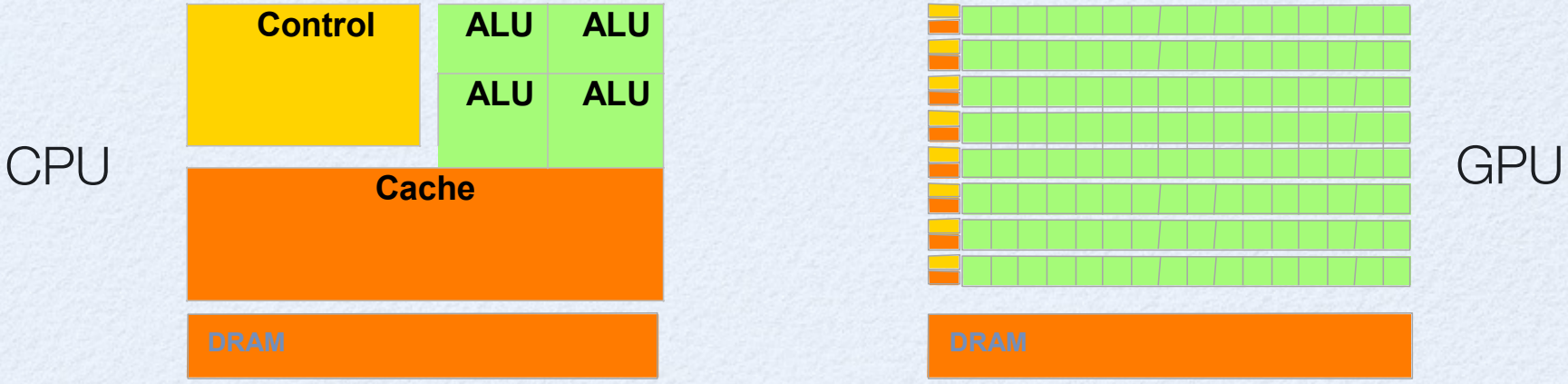
HPCSE II

GPU programming and CUDA

What is a GPU?

- Specialized for compute-intensive, highly-parallel computation, i.e. graphic output
- Evolution pushed by gaming industry
- CPU: large die area for control and caches
- GPU: large die area for data processing



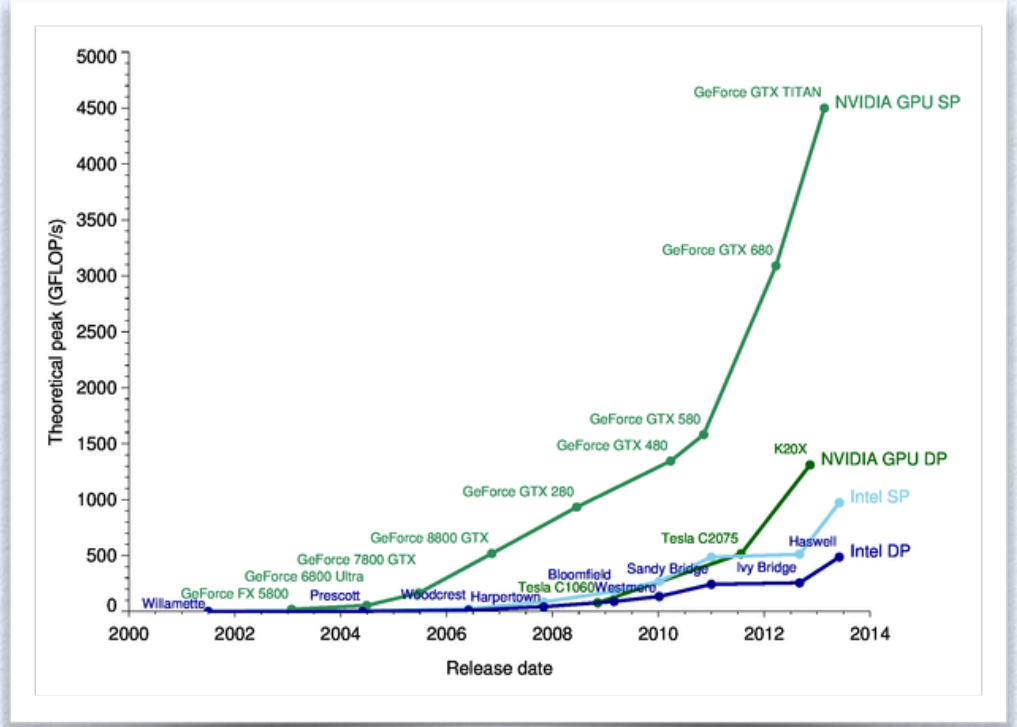
picture source: Nvidia

GPGPUs

- Graphics Processing Unit (GPU): Hardware designed for output to display
- General Purpose computing on GPUs (GPGPU) used for non-graphics tasks
  - physics simulation
  - signal processing
  - computational geometry
  - computer vision
  - computational biology
  - computational finance
  - meteorology

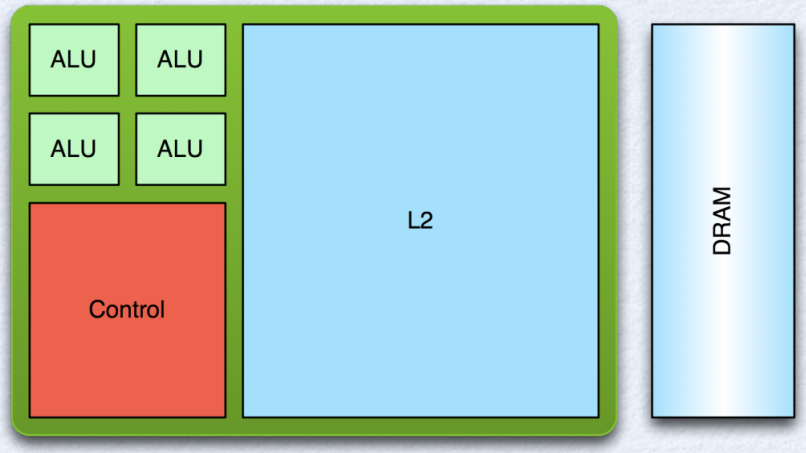
Why GPUs?

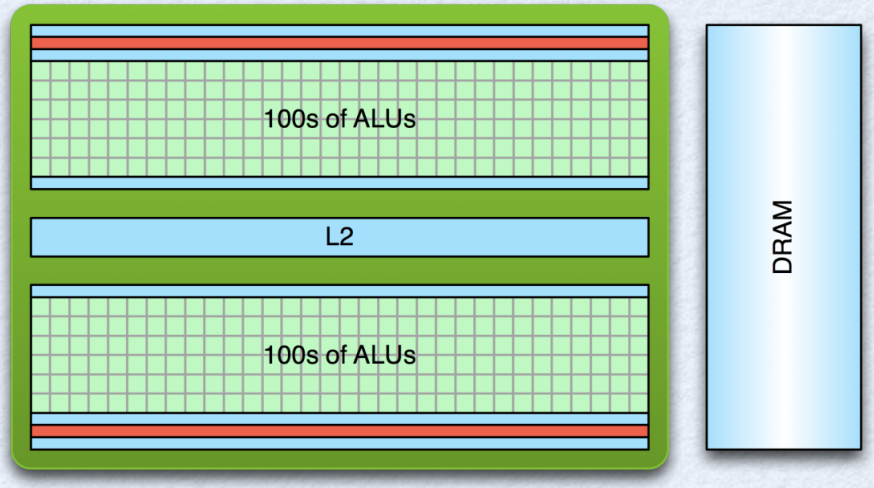
- GPU evolved into a very flexible and powerful processor
- It's programmable using high-level languages
- It offers more GFLOP/s and more GB/s than CPUs



picture source: http://hemprasad.wordpress.com/category/computing-technology/parallel/gpu-cuda/

Low Latency or High Throughput?





CPU

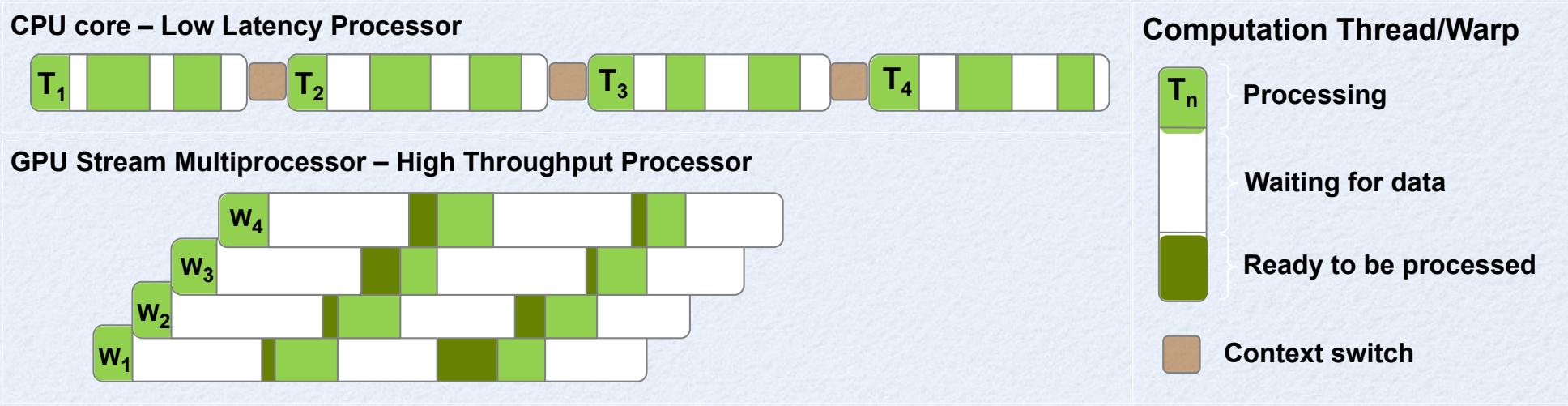
- Optimized for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution

GPU

- Optimized for data-parallel, throughput computation
- Tolerant of memory latency

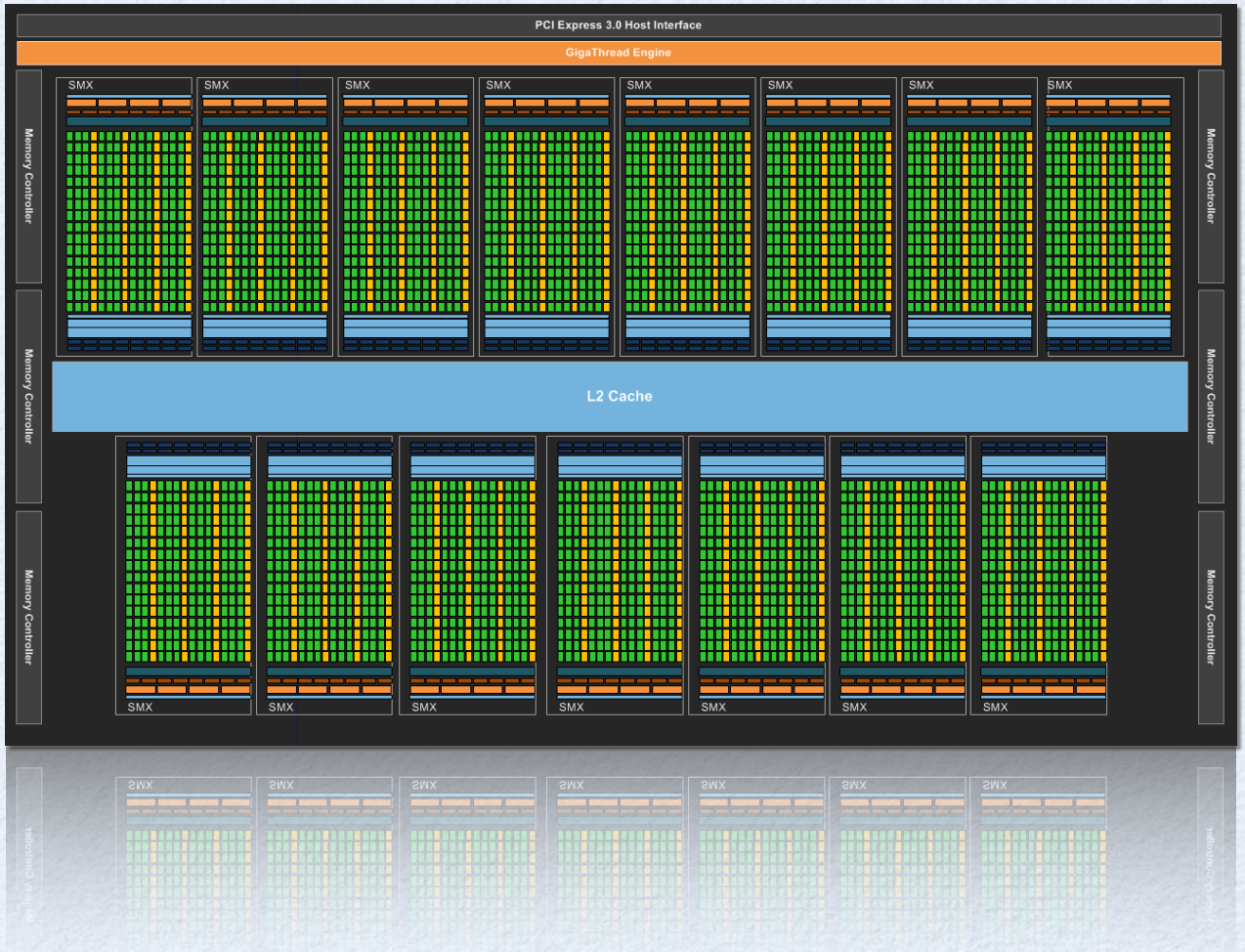
Low Latency or High Throughput?

- CPU minimized latency within each thread
- GPU hides latency with computation from other thread warps



Kepler GK110 Block Diagram

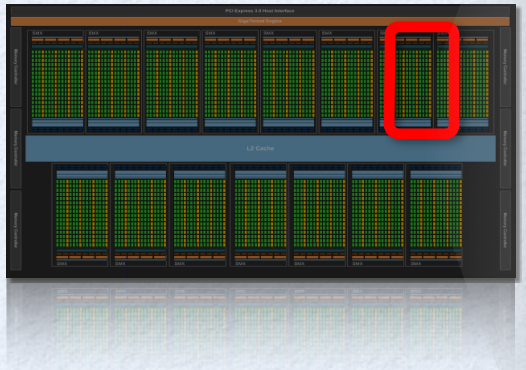
- 7.1B Transistors
- > 1 TFLOP FP64
- 1.5 MB L2 Cache
- 15 SMX units



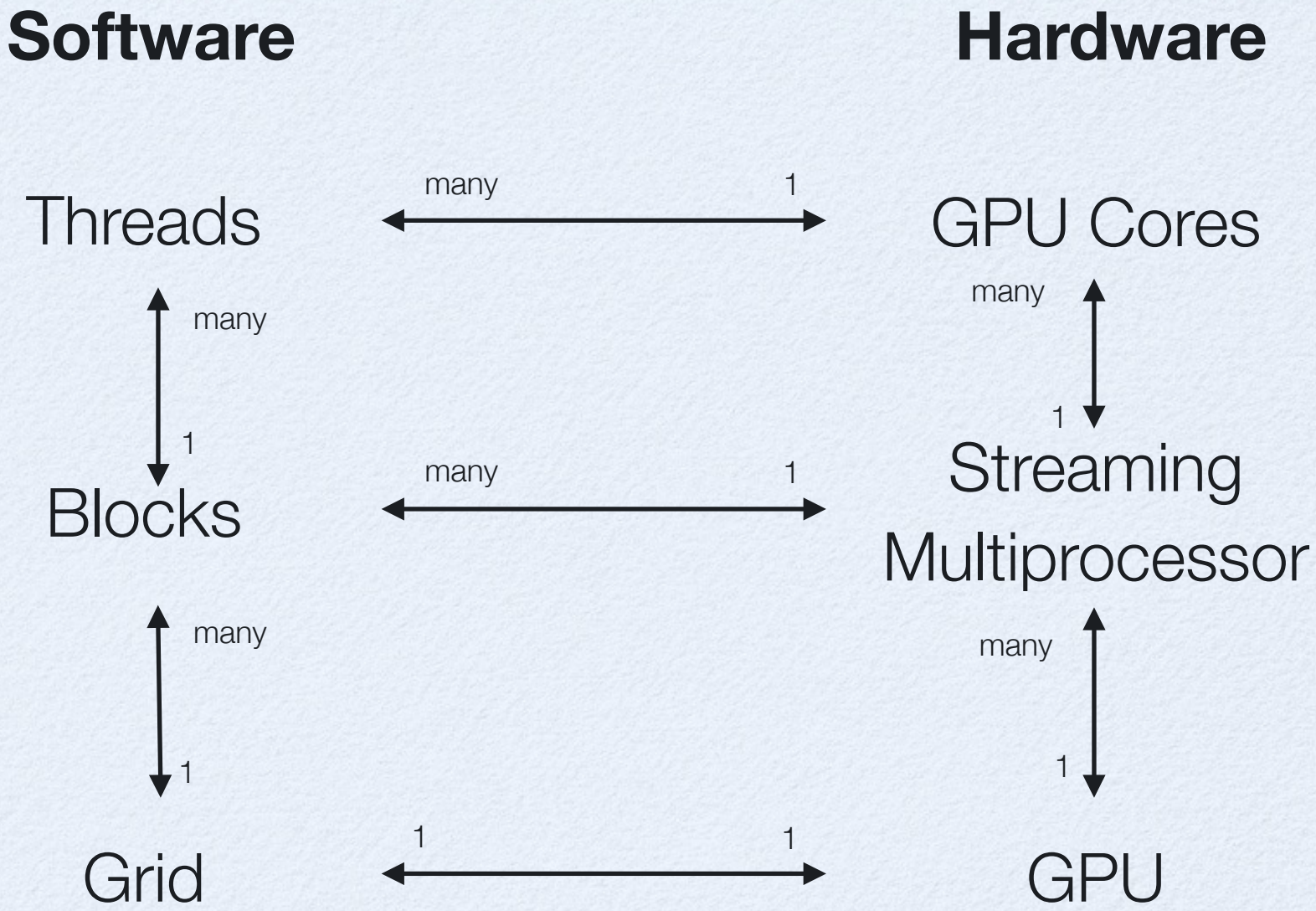
contains 15 streaming multiprocessors (SMX units)

GK110 Streaming Multiprocessor (SMX)

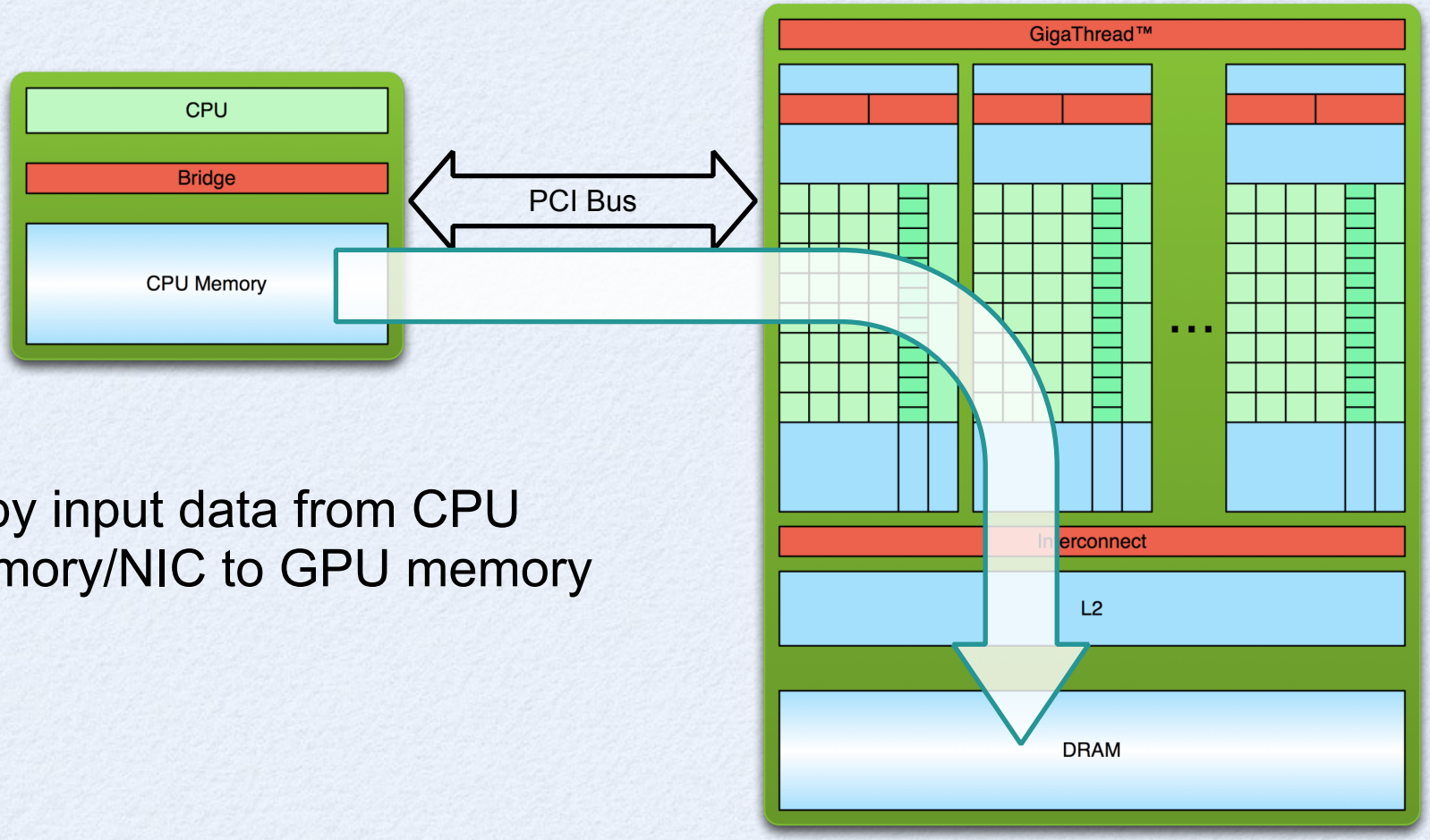
- Contains many specialized cores
- Up to 2048 threads concurrently
  - 192 fp32 ops/clock
  - 64 fp64 ops/clock
  - 160 int32 ops/clock
- 48KB shared mem
- 64K 32-bit registers



GPU Parallelism

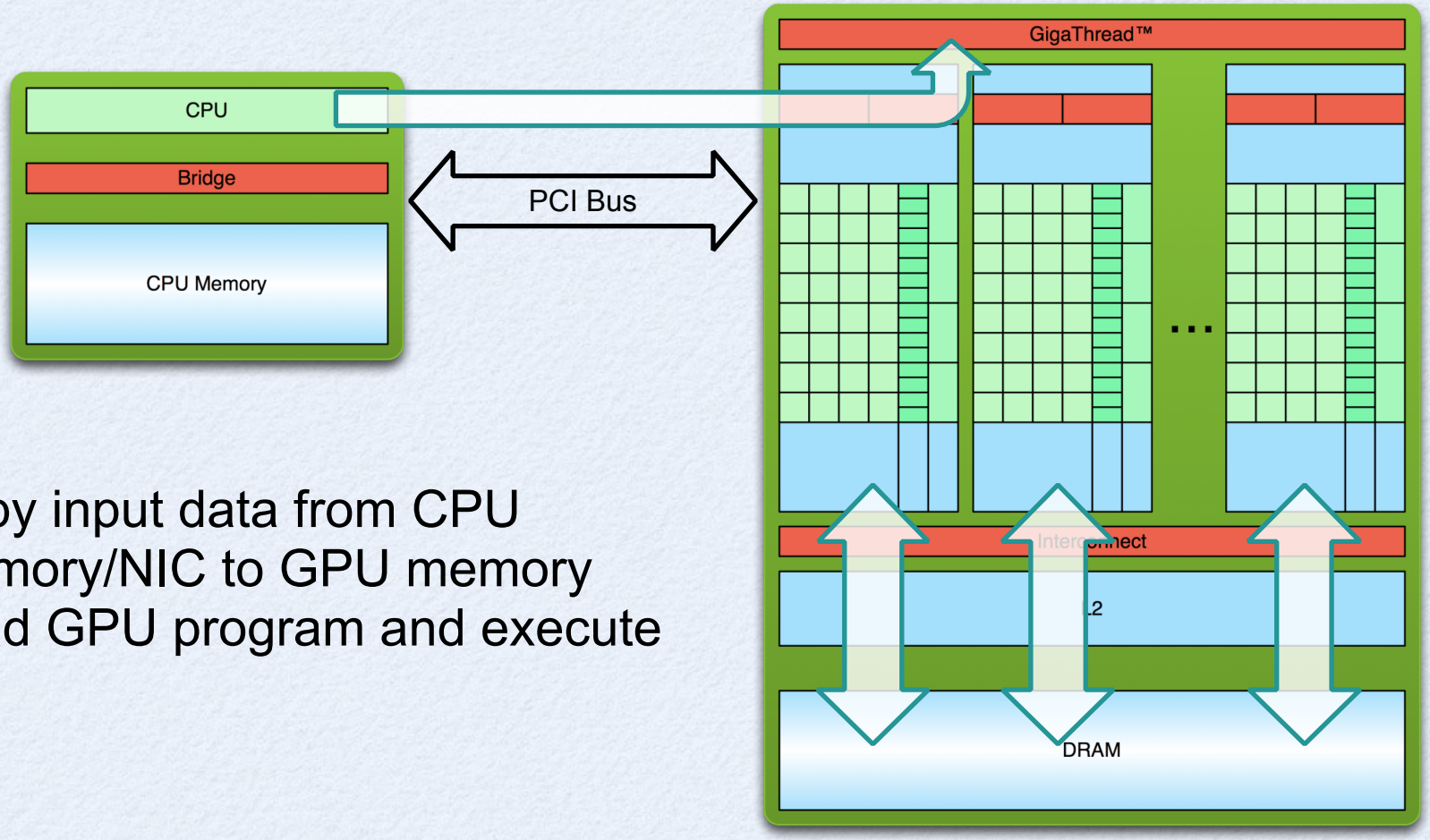


Simple Processing Flow



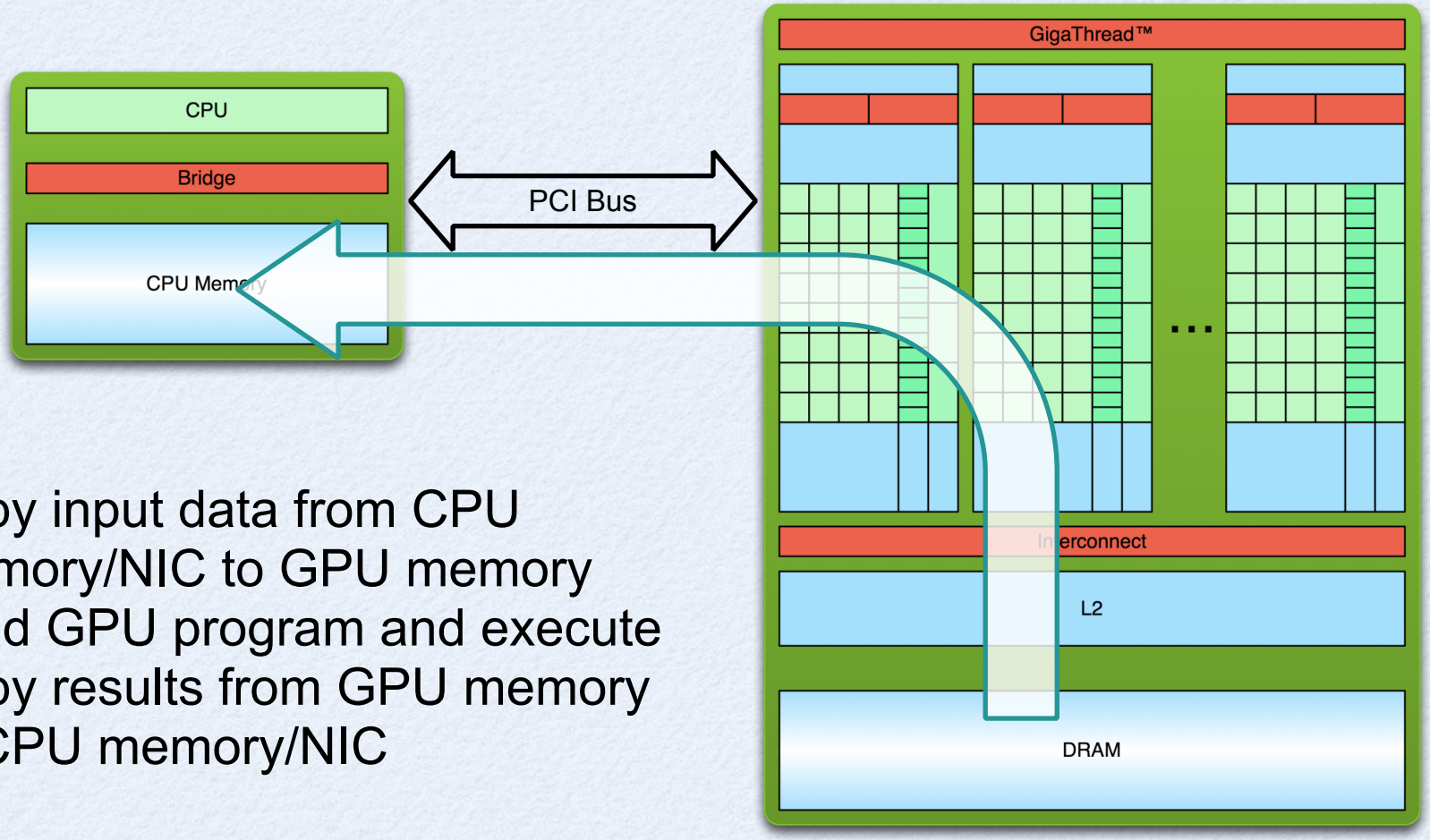
1. Copy input data from CPU memory/NIC to GPU memory

Simple Processing Flow



1. Copy input data from CPU memory/NIC to GPU memory
2. Load GPU program and execute

Simple Processing Flow



1. Copy input data from CPU memory/NIC to GPU memory
2. Load GPU program and execute
3. Copy results from GPU memory to CPU memory/NIC

3 Ways to Accelerate Applications

"Drop-in" Acceleration

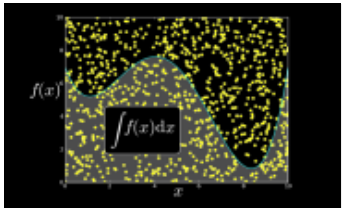
Easily Accelerate Applications

Similar to OpenMP

Maximum Flexibility

GPU accelerated libraries

NVIDIA cuBLAS



NVIDIA cuRAND

NVIDIA cuSPARSE

NVIDIA NPP

GPU VSIPL

IMSL Library
Vector Signal
Image Processing

CULA tools

GPU Accelerated
Linear Algebra

MAGMA

Matrix Algebra on
GPU and Multicore

NVIDIA cuFFT

ROGUE WAVE SOFTWARE

CUSP

Sparse Linear
Algebra

libjacket

Building-block
Algorithms for CUDA

Thrust

C++ STL Features
for CUDA

CUDA Math Libraries

- cuFFT – Fast Fourier Transforms Library
- cuBLAS – Complete BLAS Library
- cuSPARSE – Sparse Matrix Library
- cuRAND – Random Number Generation (RNG) Library
- NPP – Performance Primitives for Image & Video Processing
- Thrust – Templated C++ Parallel Algorithms & Data Structures
- math.h - C99 floating-point Library

Included in the free CUDA Toolkit

Programming for GPUs

- **Early days:**
  - OpenGL (graphics API)

- **Now**
  - **CUDA: Nvidia proprietary API, works only on Nvidia GPUs**
  - OpenCL: open standard for heterogeneous computing
  - OpenACC: open standard based on compiler directives

CUDA Example: Code

main.cu

```
#include <iostream>
#include <cassert>

#define N 32768

__global__ void scaleVector(float scale, float * input, float * output)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x;

    if (tid < N)
    {
        output[tid] = input[tid] * scale;
    }
}

int main()
{
    float * a = new float[N];
    float * b = new float[N];

    float * dev_a;
    float * dev_b;

    const float scale = 2.;

    for (int i=0; i<N; i++)
        a[i] = (float)i/2.;

    std::cout << "Initializing data on GPU\n";

    cudaMalloc( (void**)&dev_a, N*sizeof(float) );
    cudaMalloc( (void**)&dev_b, N*sizeof(float) );

    cudaMemcpy( dev_a, a, N*sizeof(float), cudaMemcpyHostToDevice);


    std::cout << "Launching kernels on GPU\n";

    const int nblocks = 128;
    const int nthreads = 256;
    scaleVector<<< nblocks, nthreads >>>(scale, dev_a, dev_b);

    std::cout << "Downloading data\n";

    cudaMemcpy( b, dev_b, N*sizeof(float), cudaMemcpyDeviceToHost);

    std::cout << "Verifying results\n";

    for (int i=0; i<N; i++)
    {
        std::cout << b[i] << std::endl;
        assert((double)i == b[i]);
    }

    std::cout << "Done!\n";

    delete [] a;
    delete [] b;
}
```

device code (runs on CPU)

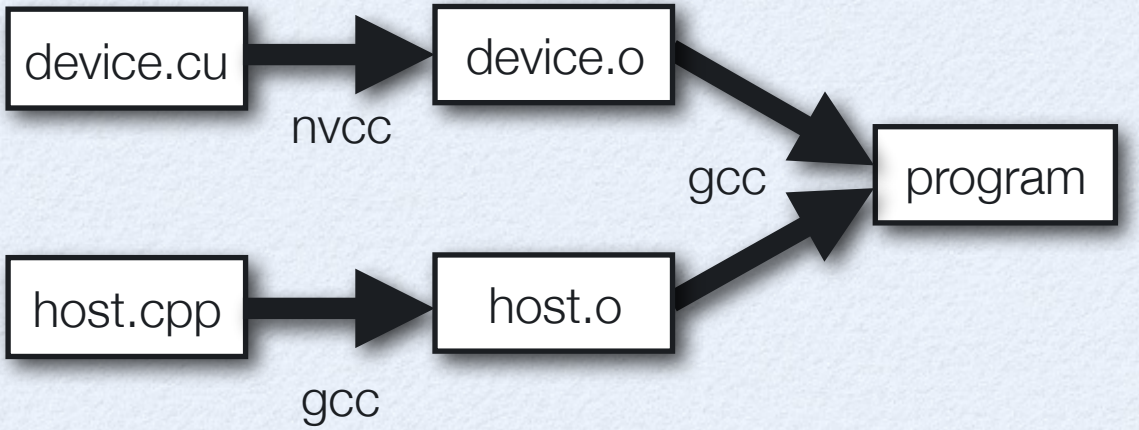
```
__global__ void scaleVector(float scale, float * input, float * output)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x;

    if (tid < N)
    {
        output[tid] = input[tid] * scale;
    }
}
```

host code (runs on CPU)

```
std::cout << "Initializing data on GPU\n";

cudaMalloc( (void**)&dev_a, N*sizeof(float) );
cudaMalloc( (void**)&dev_b, N*sizeof(float) );

cudaMemcpy( dev_a, a, N*sizeof(float), cudaMemcpyHostToDevice);

std::cout << "Launching kernels on GPU\n";

const int nblocks = 128;
const int nthreads = 256;
scaleVector<<< nblocks, nthreads >>>(scale, dev_a, dev_b);

std::cout << "Downloading data\n";

cudaMemcpy( b, dev_b, N*sizeof(float), cudaMemcpyDeviceToHost);
```

Nvidia CUDA

- C extension to write GPU code
- Only supported by Nvidia GPUs
- Code compilation (nvcc) and linking:



device.cu

```
__global__ void kernel()
{
    // do something
}
```

host.cpp

```
int main()
{
}
```

Hello World

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

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…

Hello World! with Device Code

```
mykernel<<<gridDim, blockDim>>>(…);
```

- Triple angle brackets mark a call from *host* code to *device* code, also called a “kernel launch”
- **gridDim** is the number of instances of the kernel
- **blockDim** is the number of threads within each instance
- gridDim and blockDim may be 2D or 3D vectors (type **vec3**) to simplify application programs

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

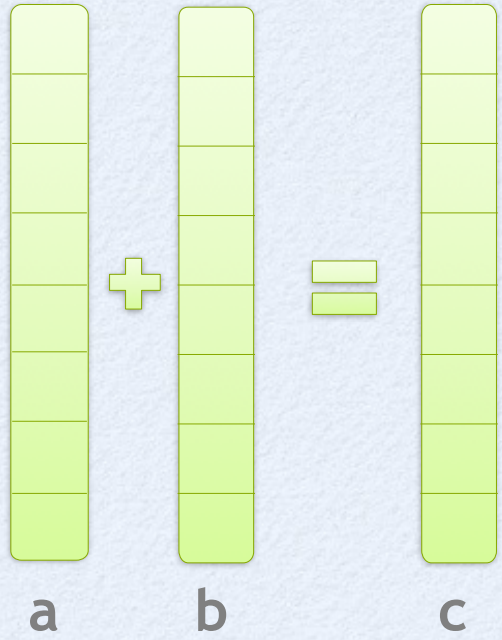
- ***`nvcc`*** separates source code into host and device components
  - Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
  - Host functions (e.g. `main()`) processed by standard host compiler

GPU Kernel qualifiers

- Function qualifiers:
  - **__global__**: called from CPU, runs on GPU
  - **__device__**: called from GPU, runs on GPU
  - **__host__**: called from CPU, runs on CPU
  - **__host__** and **__device__** can be combined

Parallel Programming in CUDA C/C++

- We need a more interesting example...
- GPU computing is about massive parallelism!
- 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

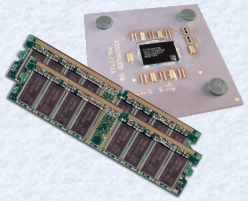
    May be passed to/from host code

    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()`

Data Transfer

```
cudaMemcpy(void* dst, void* src, size_t num_bytes,
           enum cudaMemcpyKind direction)
```

- direction can be either of
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice

Addition on the Device: `main()`

```
int main(void) {
  // host copies of a, b, c
  int a, b, c;

  // device copies of a, b, c

  int *d_a, *d_b, *d_c;

  int size = sizeof(int);

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

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

   // 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;
}
```

Going parallel

Moving to Parallel

- GPU computing is about massive parallelism
- 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

- 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 element of the array

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:

Block 0

```
c[0]  = a[0] + b[0];
```

Block 1

```
c[1]  = a[1] + b[1];
```

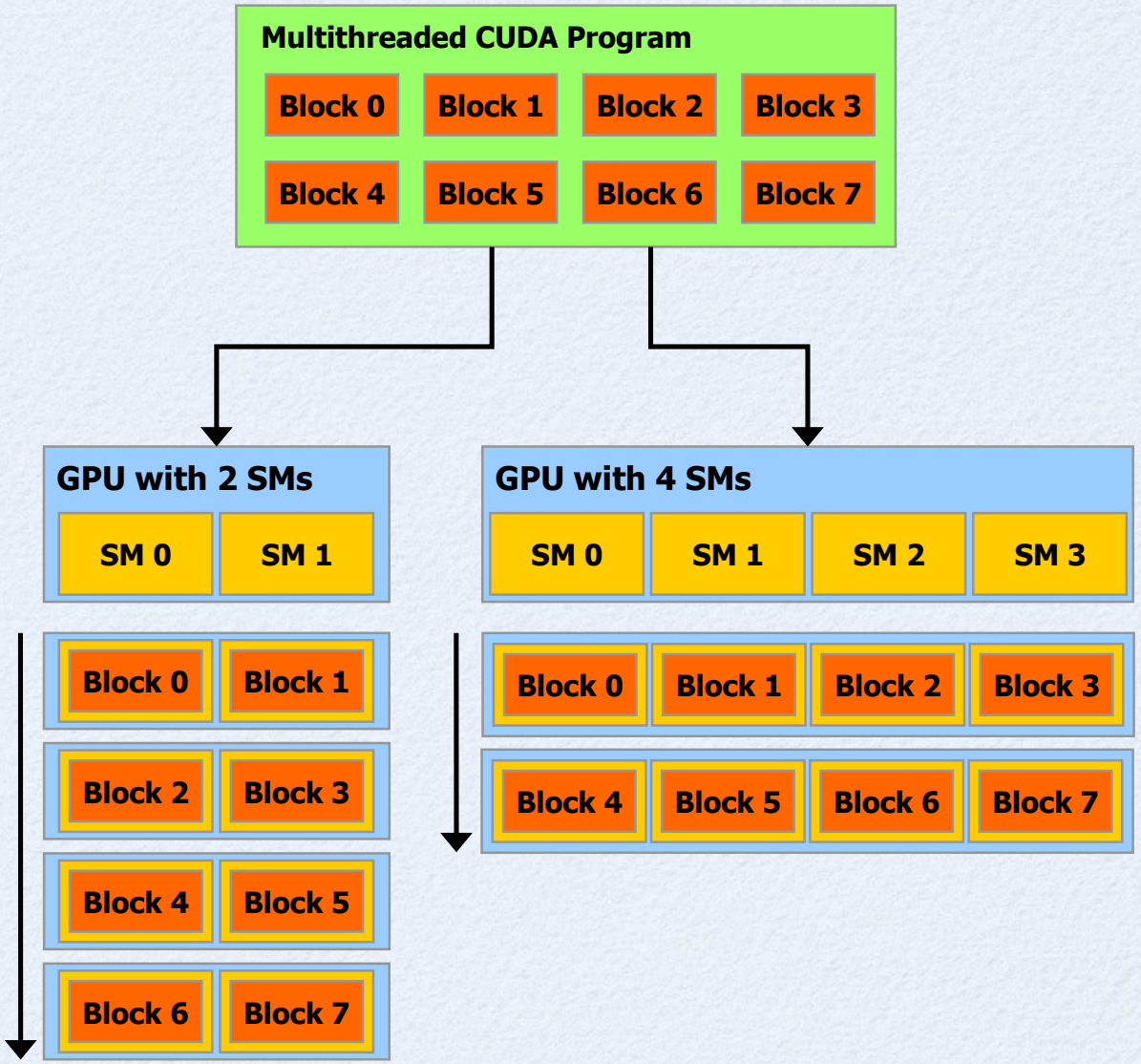
Block 2

```
c[2]  = a[2] + b[2];
```

Block 3

```
c[3]  = a[3] + b[3];
```

Block Scheduling



Special variables

- Some variables are predefined
  - `gridDim` size (or dimensions) of grid of blocks
  - `blockIdx` index (or 2D/3D indices) of block
  - `blockDim` size (or dimensions) of each block
  - `threadIdx` index (or 2D/3D indices) of thread

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;
}
```

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()`…

One change in 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;
}
```

One change in 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;
}
```

Vector addition

add <<< B,T>>> ( )

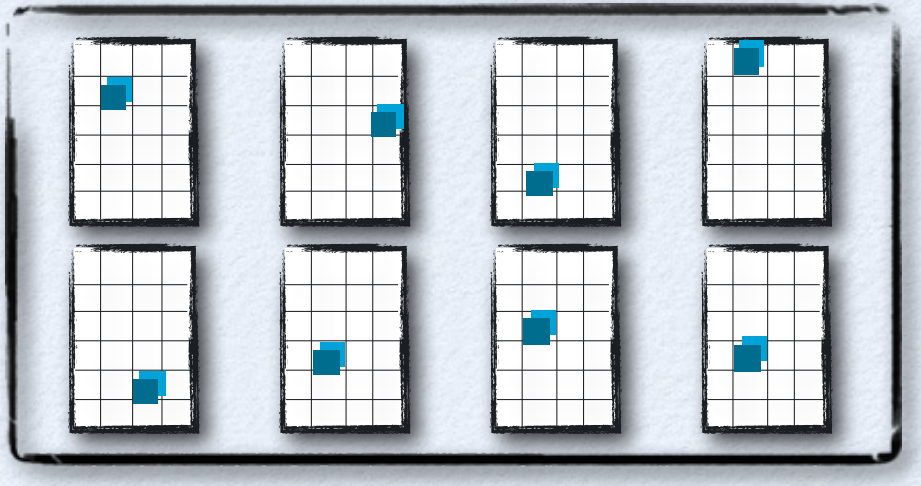
B: Number of Blocks

T: Number of threads/ Block

add <<<**N,1**>>> ( )

B=N=16

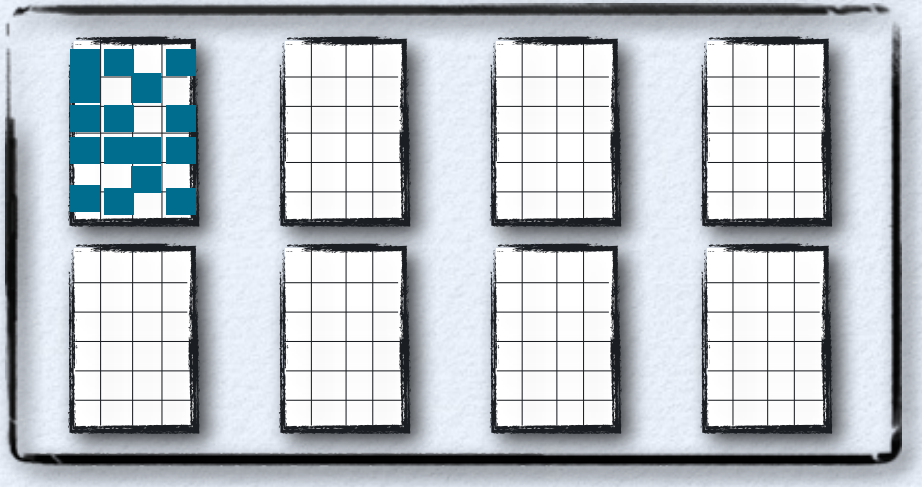
T=1



add <<<**1,N**>>> ( )

T=N=16

B=1



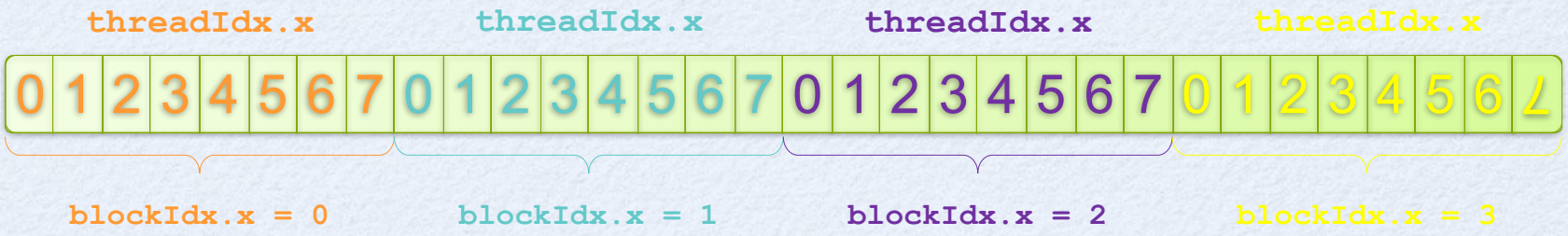
Combining Blocks and Threads

- We've seen parallel vector addition using:
  - Several blocks with one thread each
  - One block with several threads
- Let's adapt vector addition to use both ***blocks*** and ***threads***
- Why? We'll come to that...
- First let's discuss data indexing...

Indexing 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 per block, a unique index for each thread is given by:

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



Using Blocks and Threads

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

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

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

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

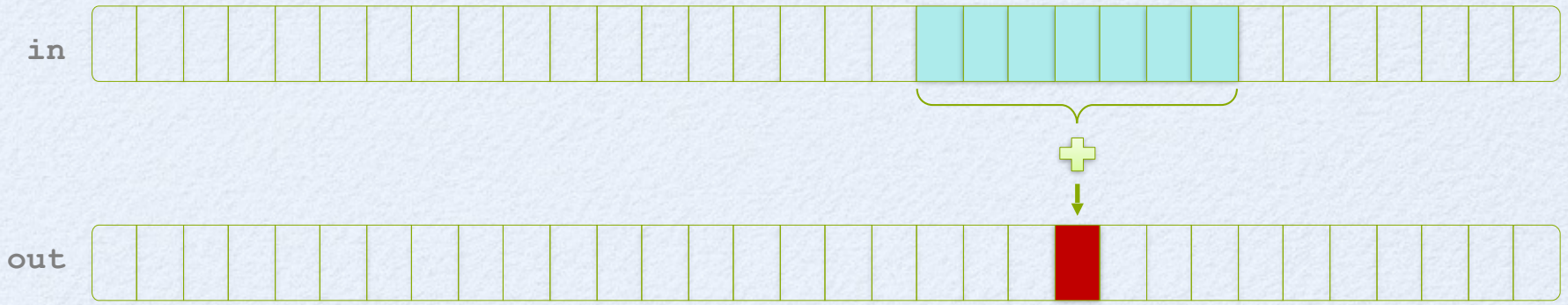
Why Bother with Threads?

- Threads seem unnecessary
  - They add a level of complexity
  - What do we gain?
- Unlike parallel blocks, threads have mechanisms to efficiently:
  - Communicate
  - Synchronize
- To look closer, we need a new example...

Using stencils

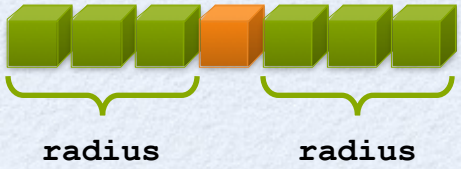
1D Stencil

- 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:



Shared memory within a block

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

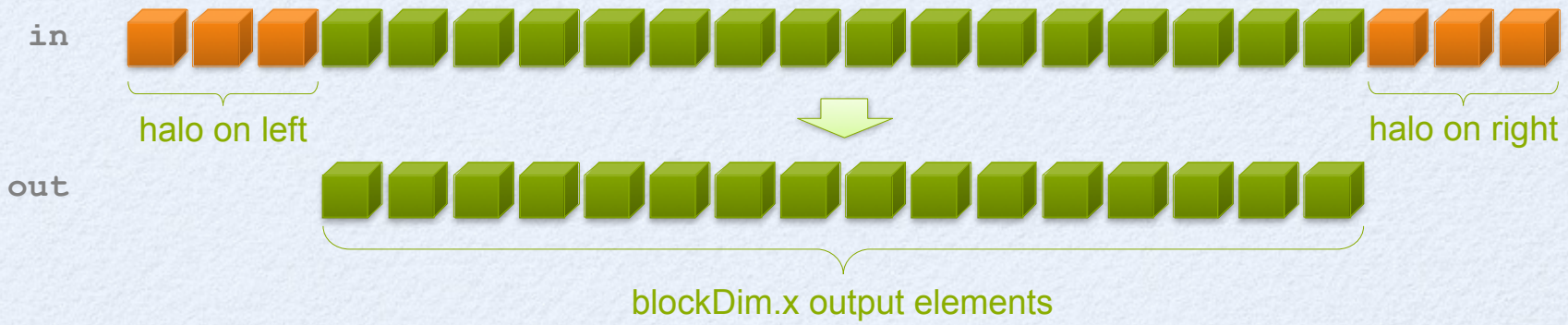


- Input elements are read several times
  - With radius 3, each input element is read seven times

- Within a block, threads can share data via shared memory
  - Extremely fast on-chip memory, but very small
  - Like a user-managed cache
  - Declare using __shared__, allocated per block
  - Data is not visible to threads in other blocks

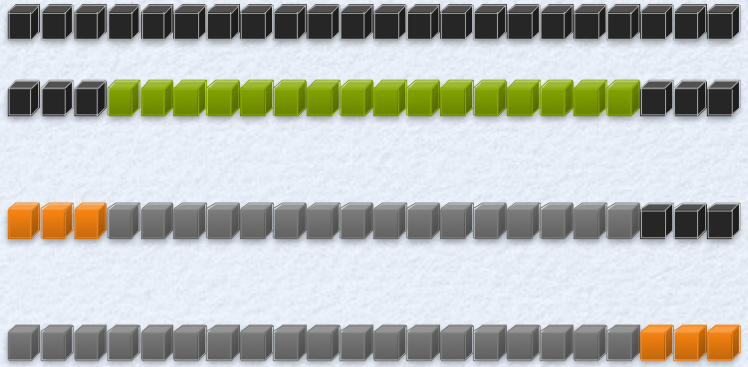
Implementing With Shared Memory

- 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 (1 of 2)

```
__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];
    }

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

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



Race condition!

__syncthreads()

- `void __syncthreads();`

- Synchronizes all threads within a block
- All threads must reach the barrier
  - In conditional code (if statements),
    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];
    }

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

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

Race condition!

Stencil Kernel

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

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

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

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

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

Review

- 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
- Assign elements to threads:

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

- 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 to avoid race conditions

Managing the Device

Coordinating Host & Device

- Kernel launches are asynchronous
- 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`)
- Get the error code for the last error:

```
cudaError_t cudaGetLastError()
```

- Get a string to describe the error:

```
char *cudaGetErrorString(cudaError_t)
if(cudaGetLastError() != cudaSuccess)
    std::cerr << 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 host threads can share a device

- A single host thread can manage multiple devices

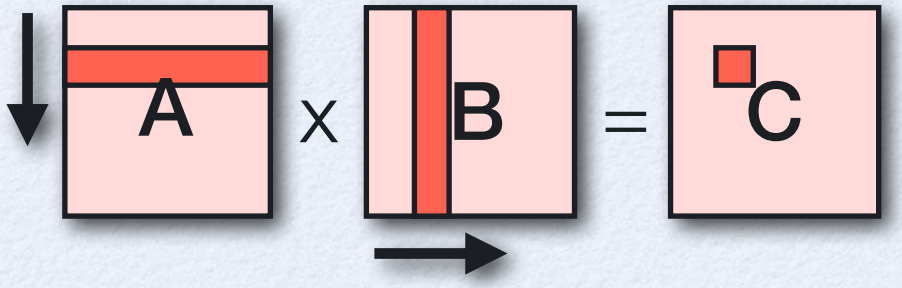
  `cudaSetDevice(i)` to select current device

  `cudaMemcpy(...)` for peer-to-peer copies

Multiplying matrices

GPU GEMM

Matrix-Matrix Multiplication




```
for (int i=0; i<N; i++)
  for (int j=0; j<N; j++)
    for (int k=0; k<N; k++)
      c[i*N+j] += a[i*N+k] * b[k*N+j];
```

Parallelize

Kernel

A matrix multiplication kernel

- Let us use a 2D grid of blocks and threads

```
__global__ void matrix(float * a, float * b, float * c, int N)
{
  int ix = threadIdx.x + blockIdx.x*blockDim.x;
  int iy = threadIdx.y + blockIdx.y*blockDim.y;

  if (ix<N && iy<N)
  {
    c[ix*N + iy] = 0;

    for (int k=0; k<N; k++)
      c[ix*N + iy] += a[ix*N + k] * b[k*N + iy];
  }
}
```

can you optimize it using shared memory?

A matrix multiplication kernel

- And call it by creating a 2D grid

```
dim3 blocks(N/4,N/4);
dim3 threads(4,4);
matrix<<< blocks, threads >>>(dev_a, dev_b, dev_c,N);
```

what number of threads is ideal on your GPU?