# A Very Quick Introduction to CUDA

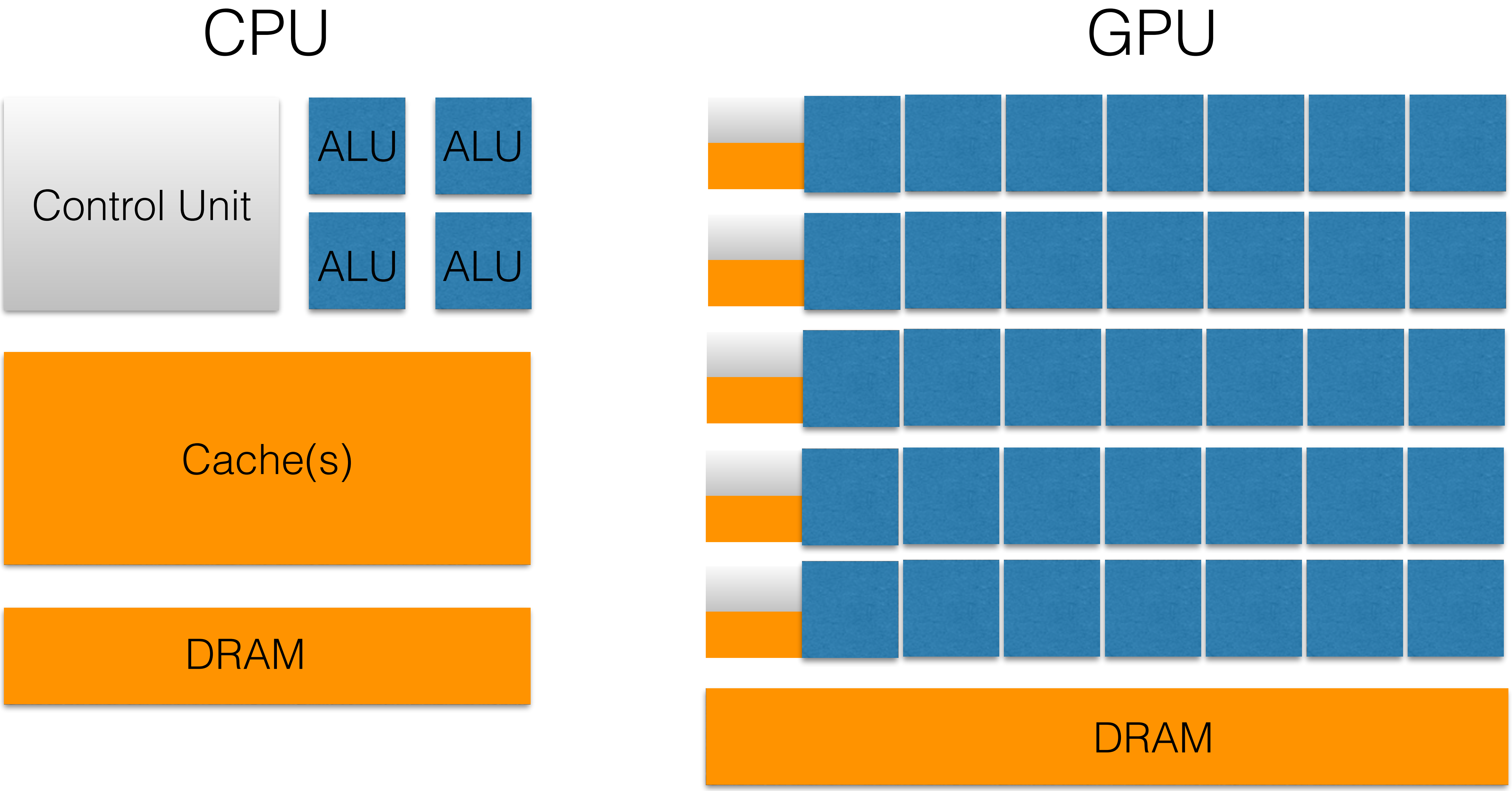
**Burak Himmetoglu**

Supercomputing Consultant

Enterprise Technology Services &
Center for Scientific Computing
University of California
Santa Barbara

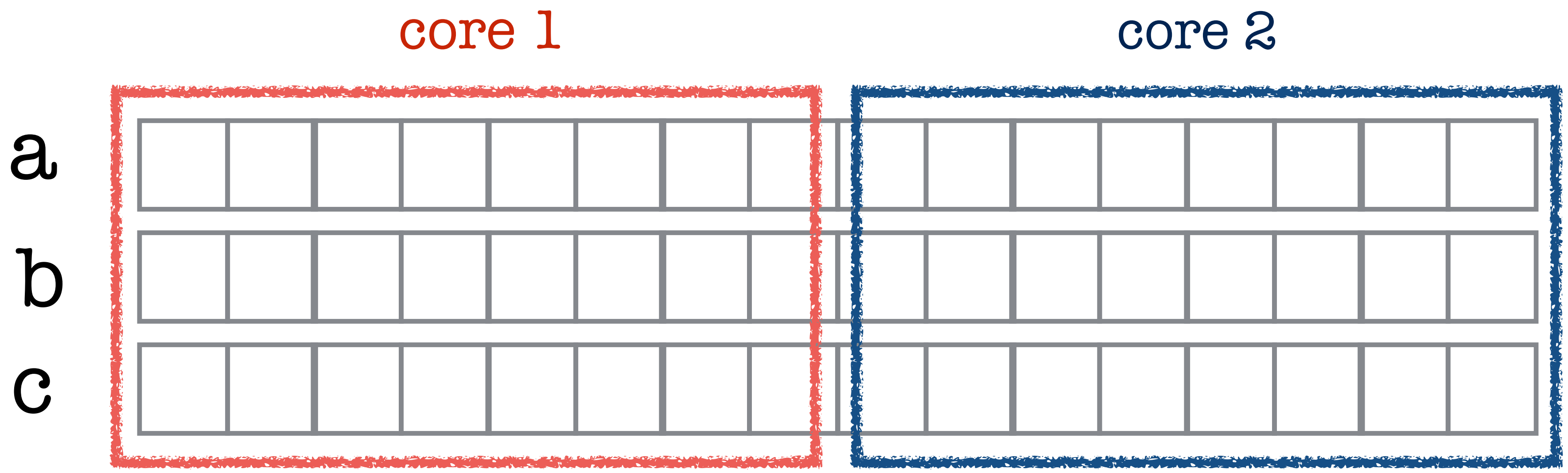
e-mail: bhimmetoglu@ucsb.edu

# Hardware Basics

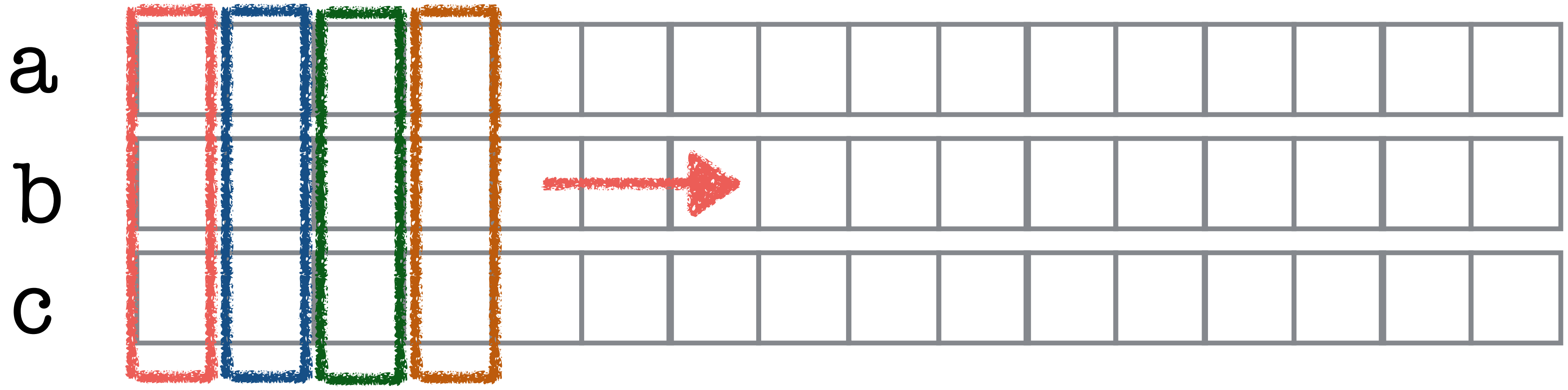


- CPUs are **latency** oriented (minimize execution of serial code)
- GPUs are **throughput** oriented (maximize number of floating point operations)

# CPU vs GPU threads



- If the CPU has n cores, each core processes 1/n elements
- Launching, scheduling threads adds overhead



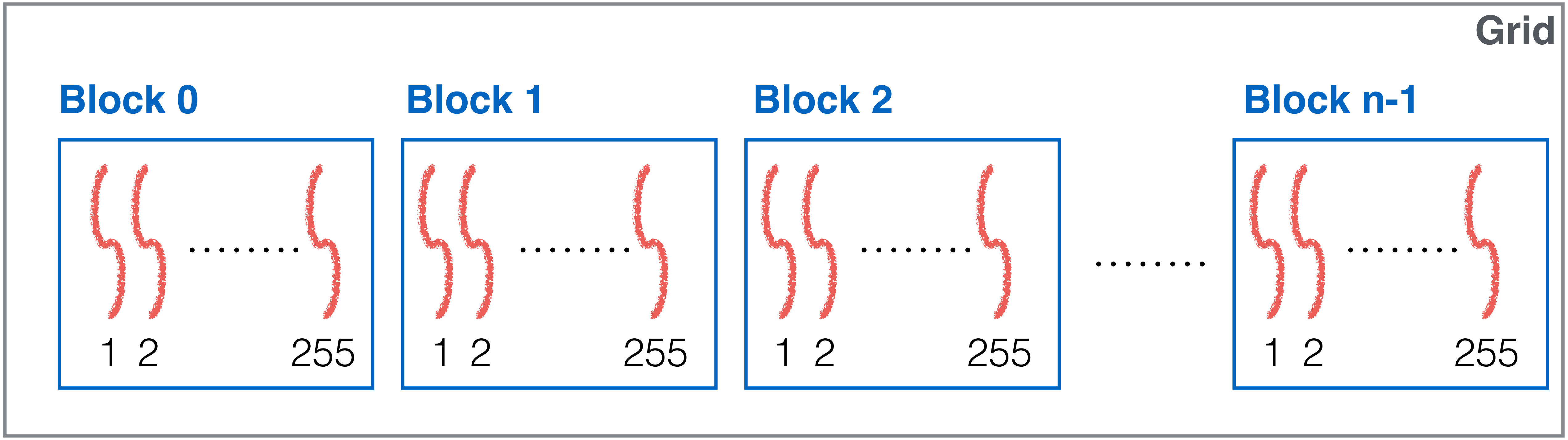
- GPUs process one element per thread
- Scheduled by GPU hardware, not by OS

# CUDA C

- **C**ompute **U**nified **D**evice **A**rchitecture
- NVIDIA GPUs can be programmed by CUDA, extension of C language (CUDA Fortran is also available)
- CUDA C is compiled with **nvcc**
- Numerical libraries: **cuBLAS, cuFFT, Magma, …**

- Host —> CPU; Device —> GPU (They do not share memory!)
- The **HOST** launches a kernel that execute on the **DEVICE**
- A kernel is a data-parallel computation, executed by many **threads**.
- The number of threads are very large (~ 1000 or more)

**Thread Organization**



# CUDA C

- Threads are grouped into blocks.
- Each block shares memory.

Eg. Vector addition:

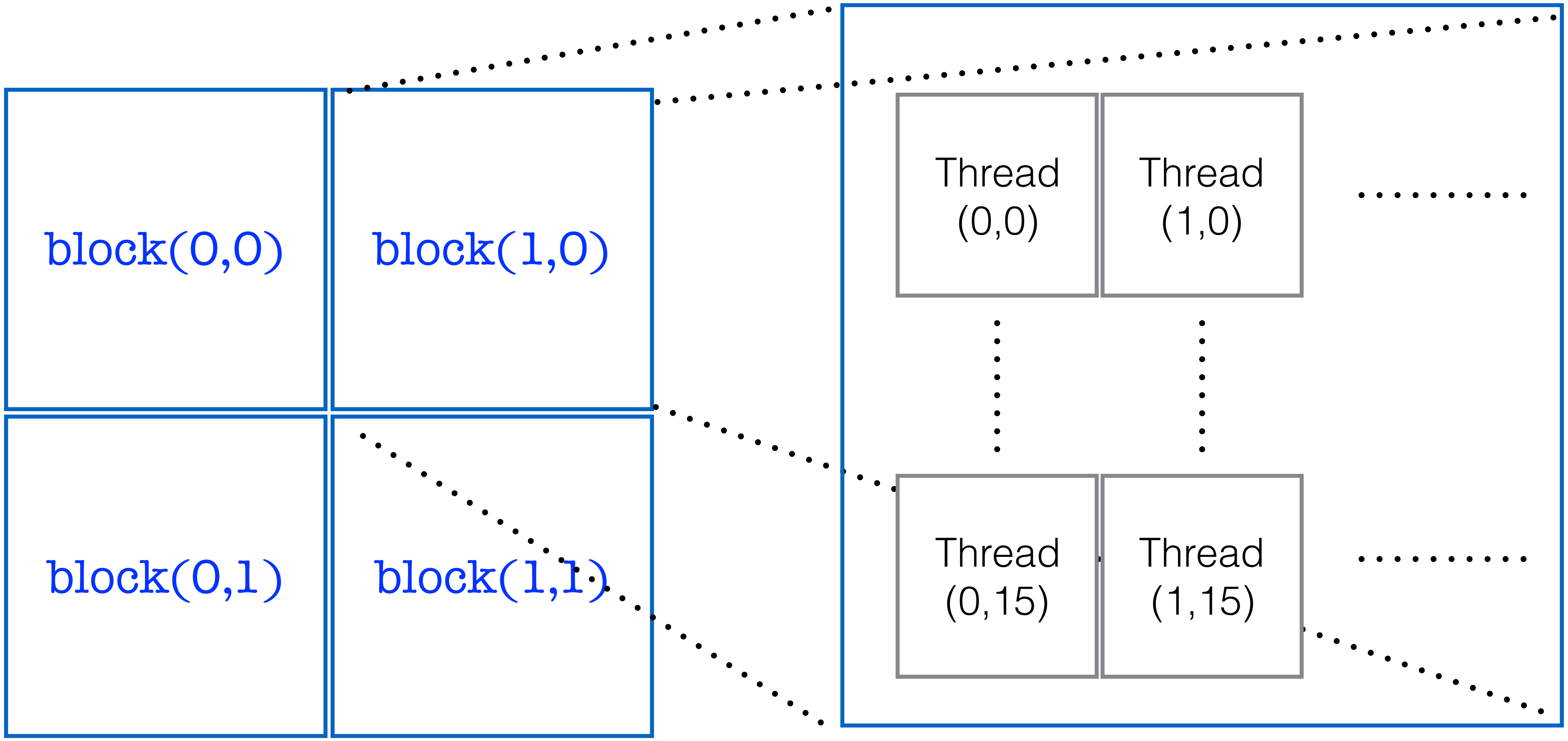
```
int main(void) {
  ...
  vecAdd<<< blocksPerGrid, THREADS_PER_BLOCK >>> (d_A, d_B, d_C);
  ...
}

__global__ static void vecAdd (float *a, float *b, float *c){
  .....
}
```

The __global__ qualifer alerts the compiler that the code block will run on the **DEVICE**, but can be called from the **HOST**.

# CUDA C

- Grids and threads can also be arranged in 2d arrays (useful for image processing)

```
dim3 blocks(2,2)
dim3 threads(16,16)
....
kernel <<< blocks, threads >>>( );
...
```



block(0,0)

block(1,0)

block(0,1)

block(1,1)

Thread (0,0)

Thread (1,0)

Thread (0,15)

Thread (1,15)

# Code Example - 1

## Hello World!

```
#include <stdio.h>

__device__ const char *STR = “HELLO WORLD!”;
const int STR_LENGTH = 12;

__global__ void hello(){
      printf(“%c\n”, STR[threadId.x % STR_LENGTH]);
}

int main(void){
      int threads_per_block = STR_LENGHT;
      int blocks_per_grid = 1;

      hello <<< blocks_per_grid, threads_per_block >>> ();

      cudaDeviceSynchronize();

      return 0;
}
```

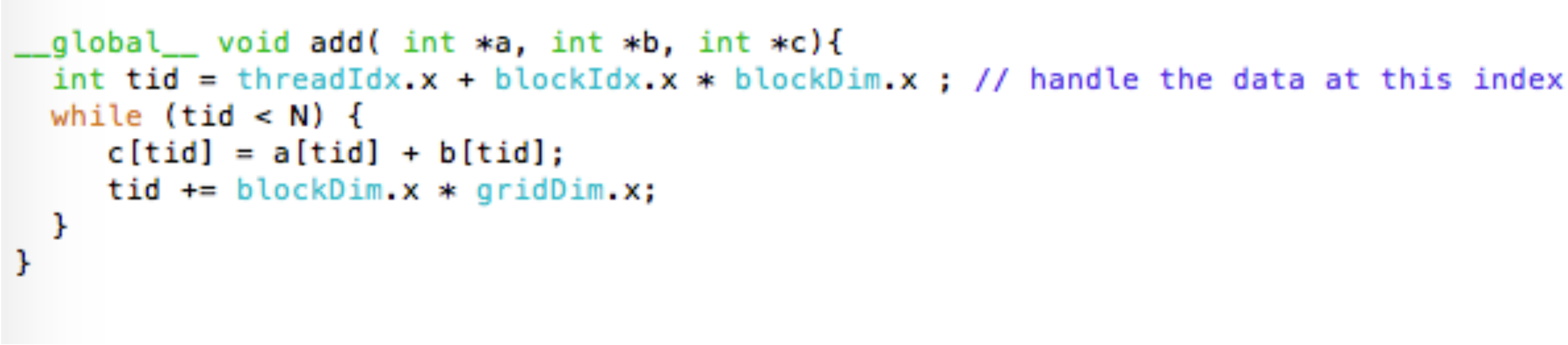
Output:

```
H
E
L
L
O

W
O
R
L
D
!
```

Halt host thread execution on CPU until the device has finished processing all previously requested tasks.

# Code Example - 2

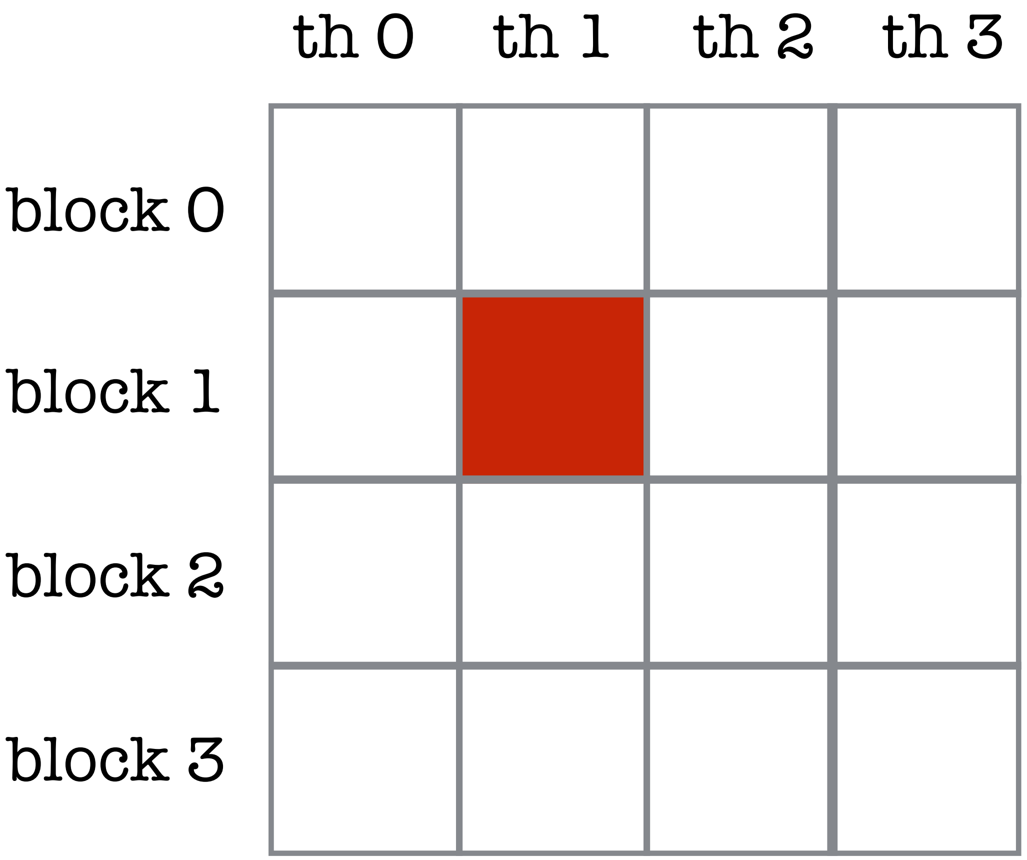
## Vector Addition (Very large vectors)



```
__global__ void add( int *a, int *b, int *c){
  int tid = threadIdx.x + blockIdx.x * blockDim.x ; // handle the data at this index
  while (tid < N) {
     c[tid] = a[tid] + b[tid];
     tid += blockDim.x * gridDim.x;
  }
}
```

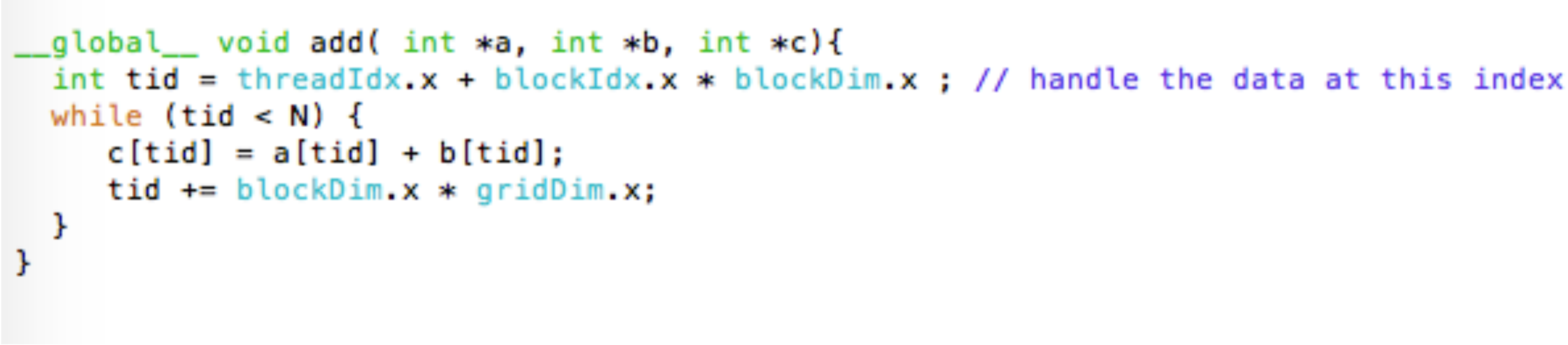
e.g.: **blockDim** = 4, **gridDim** = 4

tid = th.id + blk.id * blk.dim
= 1 + 1 * 4
= 5



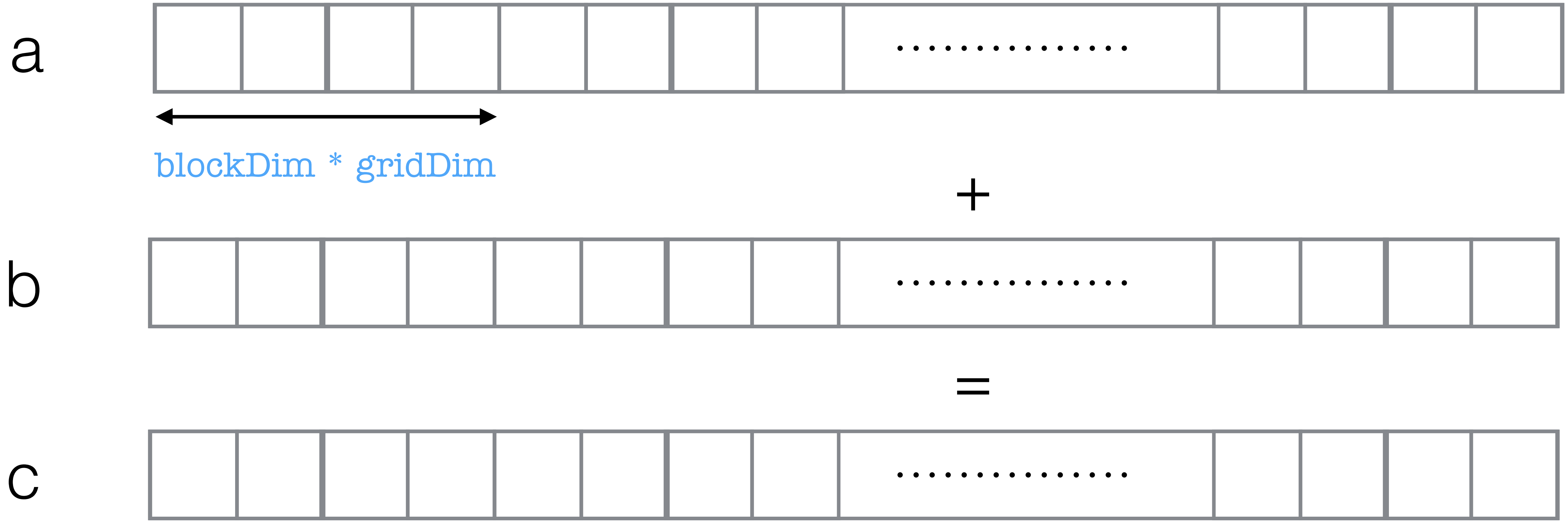
# Code Example - 2

## Vector Addition (Very large vectors)



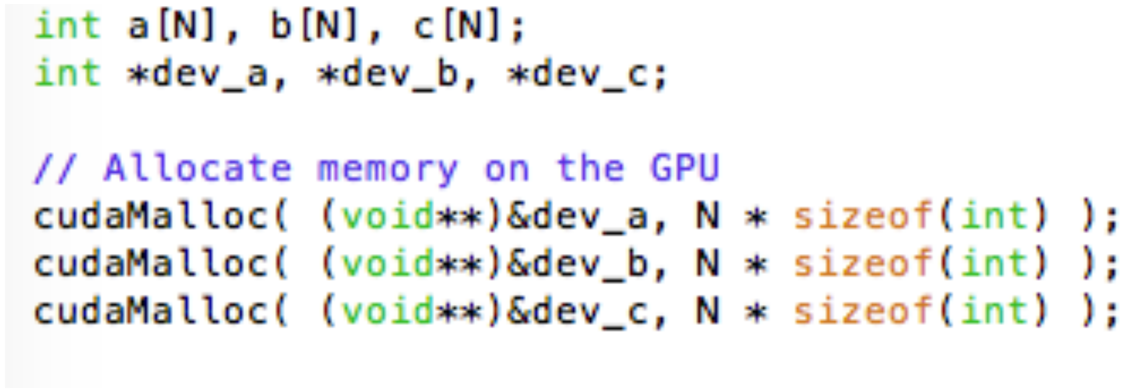
```
__global__ void add( int *a, int *b, int *c){
  int tid = threadIdx.x + blockIdx.x * blockDim.x ; // handle the data at this index
  while (tid < N) {
      c[tid] = a[tid] + b[tid];
      tid += blockDim.x * gridDim.x;
  }
}
```

e.g.: **N** = 256, **blockDim** = 2, **gridDim** = 2 —> **offset = blockDim * gridDim**



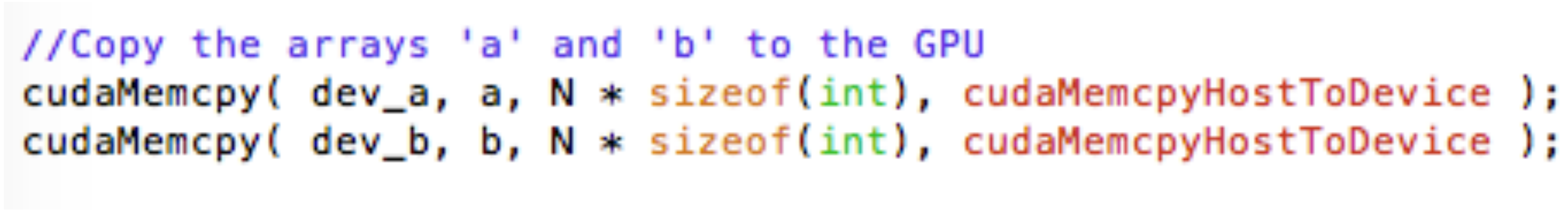
# Code Example - 2

- Define arrays to be used on the HOST, and allocate memory.



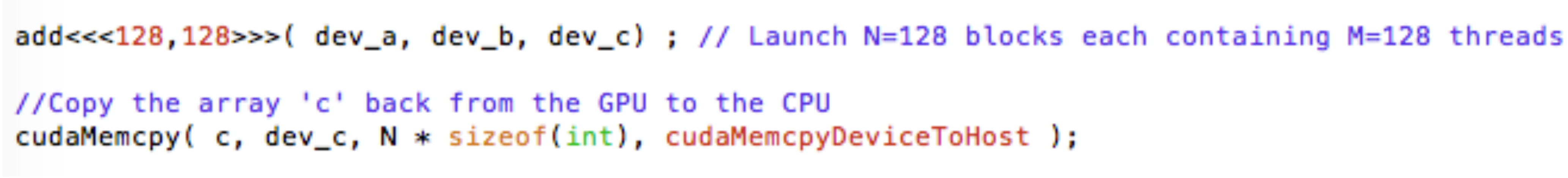
```
int a[N], b[N], c[N];
int *dev_a, *dev_b, *dev_c;

// Allocate memory on the GPU
cudaMalloc( (void**)&dev_a, N * sizeof(int) );
cudaMalloc( (void**)&dev_b, N * sizeof(int) );
cudaMalloc( (void**)&dev_c, N * sizeof(int) );
```

- Copy arrays to the DEVICE



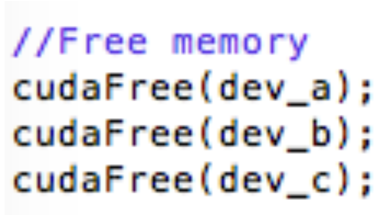
```
//Copy the arrays 'a' and 'b' to the GPU
cudaMemcpy( dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice );
cudaMemcpy( dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice );
```

- Launch the kernel, then copy result from DEVICE to HOST



```
add<<<128,128>>>( dev_a, dev_b, dev_c) ; // Launch N=128 blocks each containing M=128 threads

//Copy the array 'c' back from the GPU to the CPU
cudaMemcpy( c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost );
```

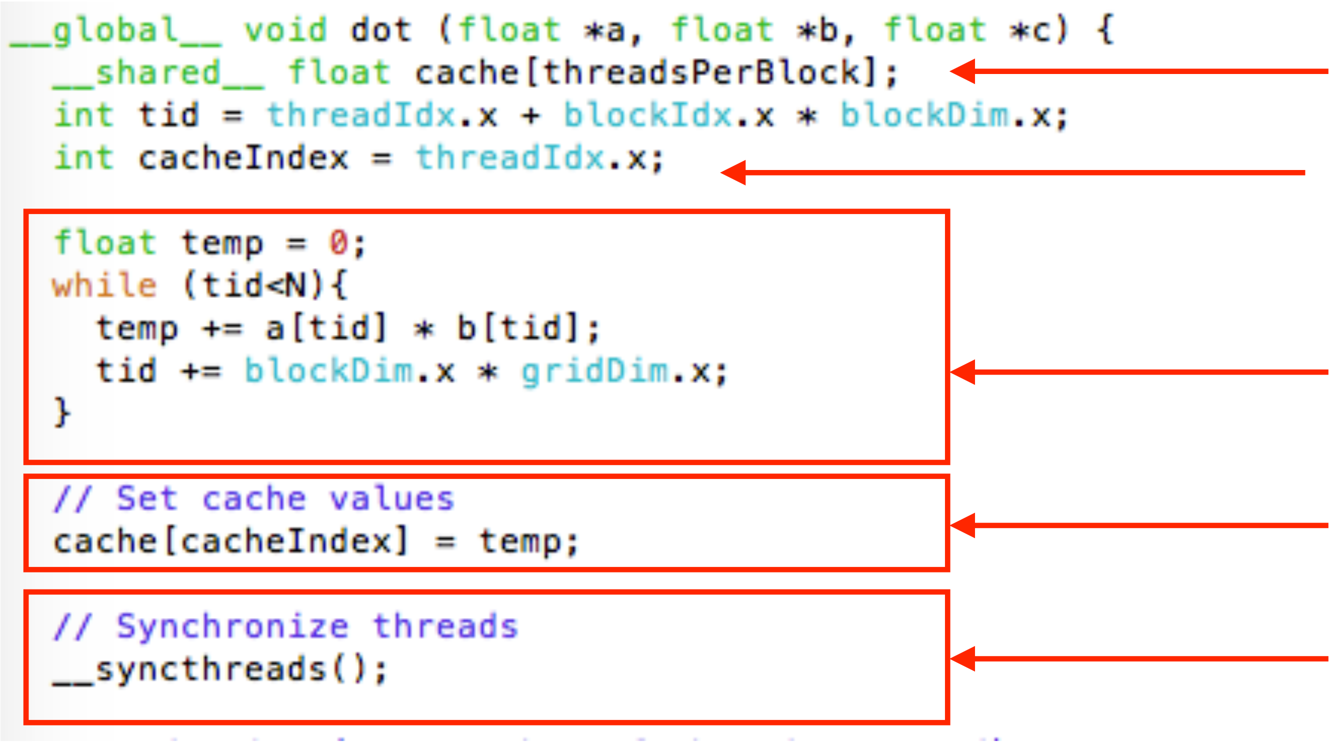
- Free memory



```
//Free memory
cudaFree(dev_a);
cudaFree(dev_b);
cudaFree(dev_c);
```

# Code Example - 3

## Dot product



```
__global__ void dot (float *a, float *b, float *c) {
  __shared__ float cache[threadsPerBlock];
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  int cacheIndex = threadIdx.x;

  float temp = 0;
  while (tid<N){
    temp += a[tid] * b[tid];
    tid += blockDim.x * gridDim.x;
  }

  // Set cache values
  cache[cacheIndex] = temp;

  // Synchronize threads
  __syncthreads();
```

vector for storing each block’s result

index used for storing

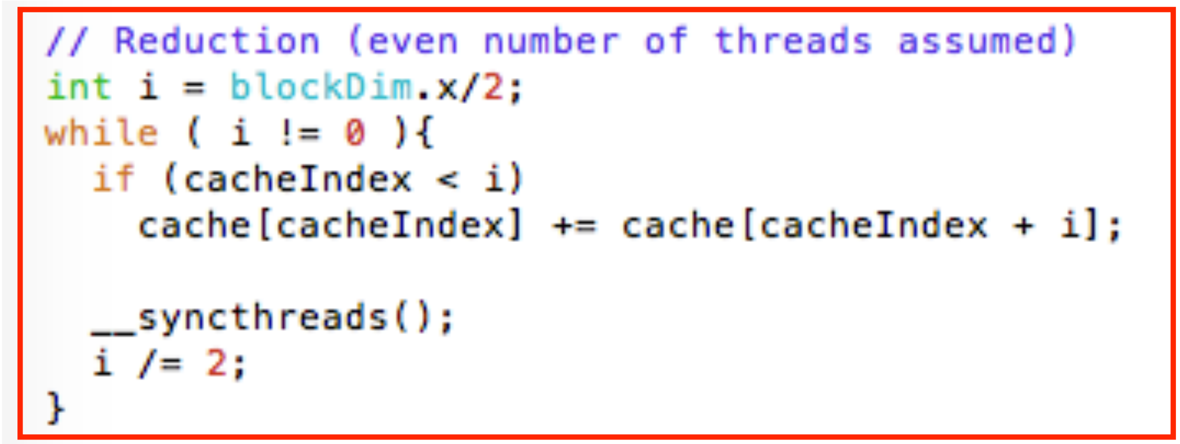
temp has the result within each block

For each block, there is a **different** cache vector.

Wait until all threads finish!

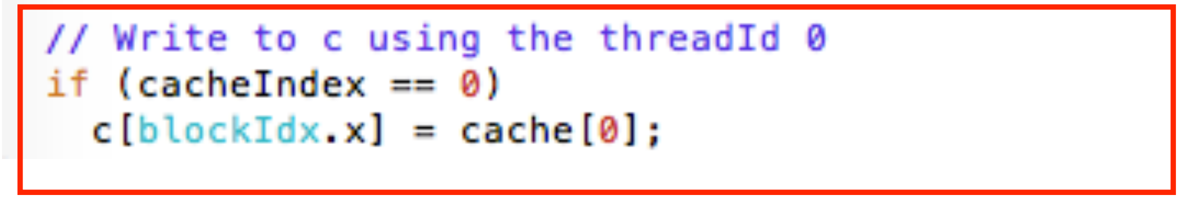
- Recall, each Block shares memory!
- Each block will have a its own copy of **`cahce[]`**, i.e. a partial result.
- Final step is reduction, i.e. summing all the partial results in **`cahce[]`** to obtain a final answer.

# Code Example - 3



```
// Reduction (even number of threads assumed)
int i = blockDim.x/2;
while ( i != 0 ){
  if (cacheIndex < i)
    cache[cacheIndex] += cache[cacheIndex + i];

  __syncthreads();
  i /= 2;
}
```

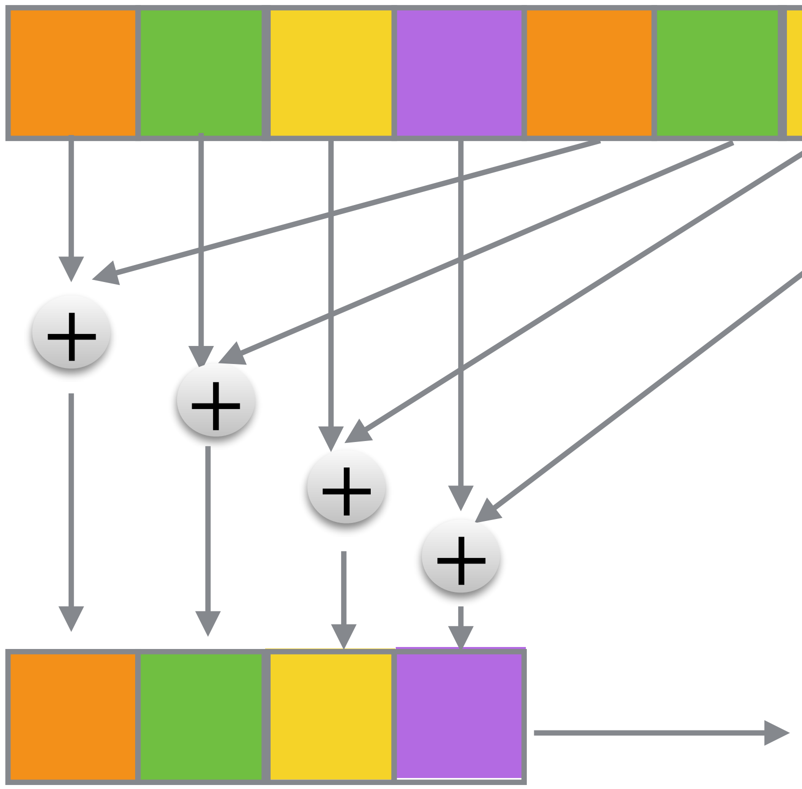
Parallel reduction



```
// Write to c using the threadId 0
if (cacheIndex == 0)
  c[blockIdx.x] = cache[0];
```

Finally, write the final answer, with one thread (serial).

Parallel reduction:
(Not the best one!)



BlockDim = 8

Repeat for BlockDim/2
(i /=2); while ( i !=0)

# GPUs on Comet

- 1944 Standard compute nodes
- 36 GPU Nodes:
  - Intel Xeon E5-2680v3
  - NVIDIA K80 GPUs (11GB)

GPU Examples:

`/share/apps/examples/GPU`

# GPUs on Comet

```
$ module load cuda
$ nvcc -o hello_cuda.x hello_cuda.cu
```

cuda.job

```
#!/bin/bash
#SBATCH -p gpu-shared
#SBATCH —gres=gpu:1
#SBATCH —job-name=“hellocuda”
#SBATCH —output=“hellocuda.%j.%N.out”
#SBATCH -t 00:01:00
#SBATCH -A TG-SEE150004

cd ~/Working_directory

./hello_cuda.x
```

```
$ sbatch cuda.job
```

# Exercise

Examine and run the code **add_vec_times.cu** and compare it with **add_vec_gpu_thd-blk.cu** and answer the following questions:

- Vary **THREADS_PER_BLOCK: 1, 2, 4, 8, 16, 32, 64, 128, 256**
- Record the time printed

1. How many blocks are launched for each case?
2. Until what value the timing decreases linearly?
3. What is the explanation of the loss of the linear behavior after this value? (Hint: search for “warps”)