CSV

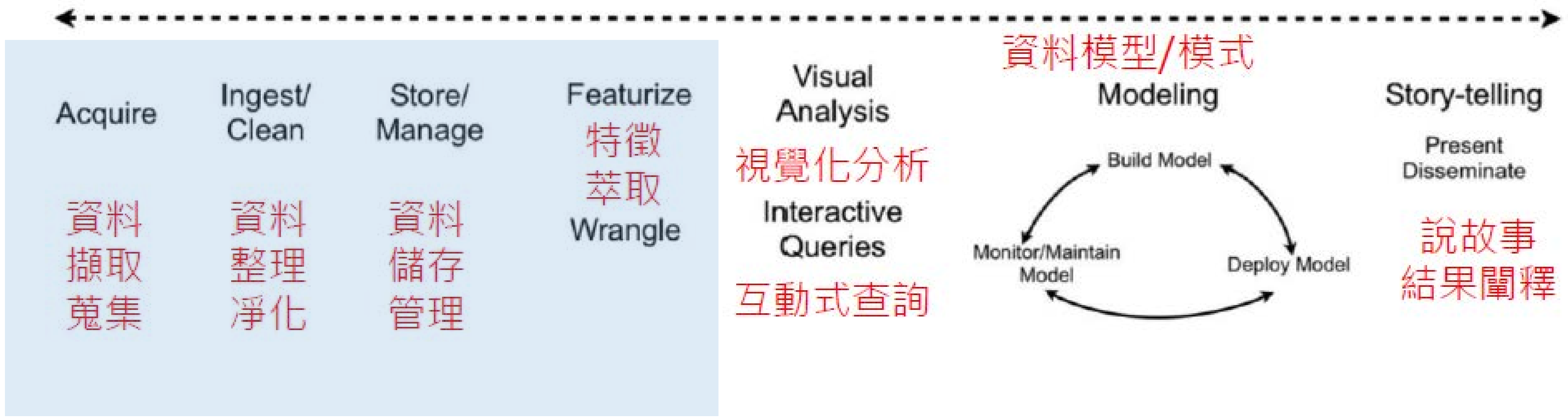
- Column Separated Value (CSV)
  - Refer to RFC 4180
  - E.g.:

```
LOTID800001|"WAFERID01"|"1"|"2"|"-154.8"|"-17.212"|"-154.8"|"-17.212"|"-145.76"|"-11.65"|"-55.936"|"-12.597"|"-7.36"|"91.933"|"95.973"|"3549.021"|"398.198"
LOTID800001|"WAFERID01"|"2"|"2"|"-154.8"|"-17.212"|"-154.8"|"-17.212"|"-145.76"|"-11.14"|"-54.208"|"74.463"|"22.8"|"71.616"|"5.933"|"-1878.786"|"1139.683"
LOTID800001|"WAFERID01"|"2"|"2"|"-154.8"|"-17.212"|"-154.8"|"-17.212"|"-145.76"|"-10.64"|"-90.192"|"62.196"|"83.978"|"10.389"|"97.151"|"-366.121"|"480.151"
LOTID800001|"WAFERID01"|"2"|"2"|"-154.8"|"-17.212"|"-154.8"|"-17.212"|"-145.76"|"-10.13"|"89.847"|"60.792"|"73.452"|"-26.149"|"45.955"|"161.482"|"490.157"
```

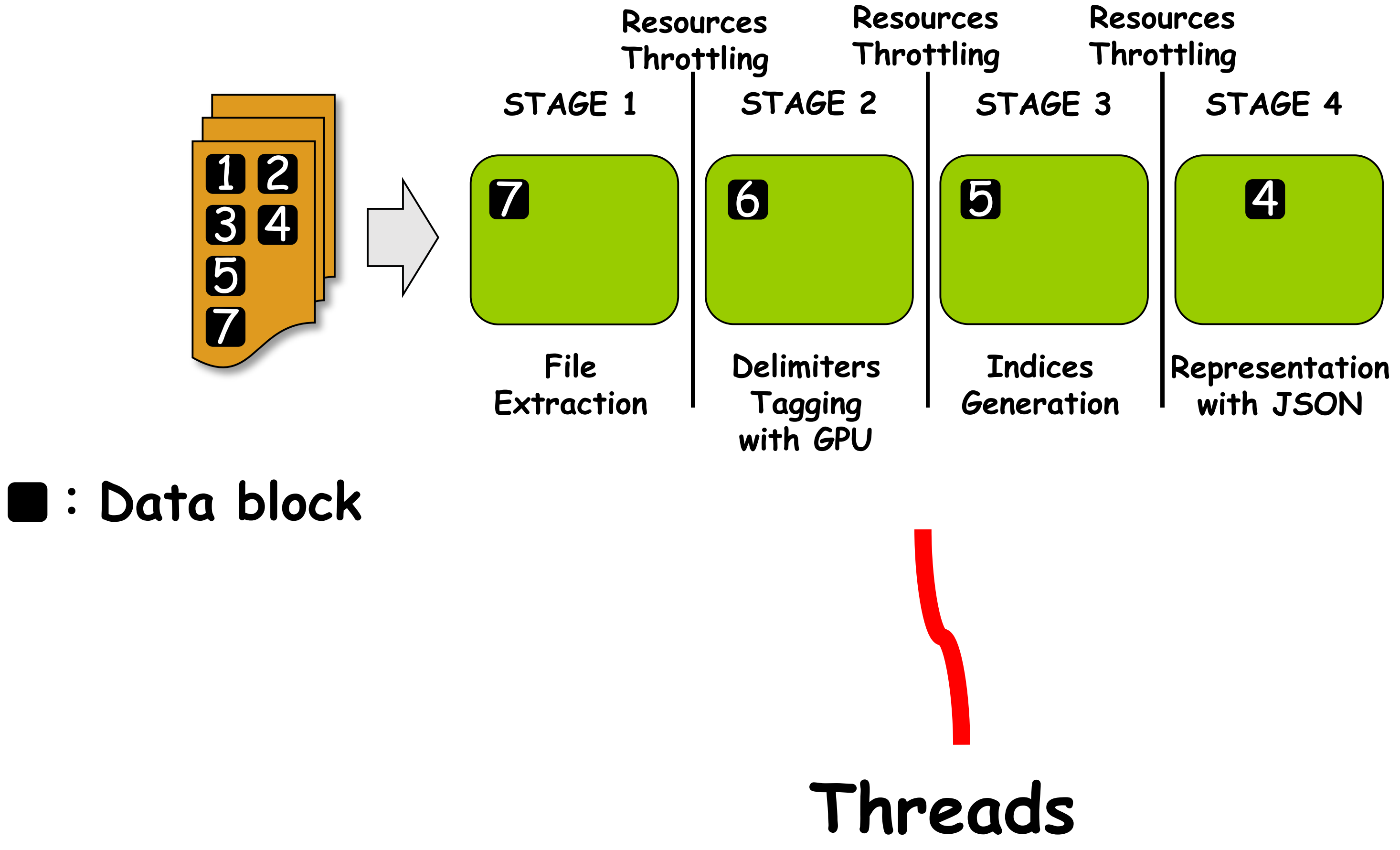
JSON


```json
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

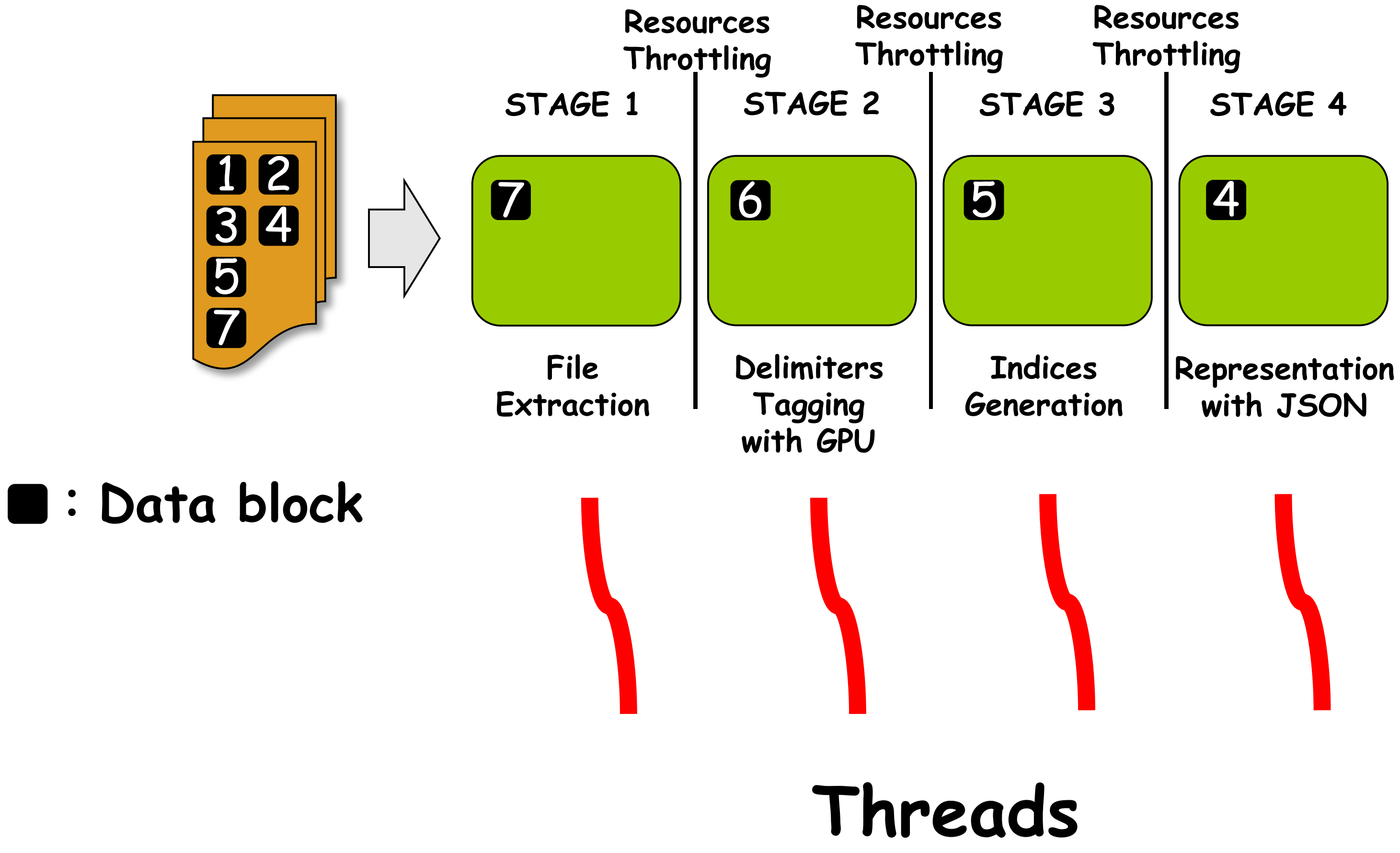
數據科學與工程中的 Extract-Transform-Load (ETL)



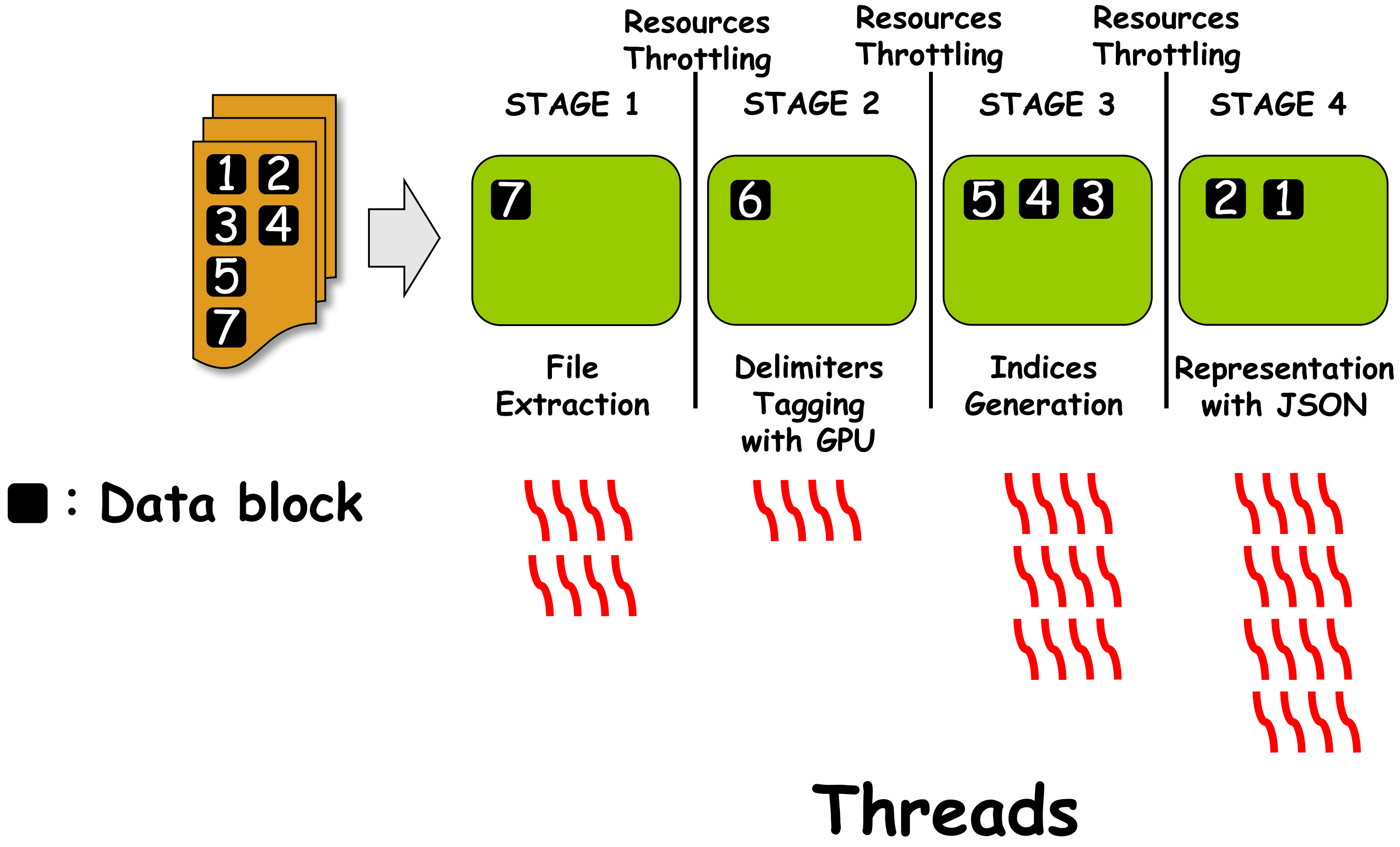
Single Thread for Extract-Transform-Load (ETL)



Pipelined, Multithreaded ETL



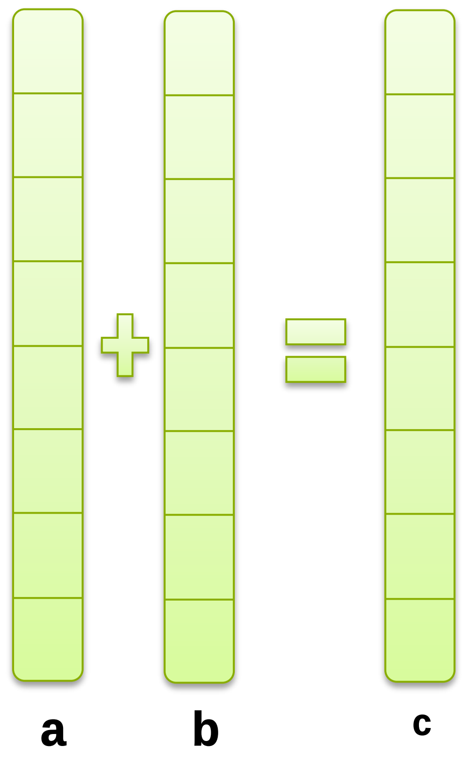
Pipelined, Multithreaded ETL (Enhanced)



Parallel Programming in CUDA C/C++

- GPU computing is about massive parallelism!

- We'll start by adding two integers and build up to vector addition



CPU vs GPU

- GPUs devote more transitors to Arithemtic Logic Units (ALUs)

Heterogeneous Computing (Programming Perspective)

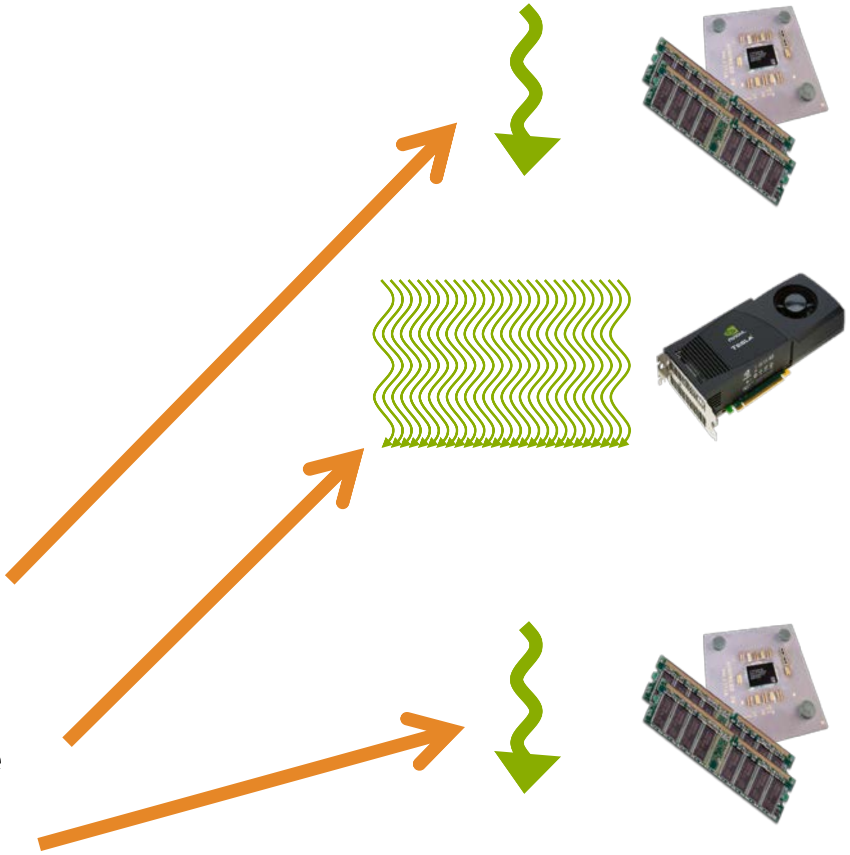
```
#include <iostream>
#include <algorithm>

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

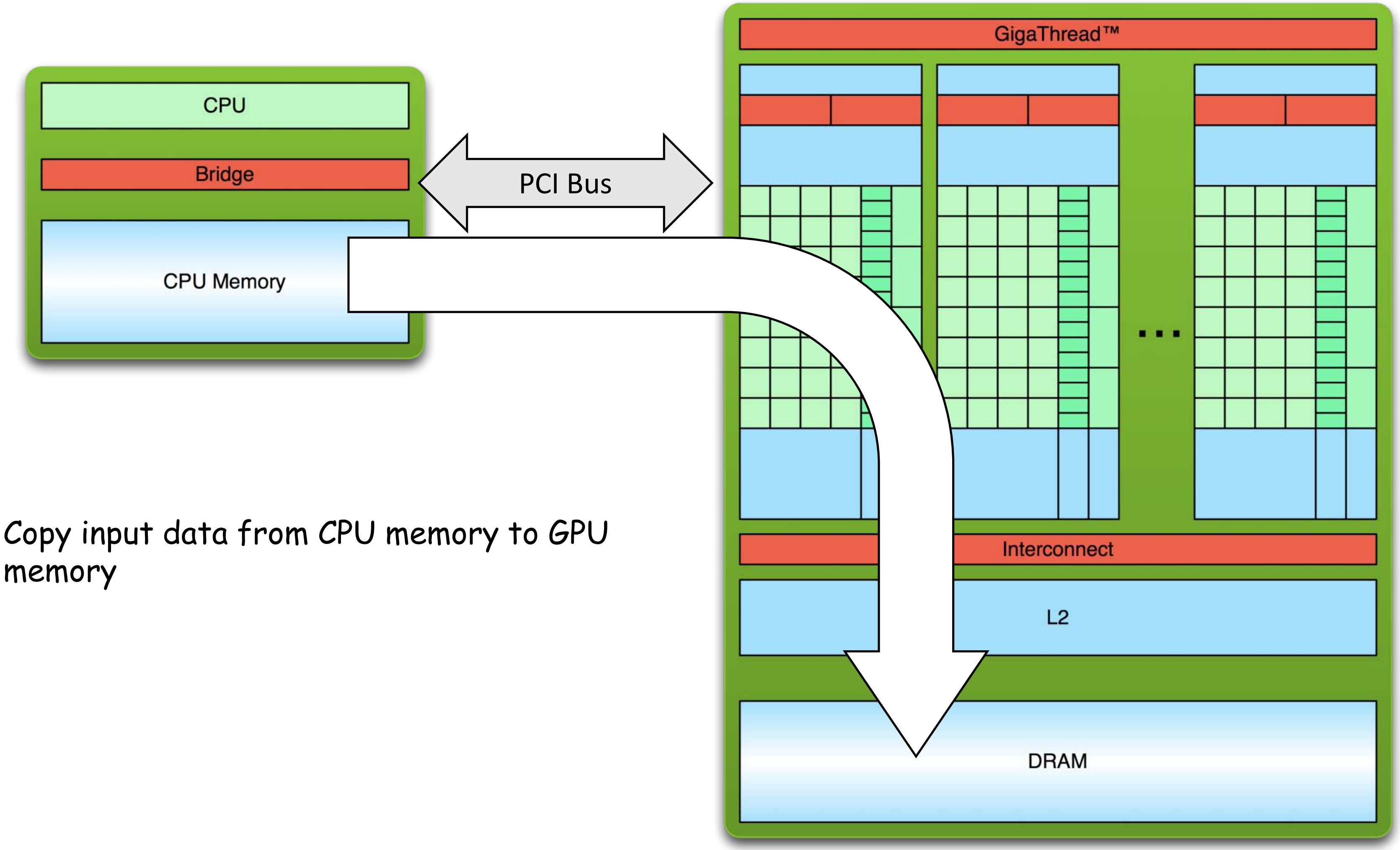
serial code

parallel code

serial code

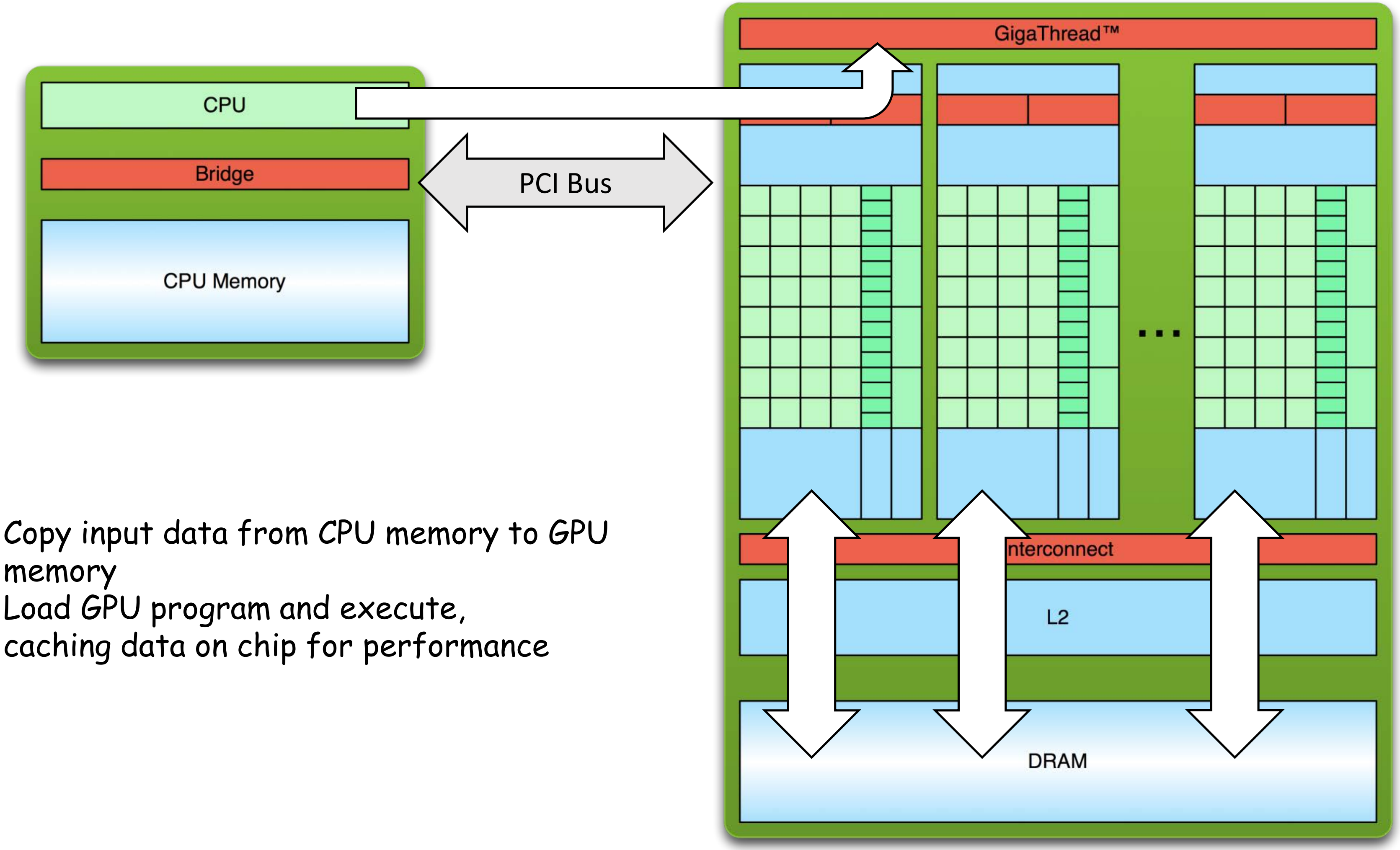


Processing Flow



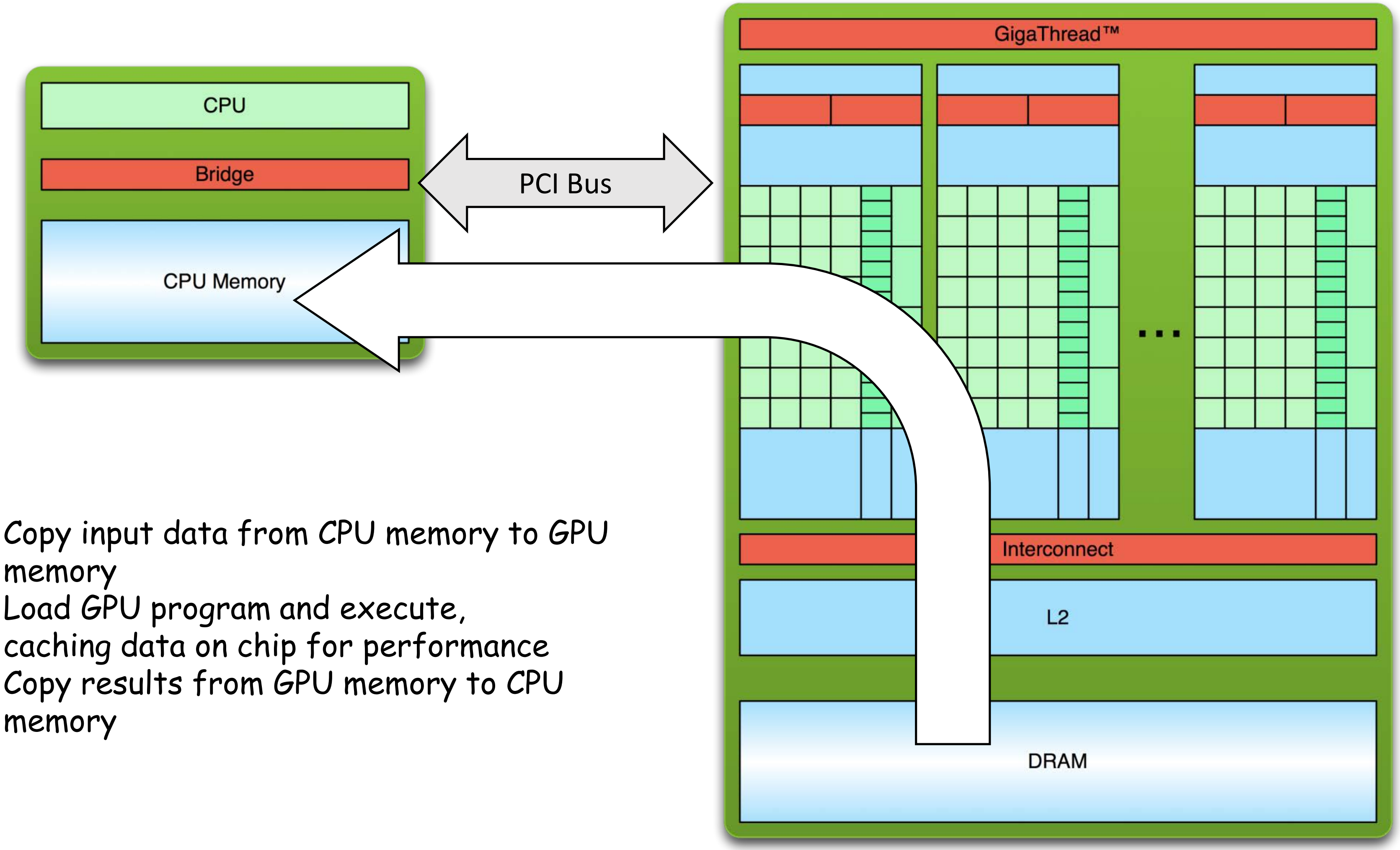
1. Copy input data from CPU memory to GPU memory

Processing Flow (cont'd)



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

Processing Flow (cont'd)



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;
}
```

Hello World! with Device Code (cont'd)

```
__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 (cont'd)

```
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!

Hello World! with Device Code (cont'd)

```
__global__ void mykernel(void){
}

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

- `mykernel()` does nothing

Output:

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

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 (cont'd)

- 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: `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;
}
```

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:

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

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;
}
```

Moving Data…

- CUDA allows us to copy data from one memory type to another.
- This includes dereferencing pointers, even in the host's memory (main system RAM)
- To facilitate this data movement CUDA provides **cudaMemcpy()**

```
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

enum cudaMemcpyKind

CUDA memory copy types

Enumerator:

cudaMemcpyHostToHost	Host -> Host.
cudaMemcpyHostToDevice	Host -> Device.
cudaMemcpyDeviceToHost	Device -> Host.
cudaMemcpyDeviceToDevice	Device -> Device.

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()`

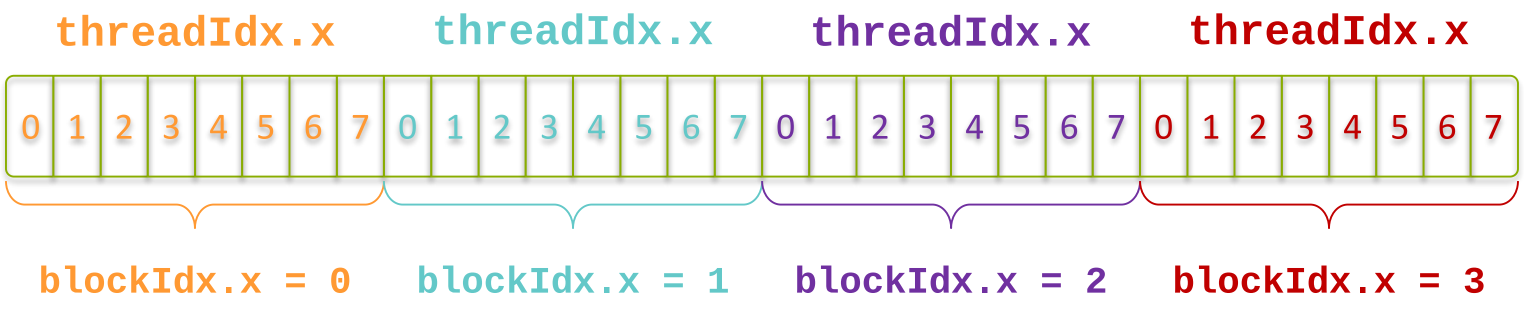
```
 // Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// 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 Blocks and Threads

- We've seen parallel vector addition using:
  - Many blocks with one thread each
  - One block with many 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 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)

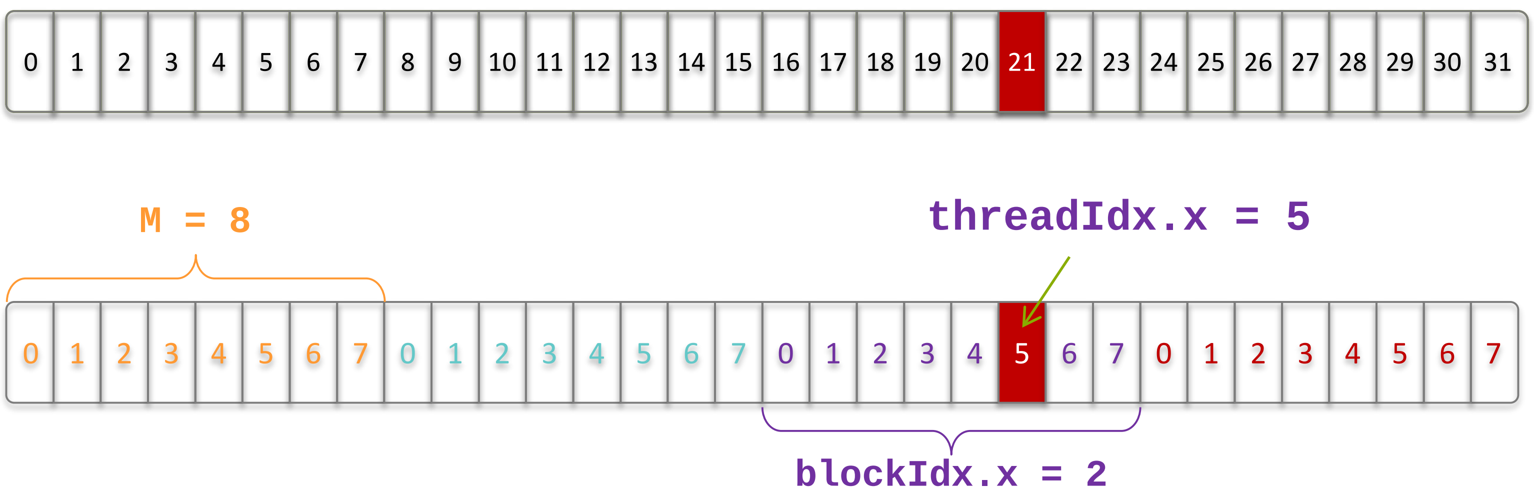


- With M threads/block a unique index for each thread is given by:

```
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;
}
```

Synchronization

- `__syncthreads();`

  Synchronizes all threads within a block
  - Used to prevent data (i.e., RAW/WAR/WAW) hazards

  All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block