Parallel Programming

CUDA Programming Model

`

Overview

- CUDA programming model
  - Host code and device code (kernel)
  - CUDA Threads: Grids, Blocks
  - CUDA memory allocation and copy
  - Kernel programs and their invocation
- Simple CUDA program examples

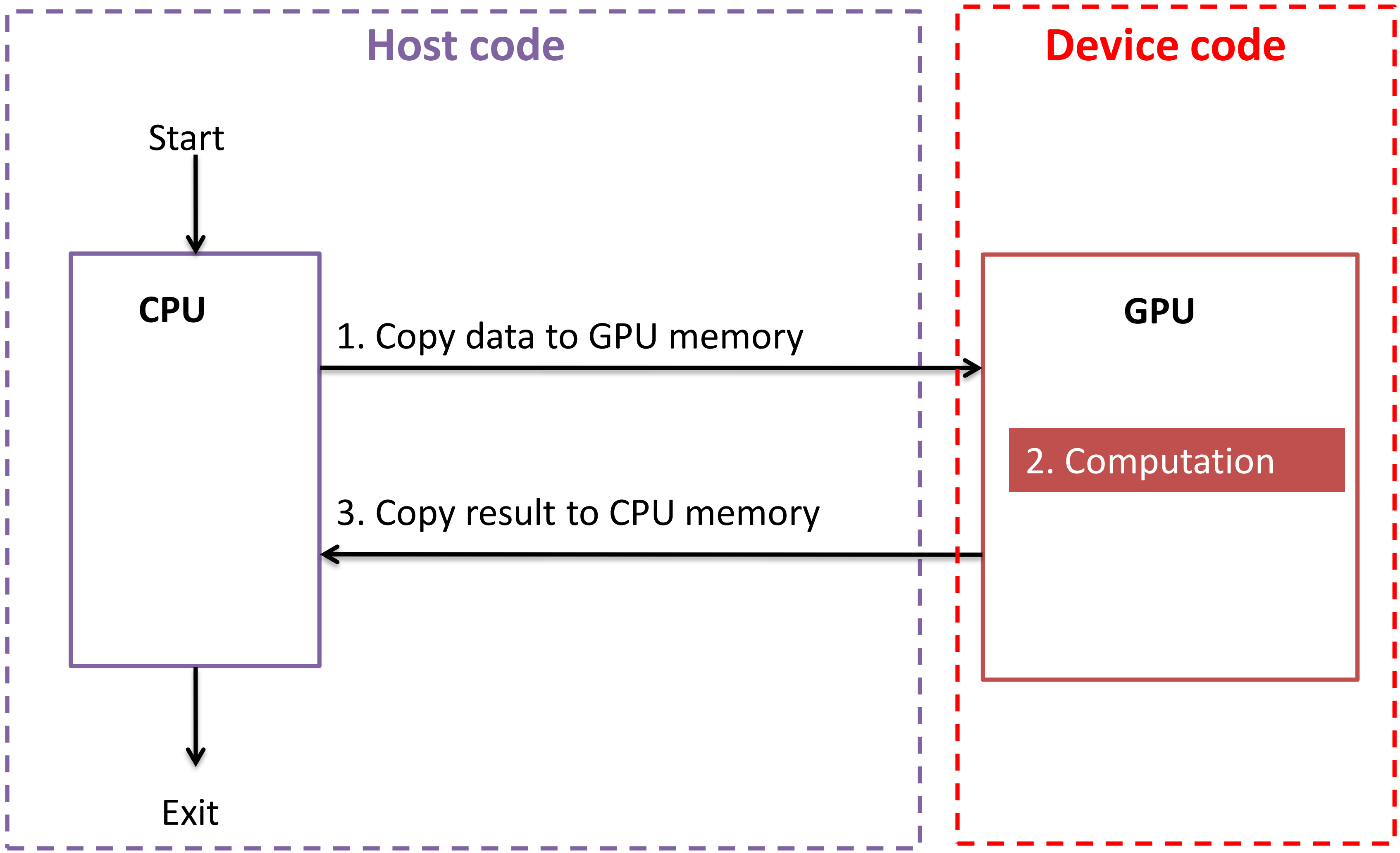
CUDA: Compute Unified Device Architecture

- A parallel computing architecture developed by NVIDIA.
  - Hardware: NVIDIA GPUs, from embedded devices, graphics cards for laptops and desktops, to dedicated server products for computation.
  - Software
    - Tool kit, device drivers, and programming SDK.
    - Support C, Fortran, Matlab, and other languages.
- We teach CUDA C programming in this course

Host and Device Code

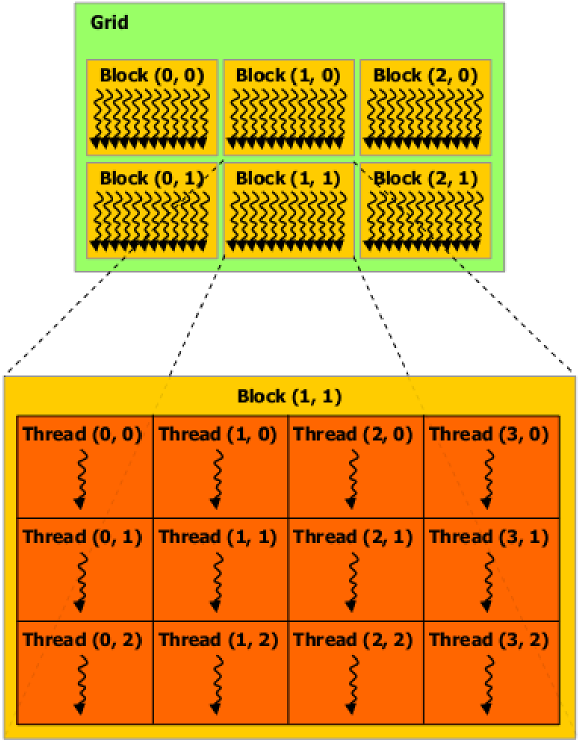
- A CUDA program consists of two parts: *host* and *device* (or *kernel*) code.
- Host code: executed on the CPU
  - Memory copy between the GPU and the CPU
  - Computation on the CPU and call GPU kernel
- Device code: executed on the GPU
  - GPU-based computation
- A CUDA program always starts from the host code, and then invokes the GPU kernels.

Processing Flow of a CUDA Program



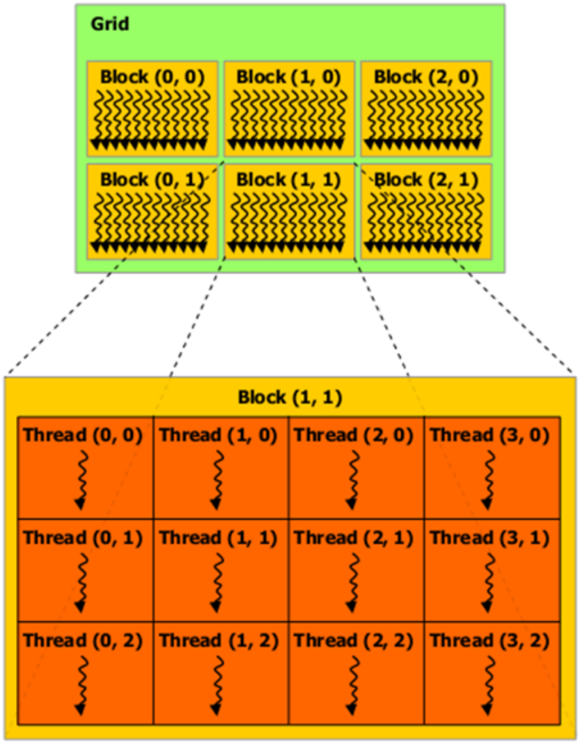
Threads in a CUDA Kernel

- Each kernel corresponds to a ***grid*** of threads.
- Each grid consists of multiple thread ***blocks***.
- Each thread block contains multiple ***threads***.



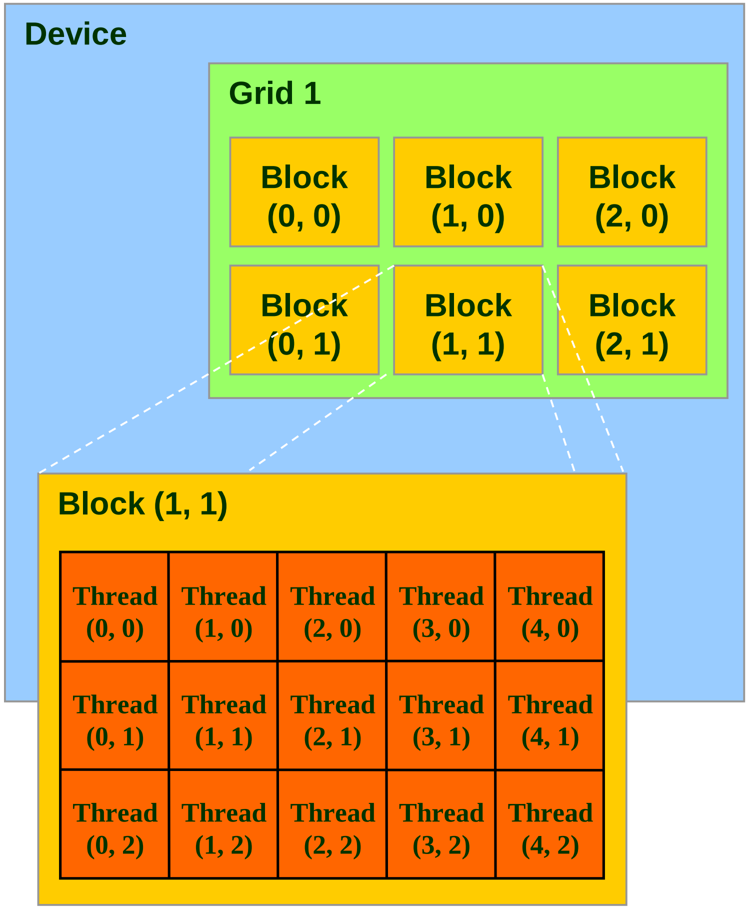
Grid and Block Dimensions

- A grid consists of i-dimension (i=1,2,3) **blocks**.
  - gridDim.x, gridDim.y, gridDim.z
- A thread block contains **threads** organized in 1-3 dimensions
  - blockDim.x, blockDim.y, blockDim.z.
- Any unspecified dimension is set to size 1.



Block and Thread IDs

- Threads and blocks have built-in IDs
  - Block ID: (blockIdx.x, blockIdx.y, blockIdx.z)
  - Thread ID: 1D, 2D, or 3D **within a block** (threadIdx.x, threadIdx.y, threadIdx.z)



Courtesy: NDVIA

Device Code (Kernel)

- The device code is the same for each thread.
- A kernel function has the prefix *__global__*, and has a *void* return type.

__global__ void kernel1*(param1, …)*

Note: device code has no direct access to main memory.

Kernel Invocation in Host Code

kernelName<<<#block, #thread, shared_size, s>>> **(*param1, …*)**

#block: number of thread blocks in the grid
#thread: number of threads per block
shared_size: optional; size of shared memory per block, default 0.

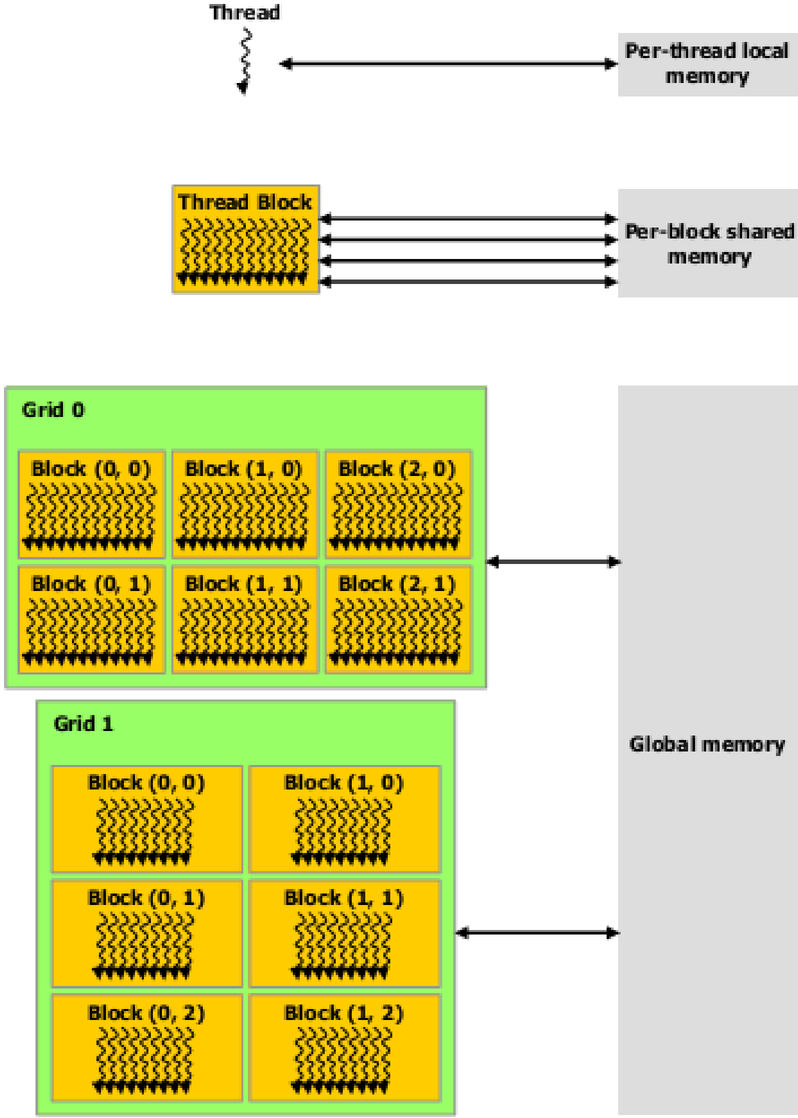
s: optional; the associated stream, default 0.

Memory Management in Host Code

- GPU memory management functions
  - GPU memory allocation:
    cudaMalloc(*devPtr*, *size*)
    cudaFree(*devPtr*)
  - Memory copy:
    cudaMemcpy(*dst*, *src*, *size*, *direction*)
    direction*: *cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost

Note: host code has no direct access to GPU memory.

CUDA Memory Hierarchy



- **Registers**: only available within a thread.
- **Shared memory**: accessed by threads in the same thread block.
- **Global memory**: can be accessed by all threads.

A Simple Program on the CPU

```
int main()
{
    int *h_A, *h_B, *h_C;
    int i;
    int N = 4096;
    size_t size = N * sizeof(int);

    // Allocate input vectors h_A and h_B in host memory
    h_A = (int*)malloc(size);
    h_B = (int*)malloc(size);
    h_C = (int*)malloc(size);

    /*initialize h_A and h_B here*/

    //vector Add
    for (i = 0; i < N; i++)
        h_C[i] = h_A[i] + h_B[i];

   //Free host memory
   free(h_A);
   free(h_B);
   free(h_C);

   return 0;
}
```

A recommended common practice is to name a host-resident structure with the prefix "*h_*" (host), and a device-resident structure with "*d_*"(device).

CUDA Program: Set Up on the Host

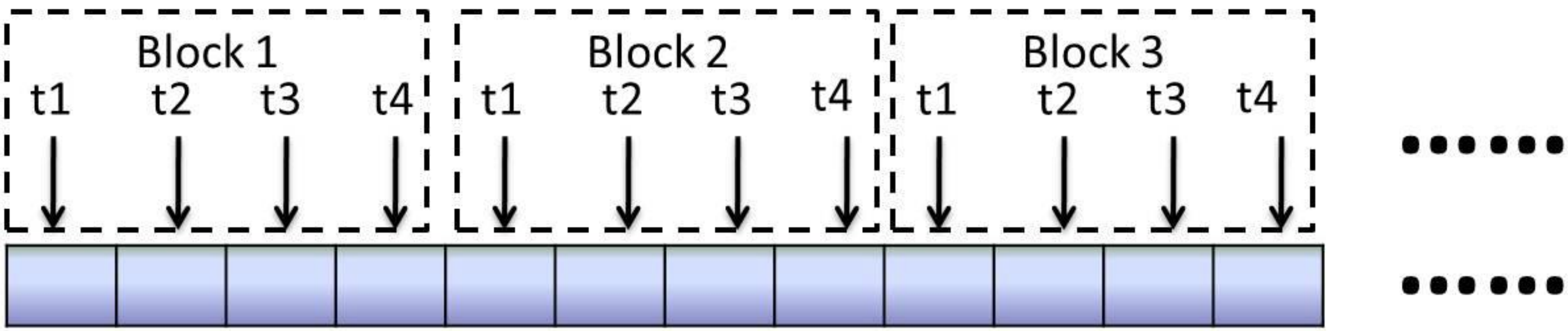
```
// Host code
int main()
{
    int *h_A, *h_B, *h_C, *d_A, *d_B, *d_C;
    int N = 4096;
    size_t size = N * sizeof(int);

    // Allocate input vectors h_A and h_B in host memory
    h_A = (int*)malloc(size);
    h_B = (int*)malloc(size);
    h_C = (int*)malloc(size);

    for (int i = 0; i < N; i++)
    {
        h_A[i] = i;
        h_B[i] = i;
    }

    // Allocate vectors in device memory
    cudaMalloc((void**)&d_A, size);
    cudaMalloc((void**)&d_B, size);
    cudaMalloc((void**)&d_C, size);

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

CUDA Program: Invoke the Kernel

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

CUDA Program: The Device Code

```
// Device code
__global__ void VecAdd(int* A, int* B, int* C)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    C[i] = A[i] + B[i];
}
```



CUDA Program: Wrap Up on the Host

```
                 .
                 .
 // Copy result from device memory to host memory
 // h_C contains the result in host memory
 cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

//Free host memory
free(h_A);
free(h_B);
free(h_C);

//Free device memory
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);

}
```

Another Example

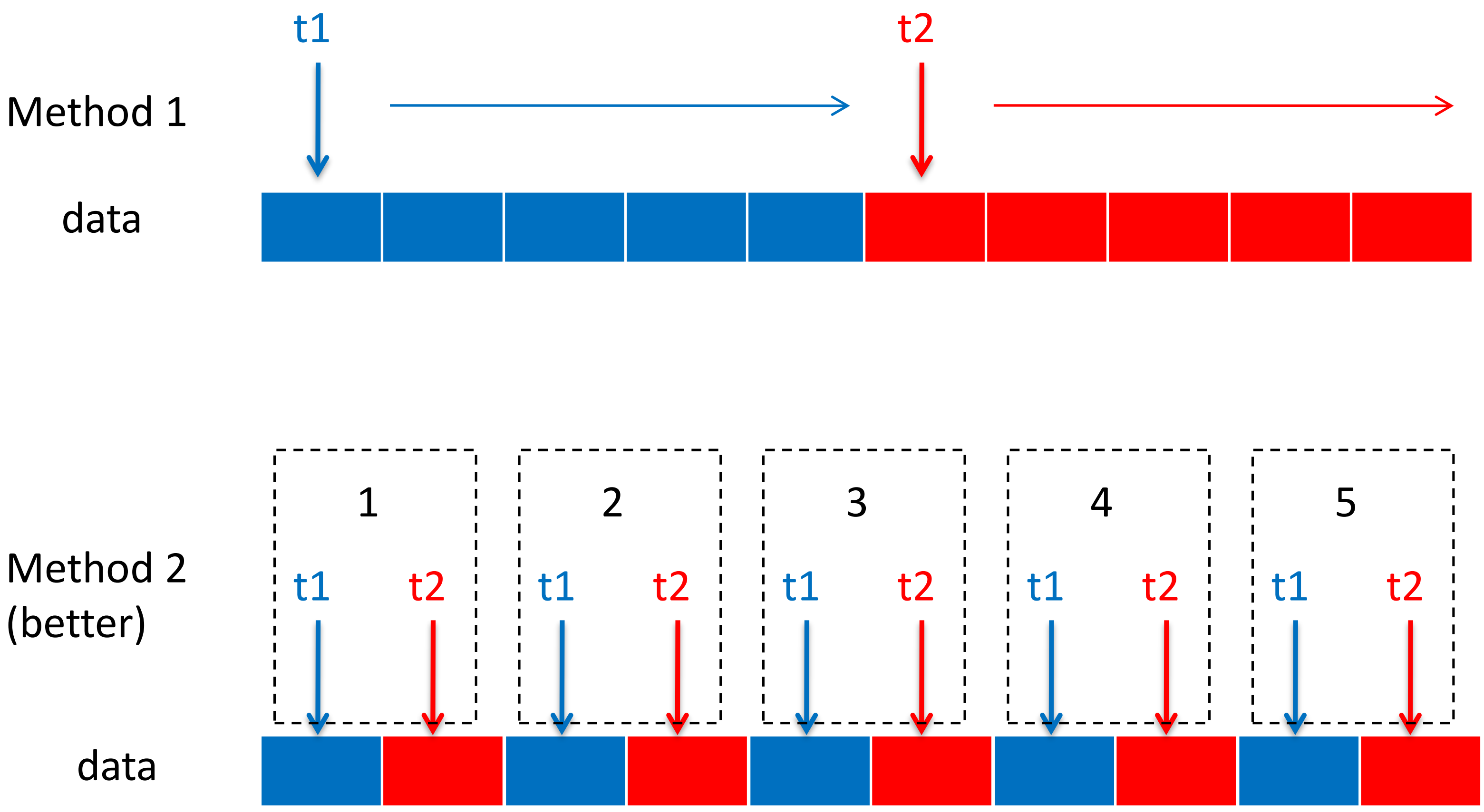
- Given an array of *n* elements, increment each element.
- A C program without CUDA

```
for(int i = 0; i < n; i++) {
        h_data[i] += 1;
}
```

The Parallelization on the GPU

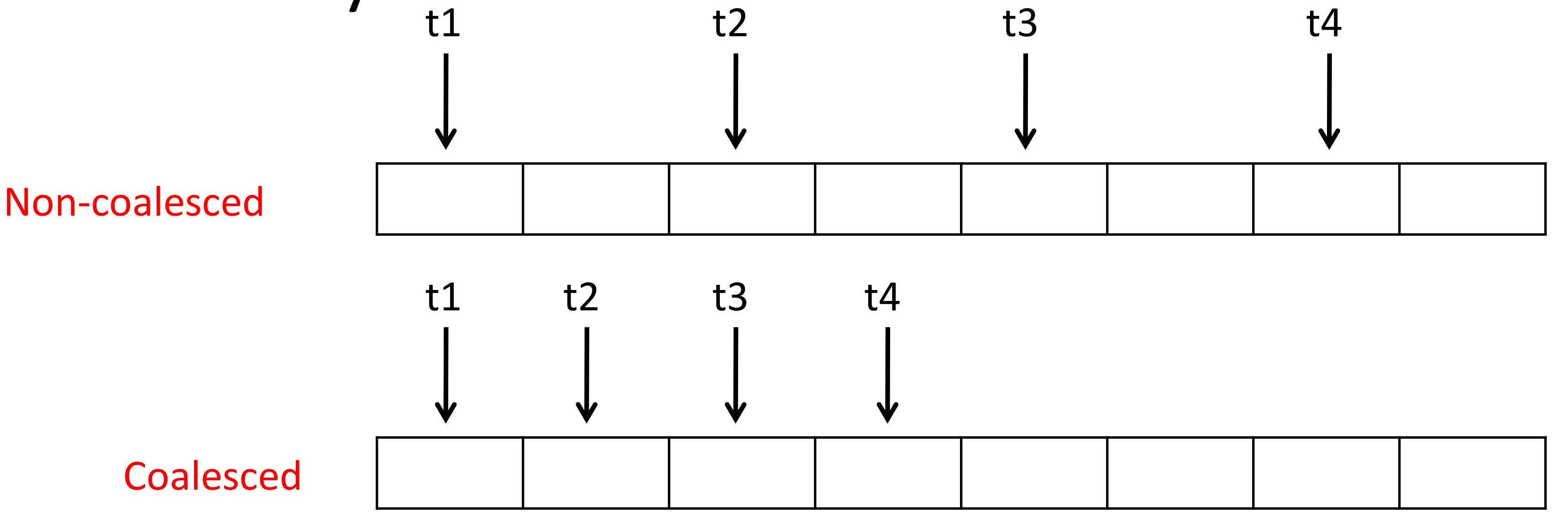
- Shall we still make one thread handle one element?
- Maybe not…
  - The numbers of blocks and threads for a kernel have a limit, e.g., up to 65535 blocks and 1024 threads per block.
  - A suitable number of threads should balance the degree of parallelism and resource usage.
- We may need to make each thread handle multiple elements for a large number of elements.

Two Parallelization Methods



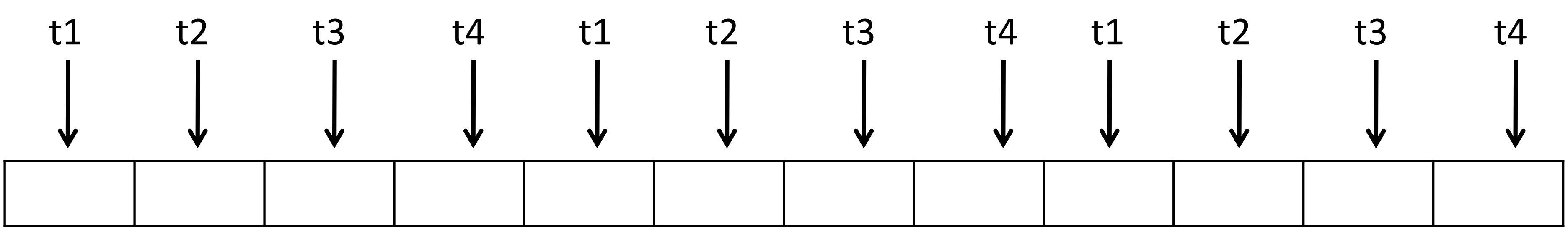
Coalesced Access

- If memory addresses accessed by threads in the same thread block are consecutive, then these memory accesses are grouped into one memory transaction.

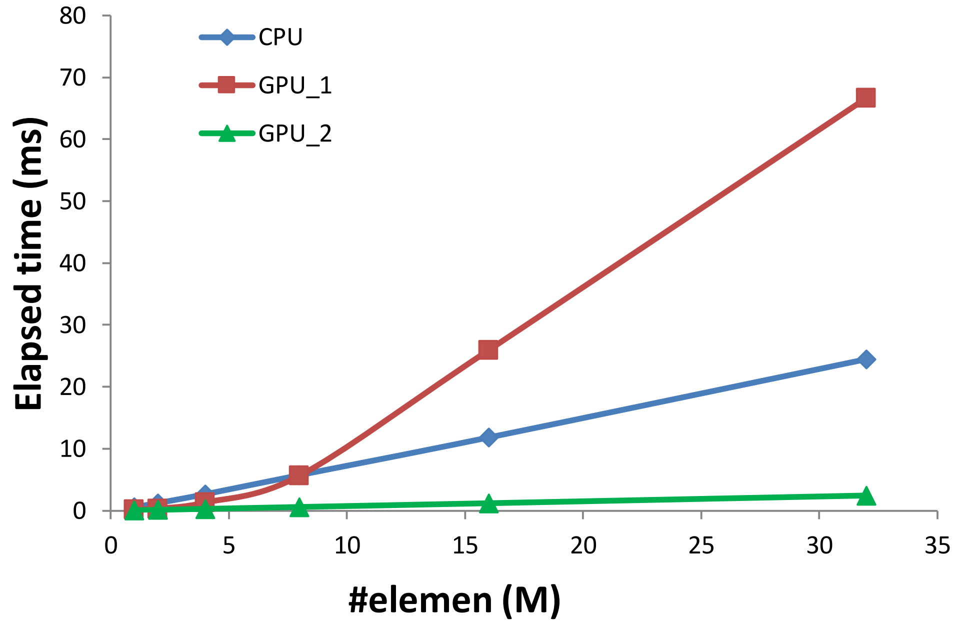


The GPU Kernel with Coalesced Access

```
__global__
void kernel2(int* d_data, const int numElement) {
        const int tid = blockDim.x*blockIdx.x + threadIdx.x;
        const int nthread = blockDim.x*gridDim.x;

        for(int i = tid; i < numElement; i += nthread) {
                d_data[i] += 1;
        }
}
```



Performance Comparison



1. Coalesced access is crucial for utilizing the GPU memory bandwidth.
2. A badly-written GPU program may be even slower than a CPU program!

Measure Kernel Execution Time

- Kernel execution is asynchronous.
- To measure the elapsed time of a kernel, we need a synchronization between the host and device.

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);
kernel1<<<1024, 512>>>(d_data);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
printf("Kernel elapsed time: %.3f ms\n", elapsedTime);
```

Summary

- A CUDA program consists of host and device code.
- The host code is in charge of GPU memory allocation, data transfer between the GPU and the CPU, and kernel launching.
- A kernel program is executed by every thread in a grid structure.
- Coalesced access effectively utilizes GPU memory bandwidth.