads seem unnecessary
  - They add a level of complexity
  - What do we gain?

- Unlike parallel blocks, threads have mechanisms to:
  - Communicate
  - Synchronize

- To look closer, we need a new example…

Block and Thread in details

- 显示芯片执行时的最小单位是thread;
- 数个 thread 可以组成一个 block
- 一个 block 中的 thread 能存取同一块共享的内存，而且可以快速进行同步的动作。
- 每一个 block 所能包含的 thread 数目是有限的.
- 执行相同程序的 block，可以组成 grid;
- 不同 block 中的 thread 无法存取同一个共享的内存,无法直接互通或进行同步;
- 每个 thread 都有自己的一份register和 local memory空间。
- 同一个 block 中的每个 thread 则有共享的一份 share memory。
- 所有的 thread（包括不同block的thread）都共享一份 global memory、constant memory和 texture memory。

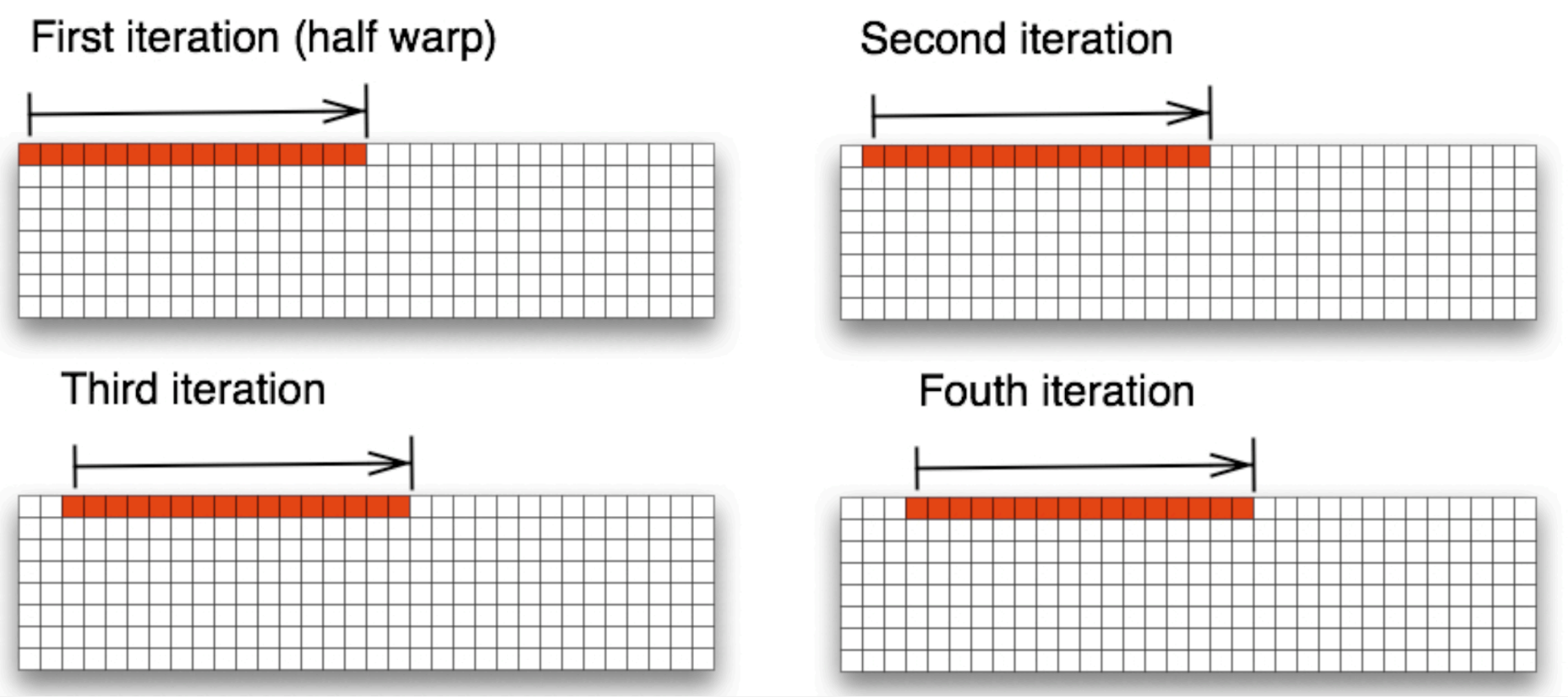


COOPERATING THREADS

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

GPU Stencil Operations



•Stencil : It is an Important Communication pattern Among Threads within a Block of a Grid, Basically it Allows to Reads Input From Fixed Neighborhood in a single location of an Array.

Stencil code

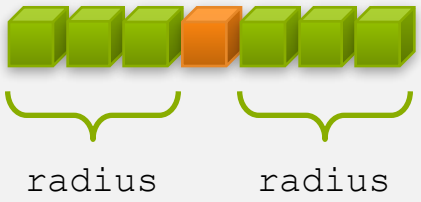
- Stencil codes are a class of iterative kernels which update array elements according to some fixed pattern, called a stencil.
- They are most commonly found in the codes of computer simulations,
  - computational fluid dynamics in the context of scientific and engineering applications；
  - solving partial differential equations,
  - the Jacobi kernel,
  - the Gauss–Seidel method
  - image processing and cellular automata.

https://en.wikipedia.org/wiki/Stencil_code

1D Stencil

- Consider applying a 1D stencil to a 1D array of elements
  - Each output element is the sum of input elements within a radius

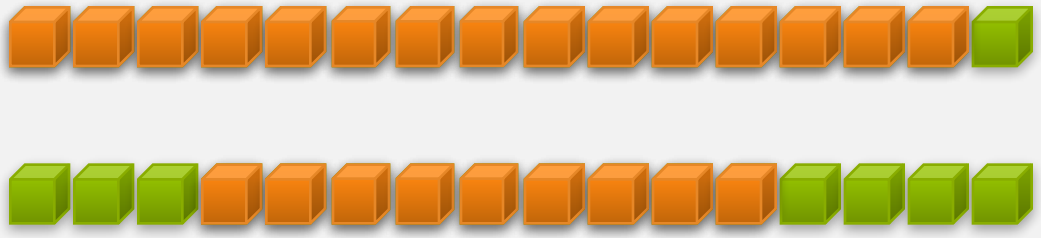
- If radius is 3, then each output element is the sum of 7 input elements, (the center element and its neighbors)



Implementing Within a Block

- Each thread processes one output element
  - blockDim.x elements per block

- Input elements read several times
  - With radius 3, each input element read seven times



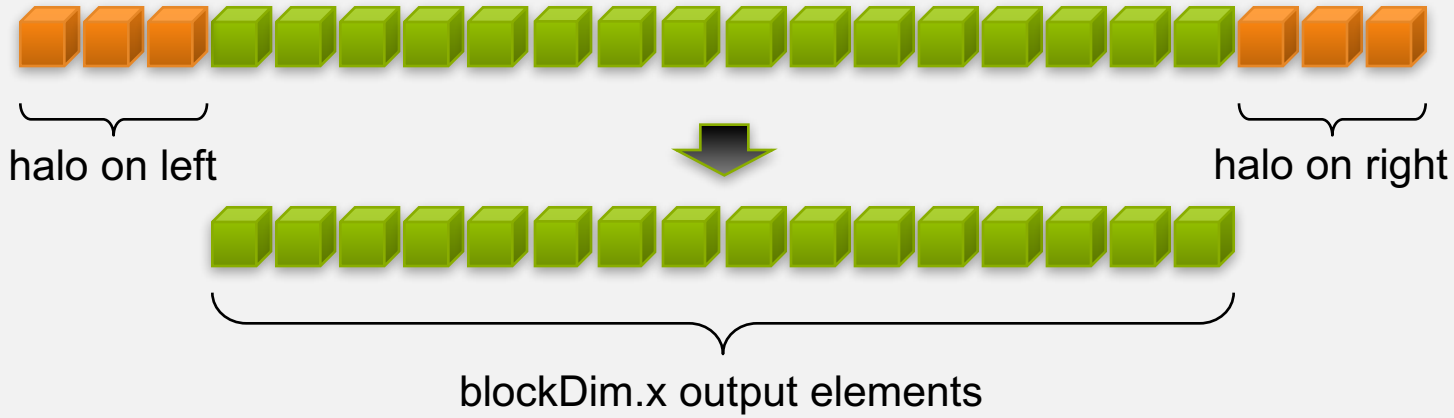
Sharing Data Between Threads

- Terminology: within a block, threads share data via shared memory
- Extremely fast on-chip memory, user-managed
- Declare using `__shared__`, allocated per block
- Data is not visible to threads in other blocks

Implementing With Shared Memory

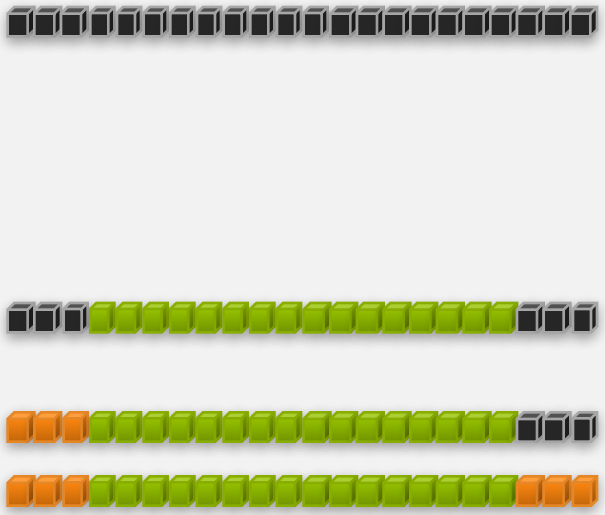
- Cache data in shared memory
  - Read (blockDim.x + 2 * radius) input elements from global memory to shared memory
  - Compute blockDim.x output elements
  - Write blockDim.x output elements to global memory

  - Each block needs a halo of radius elements at each boundary



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];
  }
```



Stencil Kernel

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

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

Data Race!

- The stencil example will not work…

- Suppose thread 15 reads the halo before thread 0 has fetched it…

```
temp[lindex] = in[gindex];          Store at temp[18]
if (threadIdx.x < RADIUS) {
  temp[lindex - RADIUS = in[gindex - RADIUS];     Skipped, threadIdx > RADIUS
  temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}

int result = 0;
result += temp[lindex + 1];         Load from temp[19]
```

__syncthreads()

- `void __syncthreads();`

- 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

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();
```

Stencil Kernel

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

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

Review (1 of 2)

- Launching parallel threads
  - Launch `N` blocks with `M` threads per block with `kernel<<<N,M>>>(…);`
  - Use `blockIdx.x` to access block index within grid
  - Use `threadIdx.x` to access thread index within block

- Allocate elements to threads:

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

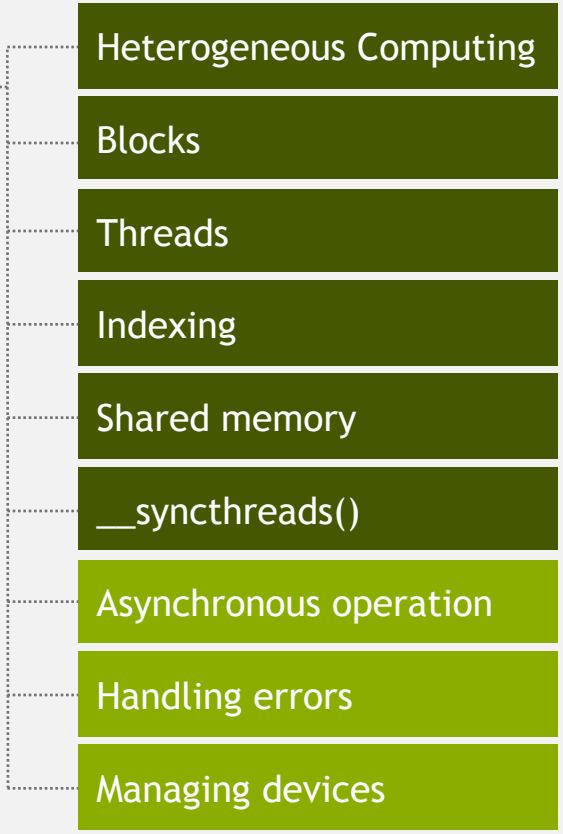
Review (2 of 2)

- Use `__shared__` to declare a variable/array in shared memory
  - Data is shared between threads in a block
  - Not visible to threads in other blocks

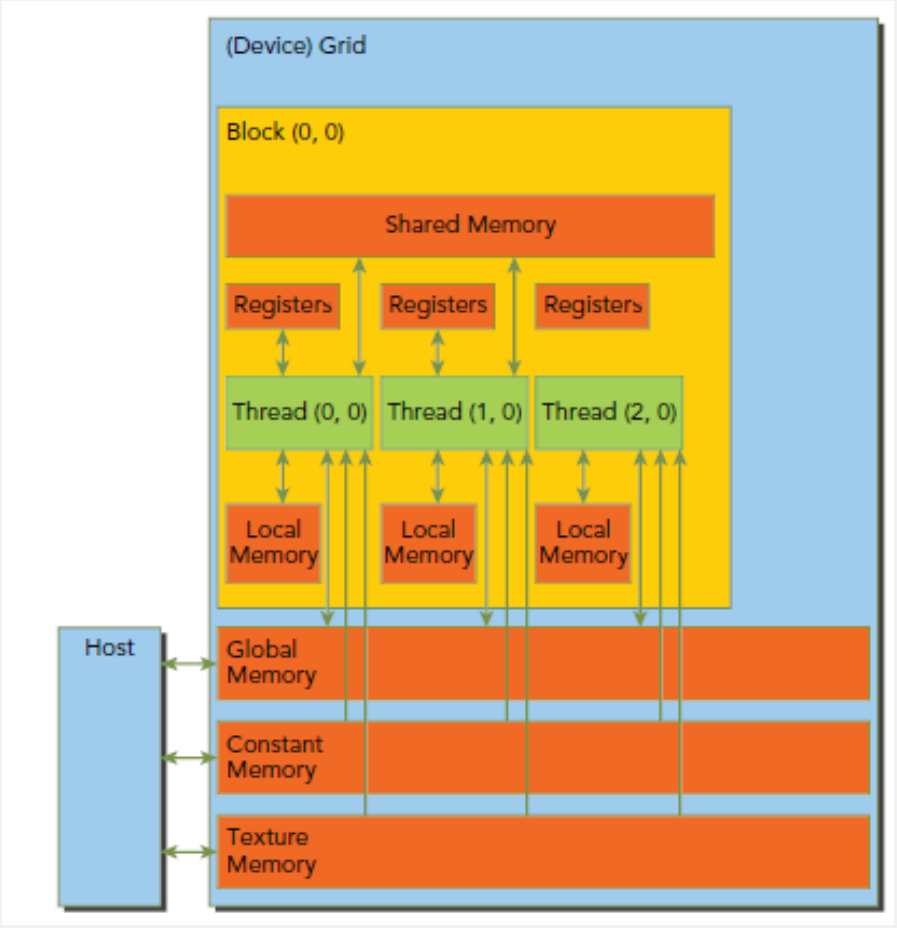
- Use `__syncthreads()` as a barrier
  - Use to prevent data hazards

MANAGING THE DEVICE

CONCEPTS



Memory of GPUs



memory的结构，他们各自都有不用的空间、生命期和cache。

memory可以分为下面两类：
Programmable：我们可以灵活操作的部分。
Non-programmable：不能操作，由一套自动机制来达到很好的性能。 在CPU的存储结构中，L1和L2 cache都是non-programmable的.CUDA来说，programmable的类型

- Registers
- Shared memory
- Local memory
- Constant memory
- Texture memory
- Global memory

Memory of GPUs

Registers:寄存器是GPU最快的memory，kernel中没有什么特殊声明的自动变量都是放在寄存器中的。当数组的索引是constant类型且在编译期能被确定的话，就是内置类型，数组也是放在寄存器中。

Shared Memory: 用__shared__修饰符修饰的变量存放在shared memory。因为shared memory是on-chip的，他相比localMemory和global memory来说，拥有高的多bandwidth和低很多的latency。他的使用和CPU的L1cache非常类似，但是他是programmable的。 shared memory是以block为单位分配的。

Local Memory: 如果register不够用了，那么就会使用local memory来代替这部分寄存器空间。除此外，下面几种情况，编译器可能会把变量放置在local memory：编译期无法决定确切值的本地数组。

local memory这个名字是有歧义的：在local memory中的变量本质上跟global memory在同一块存储区。所以，local memory有很高的latency和较低的bandwidth。在CC2.0以上，GPU针对local memory会有L1（per-SM）和L2（per-device）两级cache。

Global Memory: global Memory是空间最大，latency最高，GPU最基础的memory。“global”指明了其生命周期。任意SM都可以在整个程序的生命期中获取其状态。global中的变量既可以是静态也可以是动态声明。可以使用__device__修饰符来限定其属性

http://www.cnblogs.com/1024incn/p/4564726.html

Coordinating Host & Device

- Kernel launches are asynchronous
  - Control returns to the CPU immediately

- CPU needs to synchronize before consuming the results

`cudaMemcpy()`	Blocks the CPU until the copy is complete Copy begins when all preceding CUDA calls have completed
`cudaMemcpyAsync()`	Asynchronous, does not block the CPU
`cudaDeviceSynchronize()`	Blocks the CPU until all preceding CUDA calls have completed

Reporting Errors

- All CUDA API calls return an error code (`cudaError_t`)
  - Error in the API call itself

    OR
  - Error in an earlier asynchronous operation (e.g. kernel)

- Get the error code for the last error:

```
cudaError_t cudaGetLastError(void)
```

- Get a string to describe the error:

```
char *cudaGetErrorString(cudaError_t)
```

```
printf("%s\n", cudaGetErrorString(cudaGetLastError()));
```

Device Management

- Application can query and select GPUs

```
cudaGetDeviceCount(int *count)
cudaSetDevice(int device)
cudaGetDevice(int *device)
cudaGetDeviceProperties(cudaDeviceProp *prop, int device)
```

- Multiple threads can share a device

- A single thread can manage multiple devices

  `cudaSetDevice(i)` to select current device

  `cudaMemcpy(…)` for peer-to-peer copies †

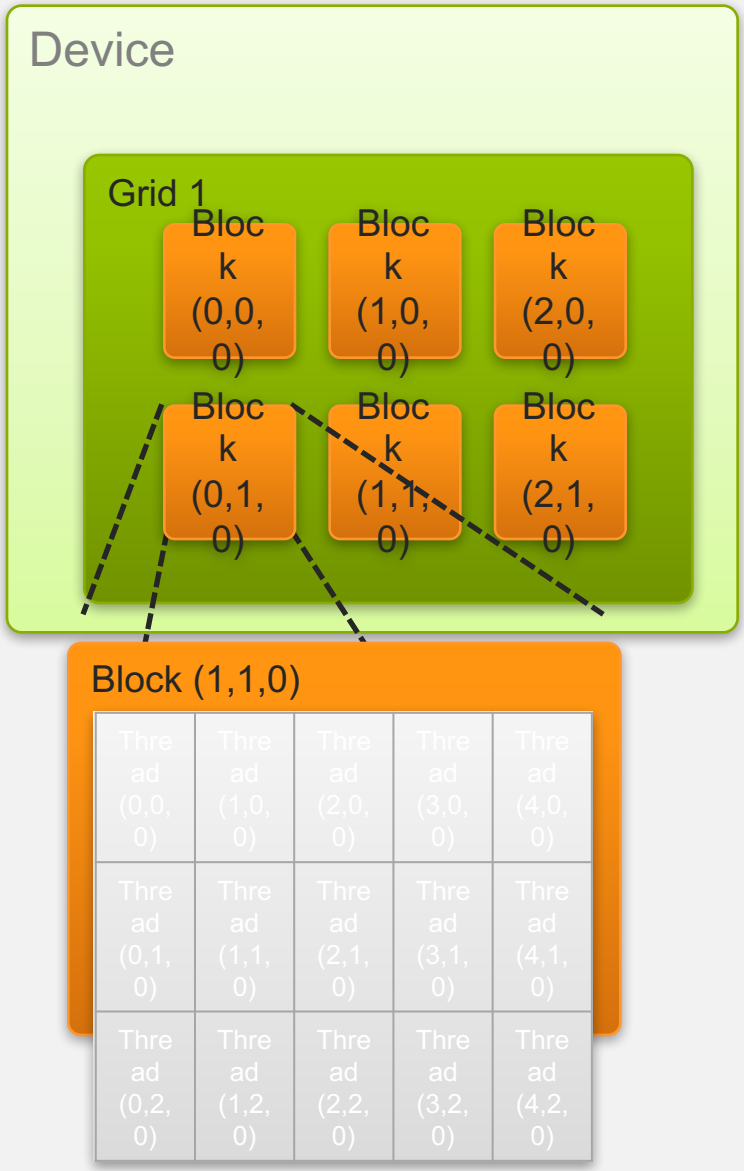
† requires OS and device support

Introduction to CUDA C/C++

- What have we learned?
  - Write and launch CUDA C/C++ kernels
    - `__global__, blockIdx.x, threadIdx.x, <<<>>>`
  - Manage GPU memory
    - `cudaMalloc(), cudaMemcpy(), cudaFree()`
  - Manage communication and synchronization
    - `__shared__, __syncthreads()`
    - `cudaMemcpy()` VS `cudaMemcpyAsync(), cudaDeviceSynchronize()`

IDs and Dimensions

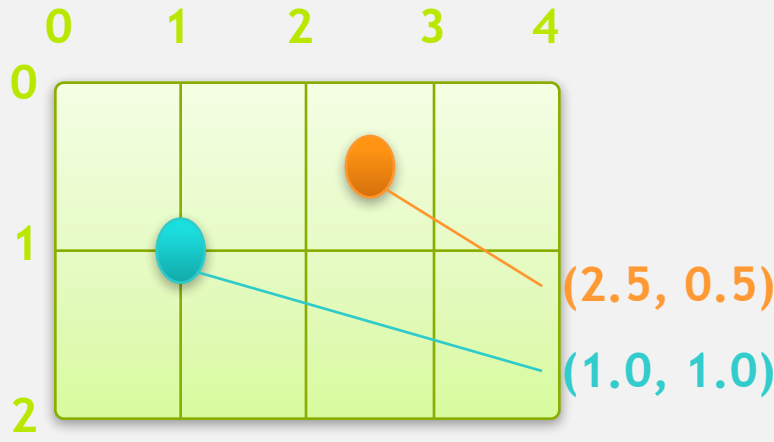
– A kernel is launched as a grid of blocks of threads
  - `blockIdx` and `threadIdx` are 3D
  - We showed only one dimension (`x`)



- Built-in variables:
  – `threadIdx`
  – `blockIdx`
  – `blockDim`
  – `gridDim`

Textures

- Read-only object
  - Dedicated cache
- Dedicated filtering hardware
  (Linear, bilinear, trilinear)
- Addressable as 1D, 2D or 3D
- Out-of-bounds address handling
  (Wrap, clamp)



Topics we skipped

- We skipped some details, you can learn more:
  - CUDA Programming Guide
  - CUDA Zone – tools, training, webinars and more

    developer.nvidia.com/cuda

- Need a quick primer for later:
  - Multi-dimensional indexing
  - Textures

- online CUDA documentation
- CUDA homepage
- CUDA Runtime API
- CUDA C Best Practices Guide
- CUDA Compiler Driver NVCC
- CUDA Visual Profiler
- CUDA-gdb debugger
- CUDA-memcheck memory checker
- CUBLAS library
- CUFFT library
- CUSPARSE library
- CURAND library
- NVIDIA webpage listing Compute Capability type of all GPUs
- PTX (low-level instructions)
- Nsight Visual Studio
- Nsight Eclipse
- Nsight Eclipse -- Getting Started
- Floating point accuracy on NVIDIA GPUs
- Kepler Tuning Guide
- Kepler Compatibility Guide
- Kepler Architecture White Paper
- Maxwell Tuning Guide
- Maxwell Compatibility Guide
- Maxwell GTX 750Ti White Paper
- Maxwell GTX 980 White Paper
- Pascal GTX 1080 White Paper
- Pascal P100 White Paper

- NVIDIA blog on Pascal GPU -- particularly good for NVlink discussion: 8-GPU example corresponds to NVIDIA's DGX-1 Deep Learning Server
- Tweaktown article on NVlink with IBM's new server based on Power 8 CPU
- French webpage with block diagrams of Pascal GPUs showing L1 cache size
- CUDA SDK examples
- helper_math.h header file defining operator-overloading operations for CUDA intrinsic vector datatypes such as float4 (also available in CUDA SDK in /usr/local/cuda/samples/common/inc)
- dbldbl.h header file defining double-double arithmetic for quad-precision (originally developed by NVIDIA, but not supported)
- Programming Massively Parallel Processors: a Hands-on Approach -- a book by David Kirk and Wen-mei Hwu
- CUDA by Example: An Introduction to General-Purpose GPU Programming -- a book by Jason Sanders and Edward Kandrot
- my hardware recommendations for those starting CUDA programming
- Wikipedia pages on NVIDIA Tesla, GeForce 900 and GeForce 10 GPUs
- Microway webpage on PCIe and GPU computing

APPENDIX

Compute Capability

- The **compute capability** of a device describes its architecture, e.g.
  - Number of registers
  - Sizes of memories
  - Features & capabilities

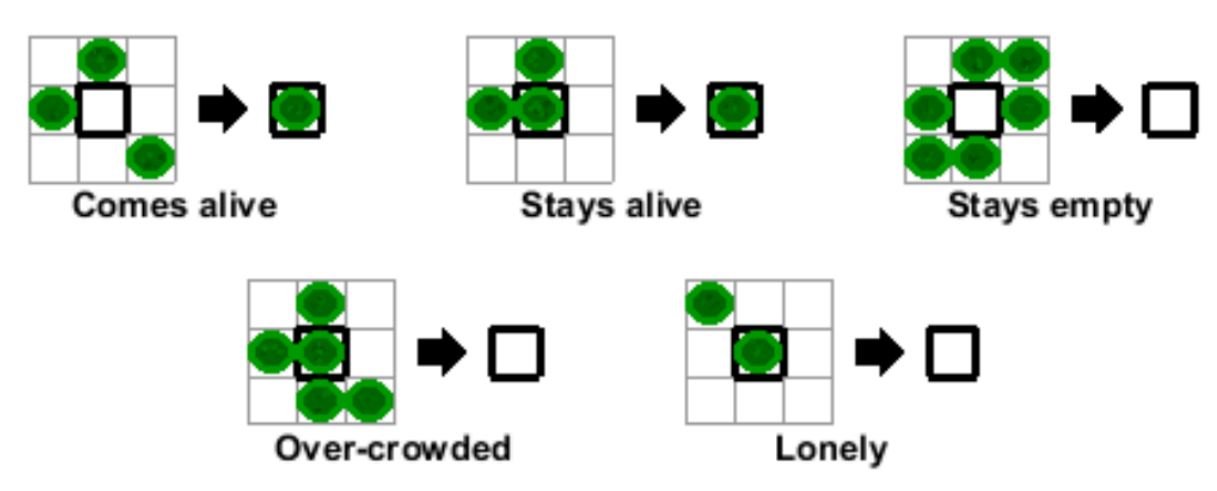
Compute Capability	Selected Features (see CUDA C Programming Guide for complete list)	Tesla models
1.0	Fundamental CUDA support	870
1.3	Double precision, improved memory accesses, atomics	10-series
2.0	Caches, fused multiply-add, 3D grids, surfaces, ECC, P2P, concurrent kernels/copies, function pointers, recursion	20-series

- The following presentations concentrate on Fermi devices
  - Compute Capability >= 2.0

Game of Life

The "Game of Life" follows a few simple rules
Cells are arranged in a 2D grid

- At each step, the fate of each cell is determined by the vitality of its eight nearest neighbors
- Any cell with exactly three live neighbors comes to life at the next step
- A live cell with exactly two live neighbors remains alive at the next step
- All other cells (including those with more than three neighbors) die at the next step or remain empty



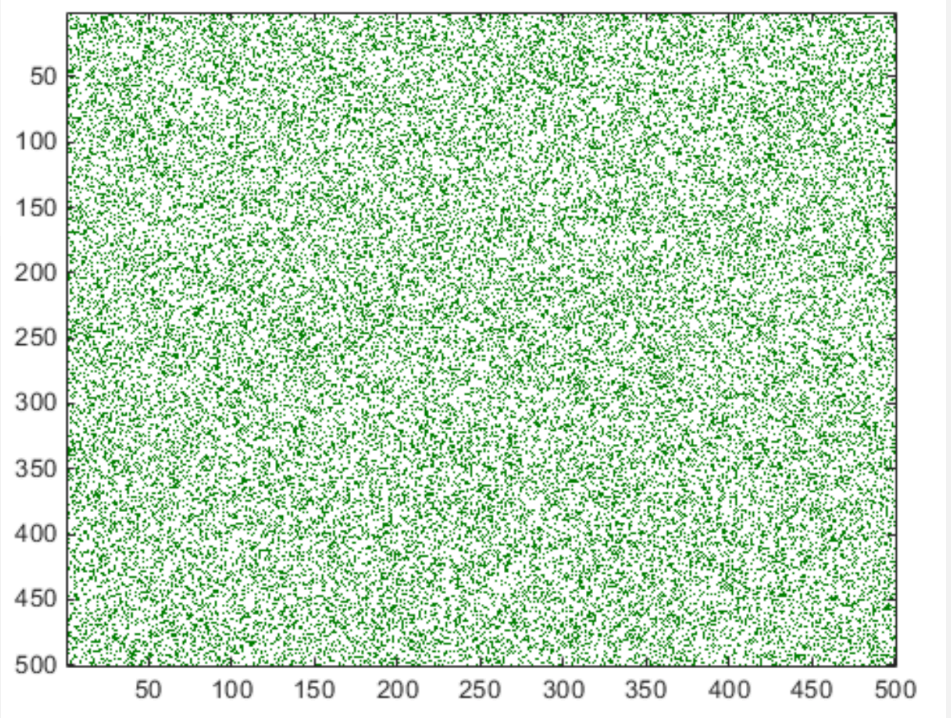
https://cn.mathworks.com/help/distcomp/examples/stencil-operations-on-a-gpu.html

Generate a random initial population

An initial population of cells is created on a 2D grid with roughly 25% of the locations alive.

```
gridSize = 500;
numGenerations = 100;
initialGrid = (rand(gridSize,gridSize) > .75);
gpu = gpuDevice();

% Draw the initial grid
hold off
imagesc(initialGrid);
colormap([1 1 1;0 0.5 0]);
title('Initial Grid');
```

paralleldemo_gpu_stencil()



```
    function X = updateGrid(X, N)
        p = [1 1:N-1];
        q = [2:N N];
        % Count how many of the eight neighbors are alive.
        neighbors = X(:,p) + X(:,q) + X(p,:) + X(q,:) + ...
            X(p,p) + X(q,q) + X(p,q) + X(q,p);
        % A live cell with two live neighbors, or any cell with
        % three live neighbors, is alive at the next step.
        X = (X & (neighbors == 2)) | (neighbors == 3);
    end

grid = initialGrid;
% Loop through each generation updating the grid and displaying it
for generation = 1:numGenerations
    grid = updateGrid(grid, gridSize);

    imagesc(grid);
    title(num2str(generation));
    drawnow;
end
```