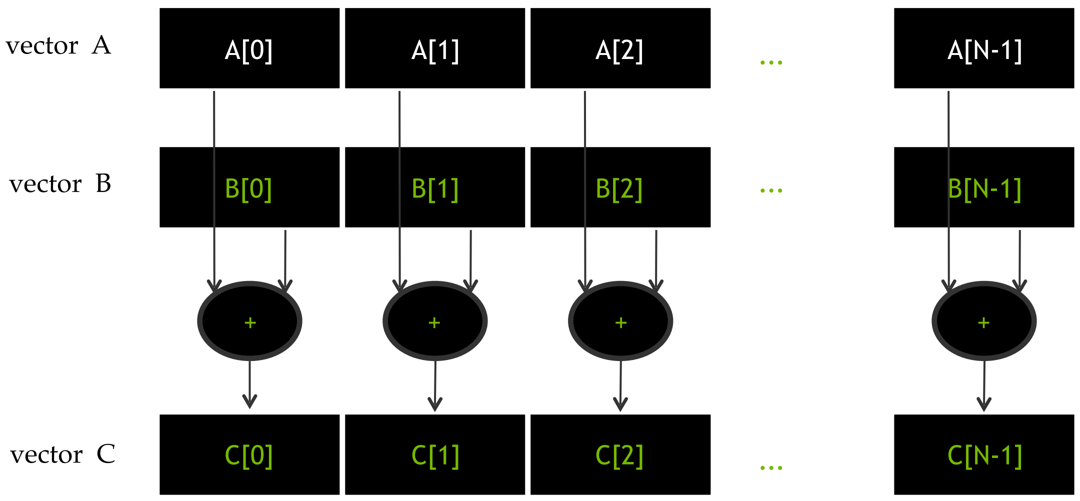
Introduction to CUDA C

Chapter 3 (2nd Edition)/Chapter 2 (3rd Edition)

Data Parallelism

- Re-organize computation around data: these computations can be run in parallel to complete the job.



GPU Programming Languages

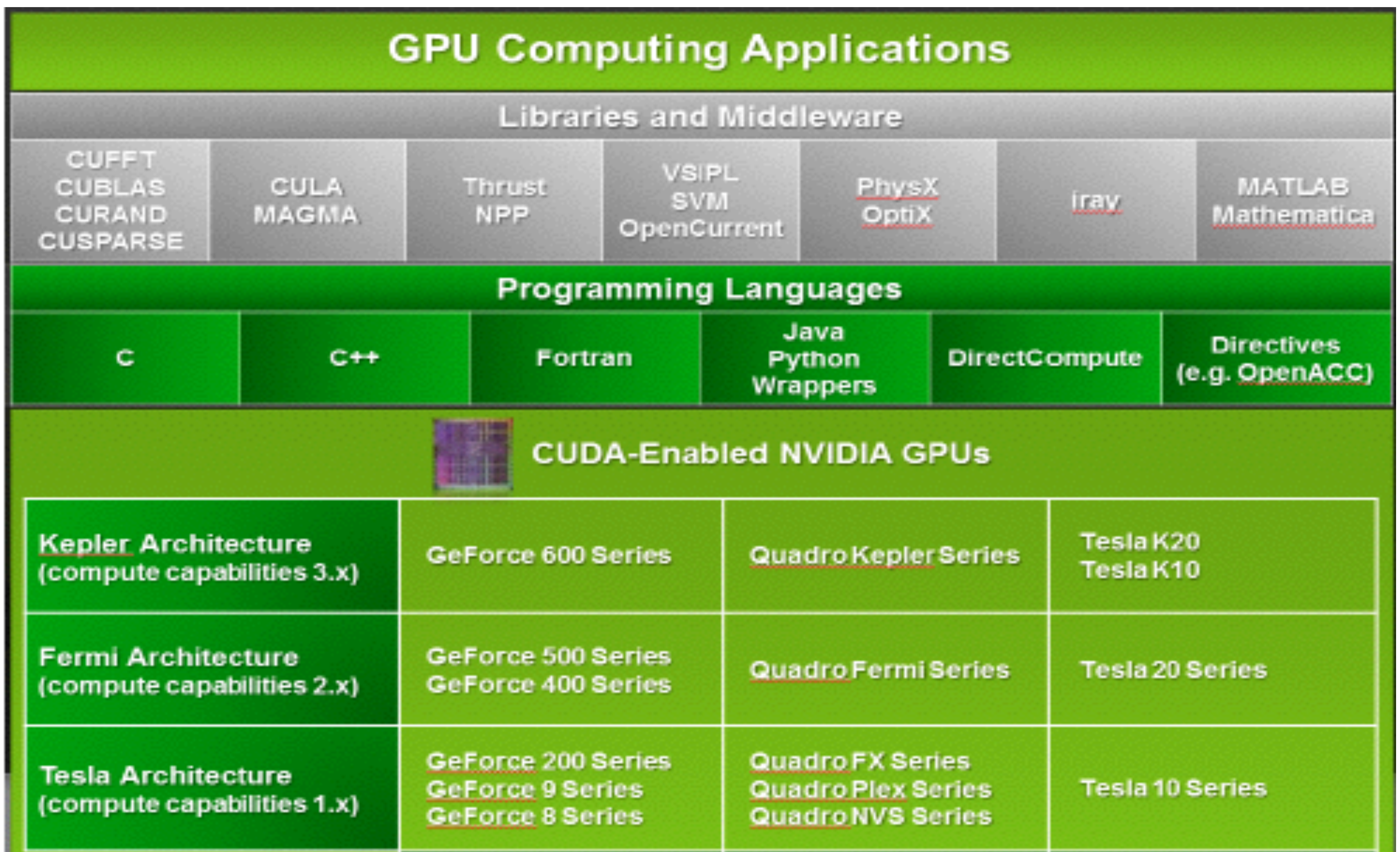
Numerical analytics	MATLAB, Mathematica, LabVIEW
Fortran	CUDA Fortran
C	CUDA C
C++	CUDA C++
Python	PyCUDA, Copperhead, Numba, NumbaPro
F#	Alea.cuBase

CUDA C

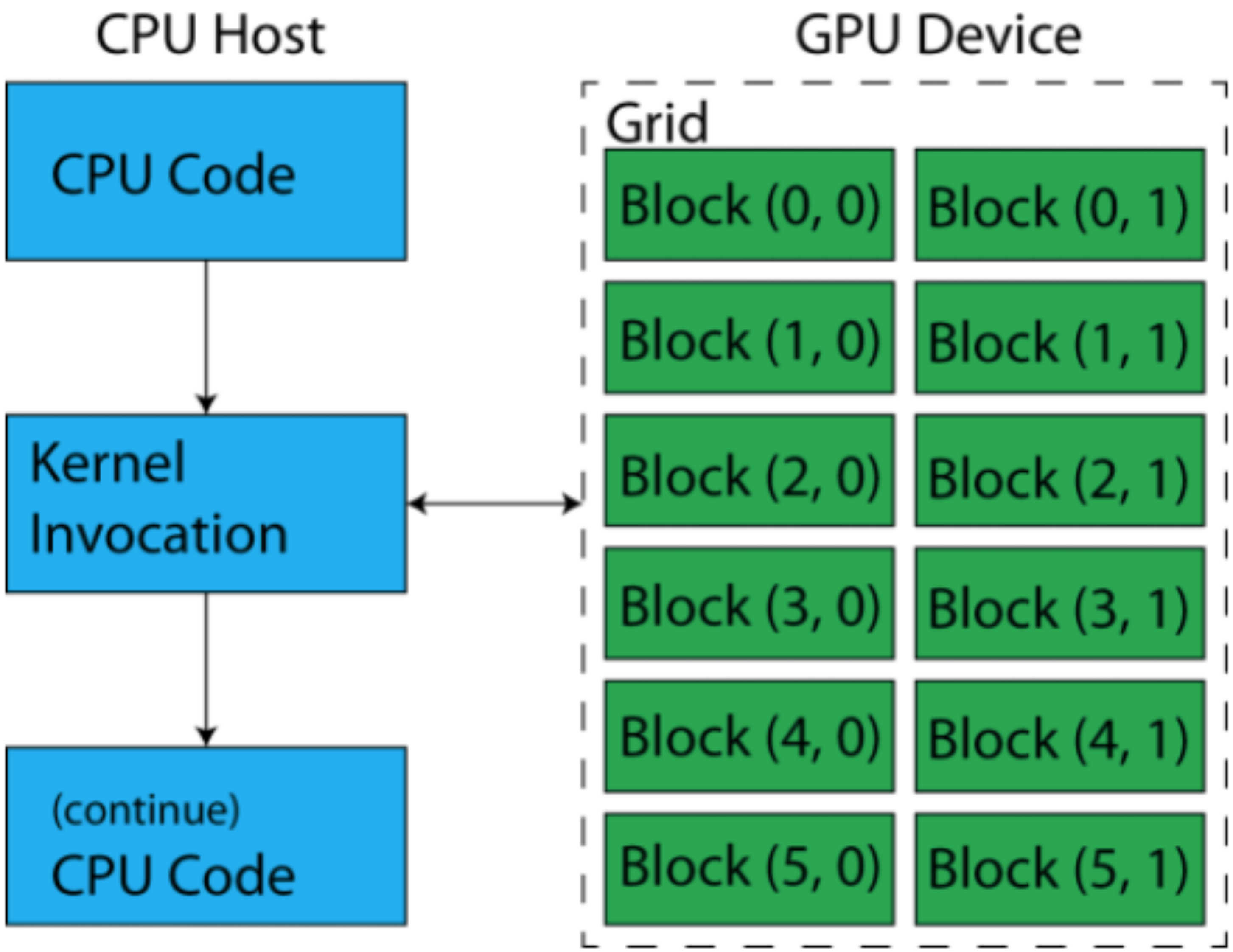
Easy to use
Most Performance

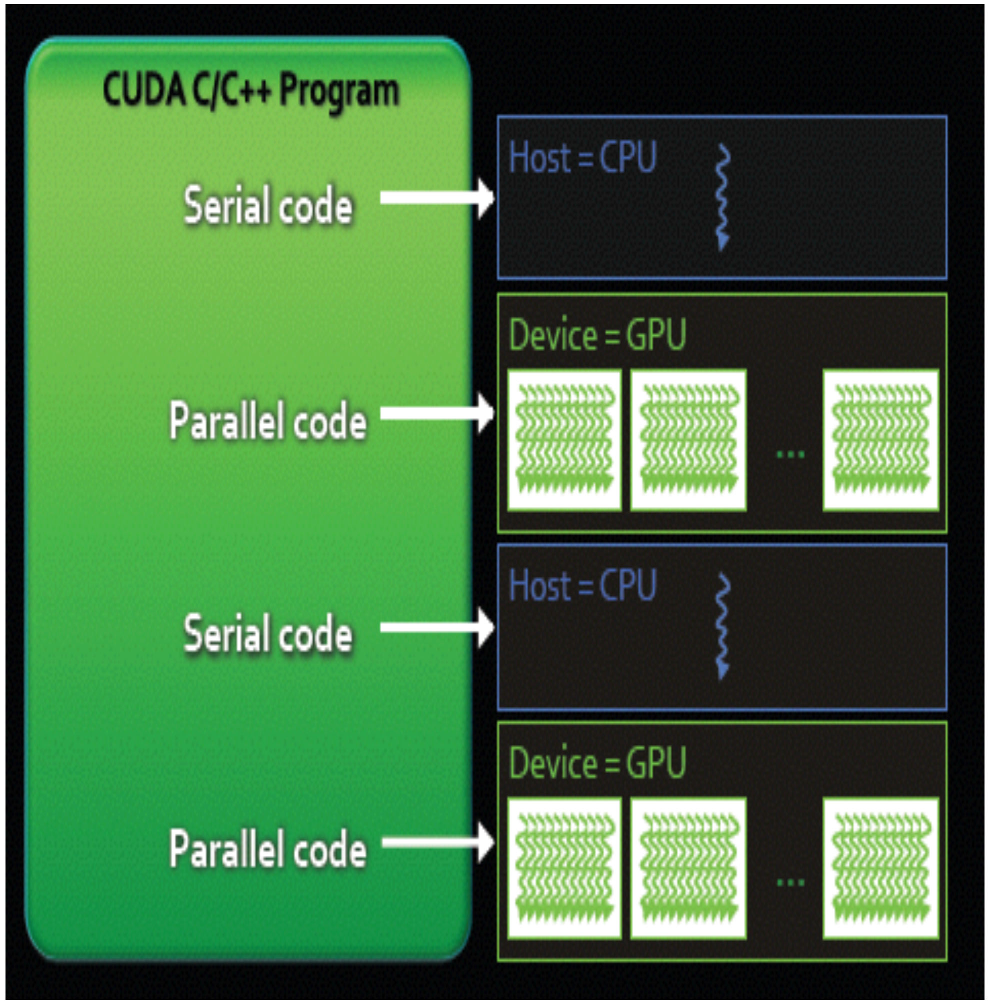
Easy to use
Portable code

Most Performance
Most Flexibility

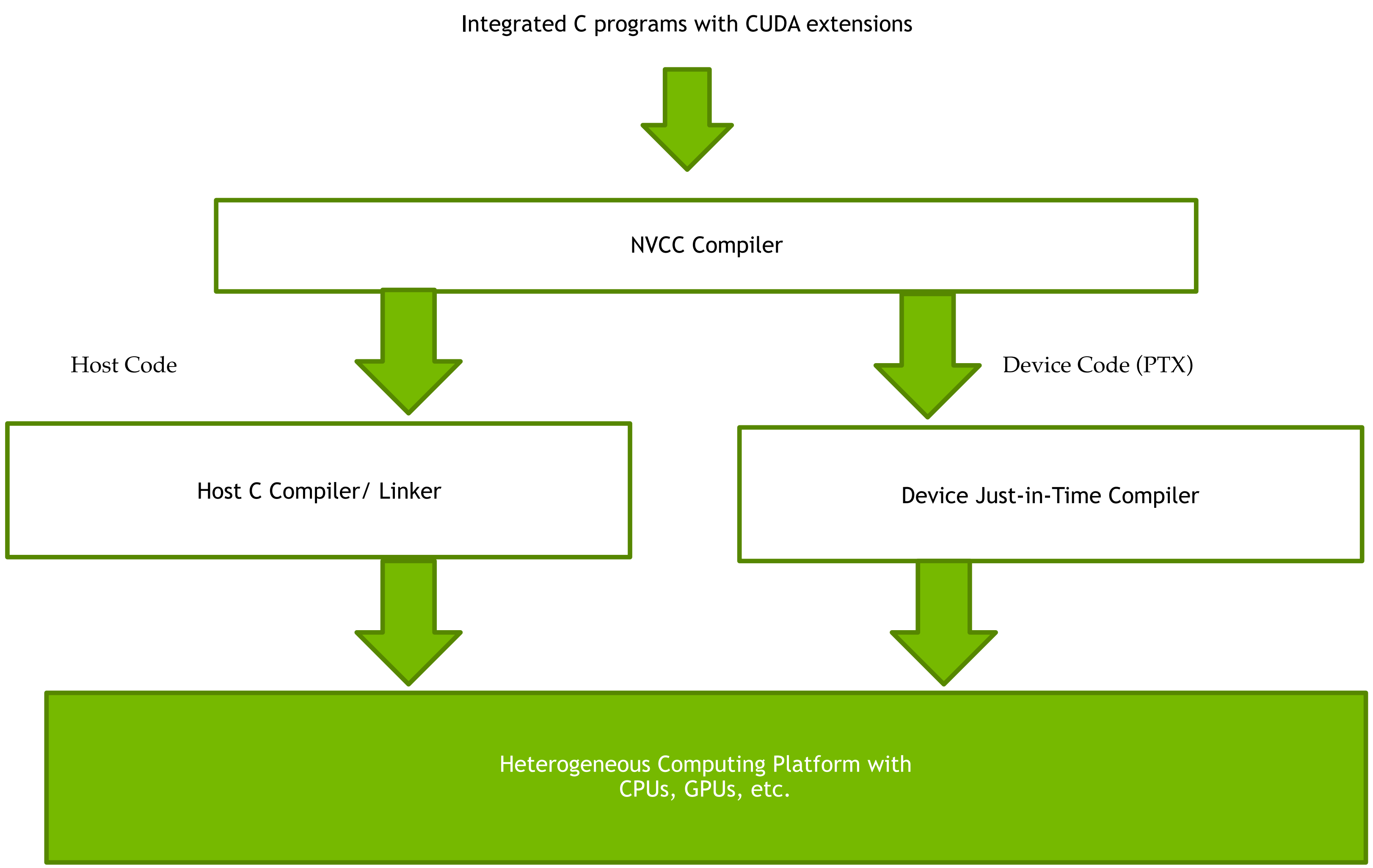


CUDA C Program Structure





Compilation Process Of CUDA Program

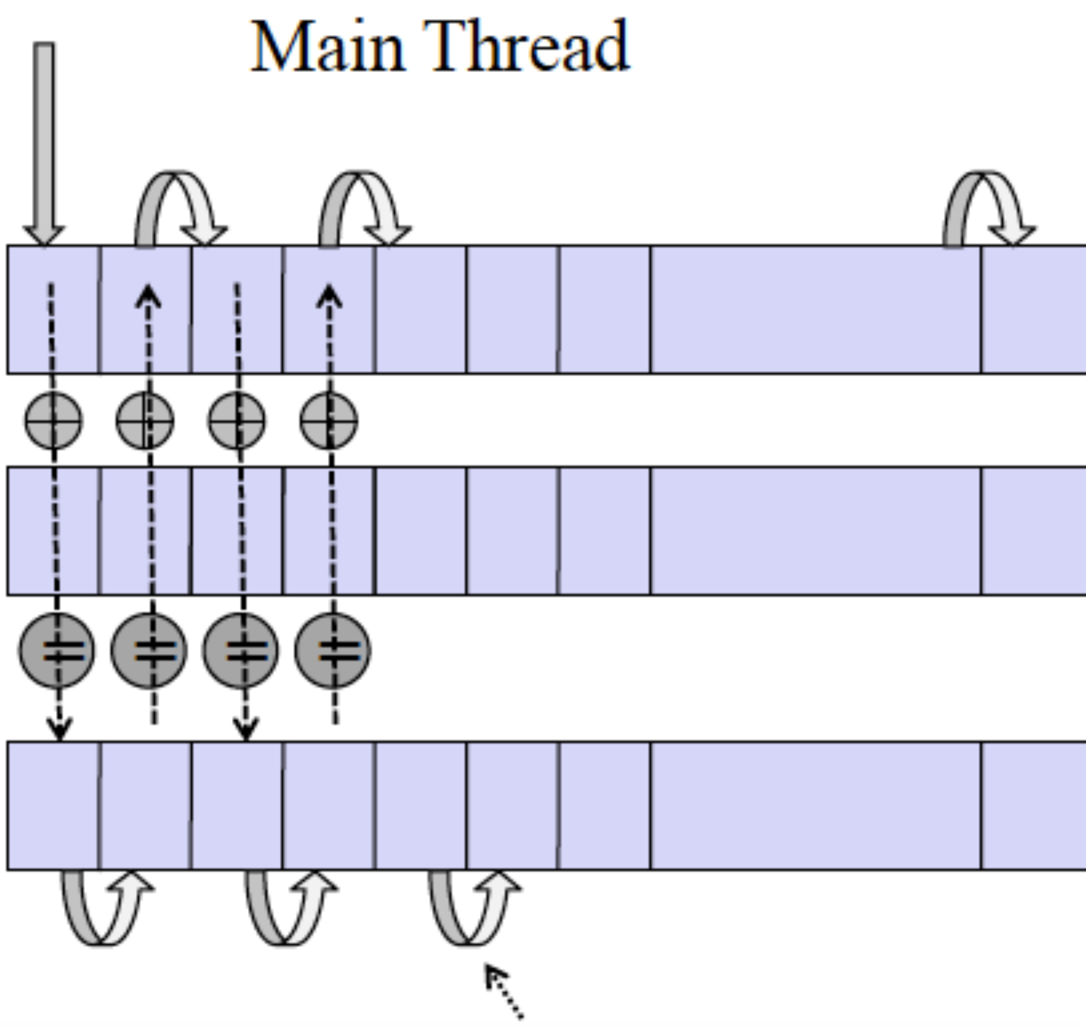


Hello World Program

```
#include <cstdio>
__global__ void mykernel()
{
  printf("Kernel");
}
int main() {
 printf("Hello World!\n");
  mykernel<<<2,5>>>();
  return 0;
}
```

A Vector Addition Serial Code

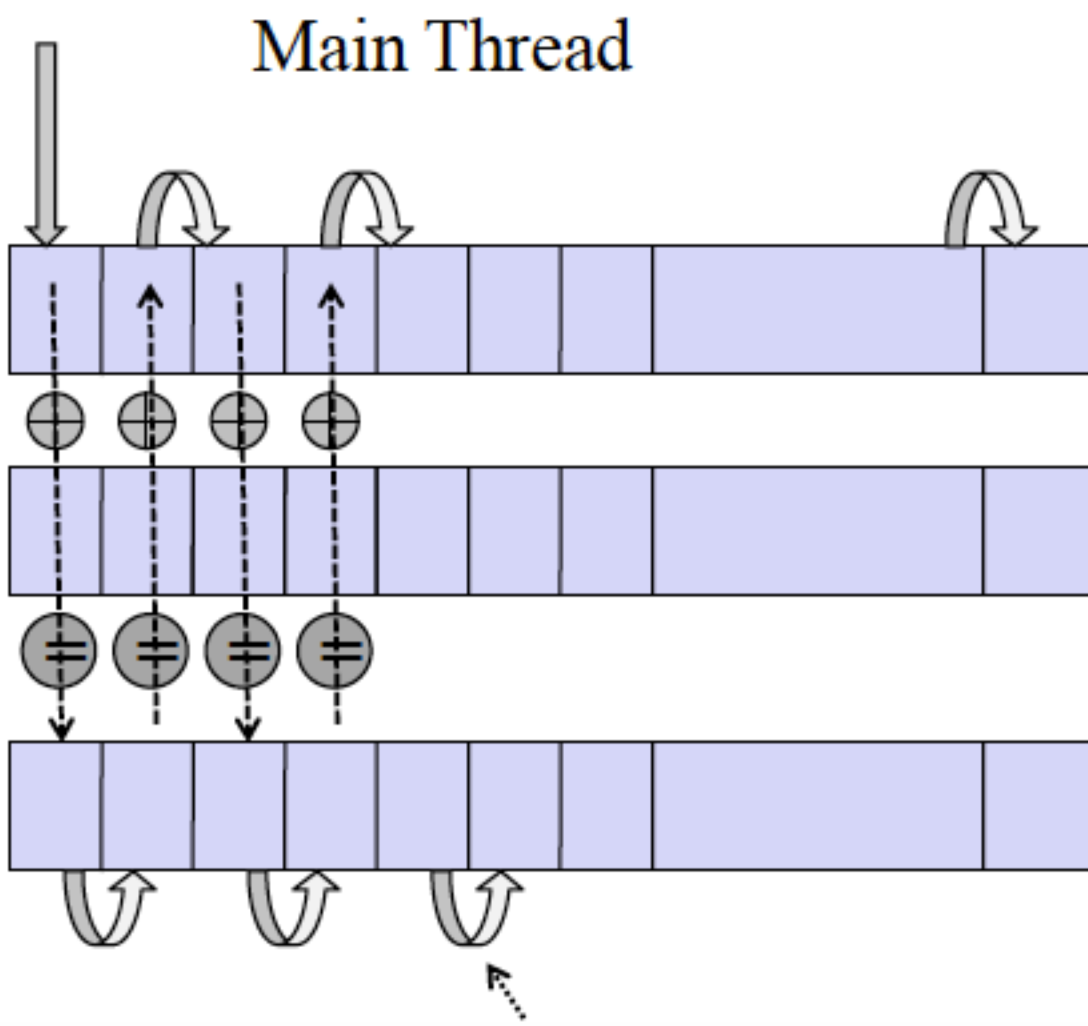
```
void add_vec(float* a, float *b, float* c, int N)
{
    int index;
   for(index=0;index<N;++index)
   {
     c[index]=a[index]+b[index];
   }
   }

int main()
{
   // Memory allocation for h_A, h_B, and h_C
    // I/O to read h_A and h_B, N elements
     …
   vecAdd(h_A, h_B, h_C, N);
}
```



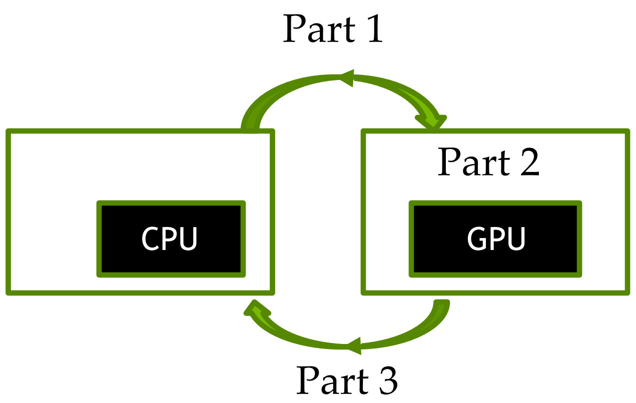
A Vector Addition Serial Code

```
void add_vec(float* a, float *b, float* c, int N)
{
    int index;
   for(index=0;index<N;++index)
   {
     c[index]=a[index]+b[index];
   }
   }
```

```
int main()
{
    // Memory allocation for h_A, h_B, and h_C
     // I/O to read h_A and h_B, N elements
     …
    vecAdd(h_A, h_B, h_C, N);
}
```

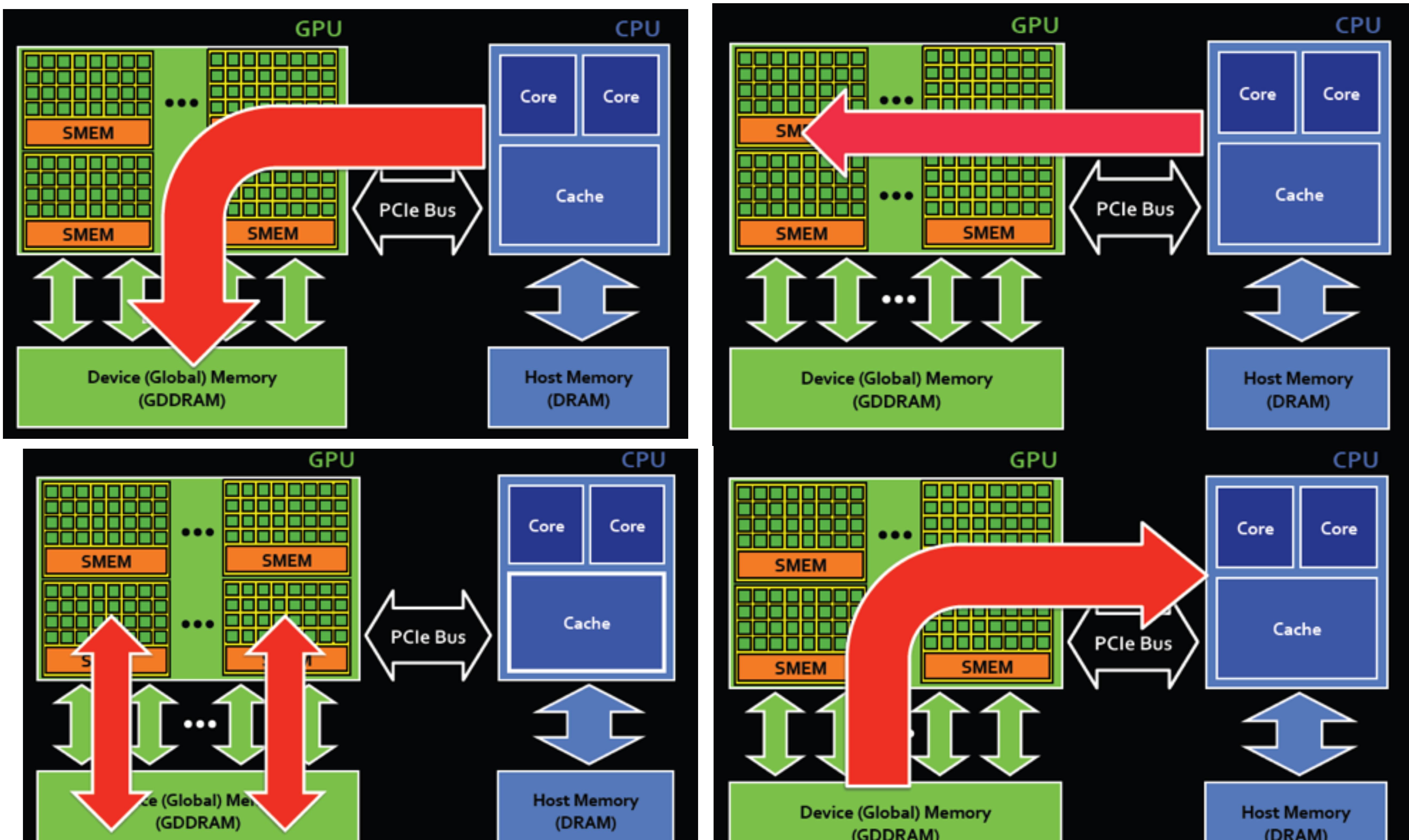


A Parallel Vector Addition (VERSION 1)

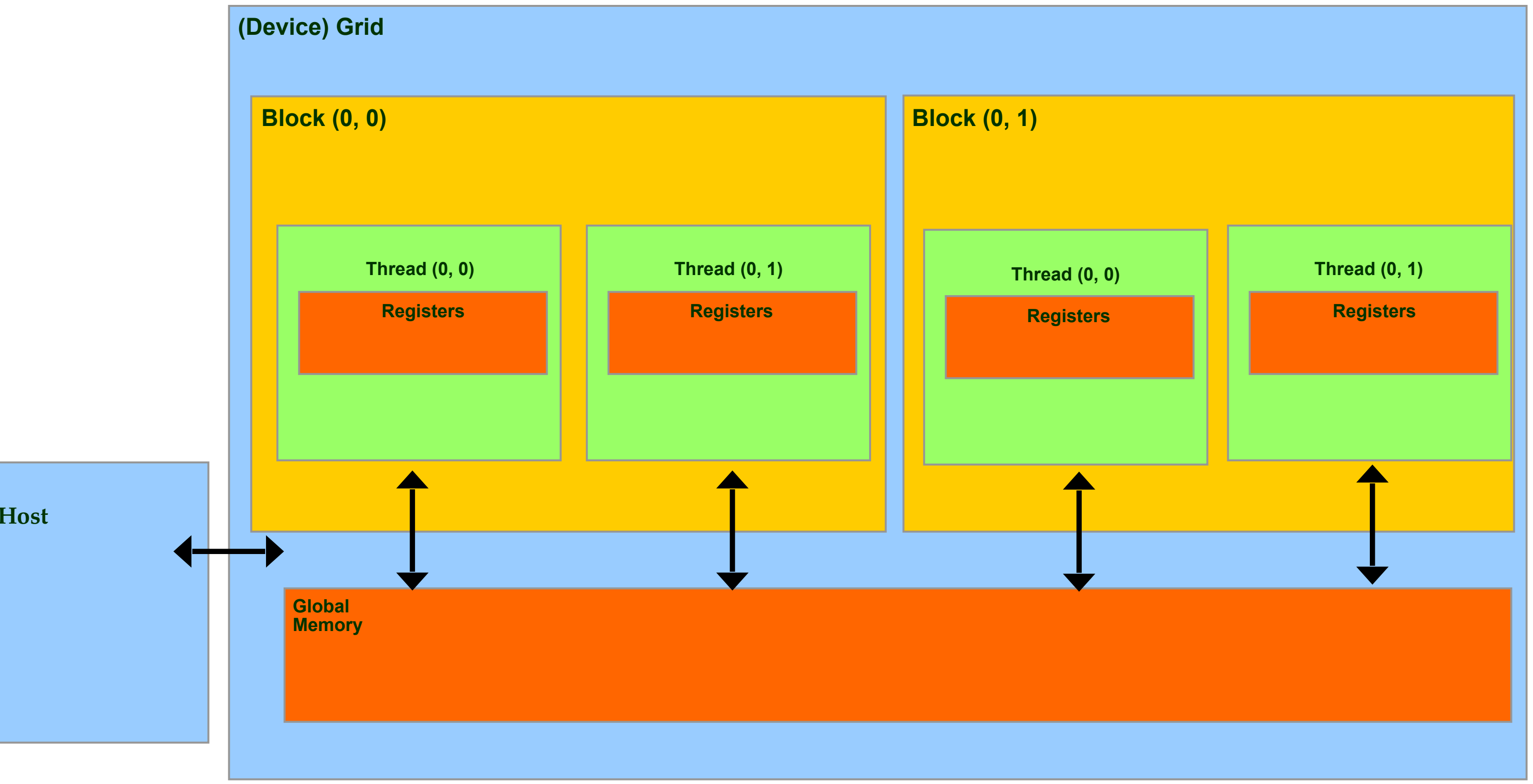



```
#include <cuda.h>
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
   int size = n* sizeof(float);
   float *d_A, *d_B, *d_C;
   // Part 1
   // Allocate device memory for A, B, and C
   // copy A and B to device memory

   // Part 2
   // Kernel launch code – the device performs the actual
   vector addition

   // Part 3
   // copy C from the device memory
   // Free device vectors
}
```

Typical CUDA Program Steps

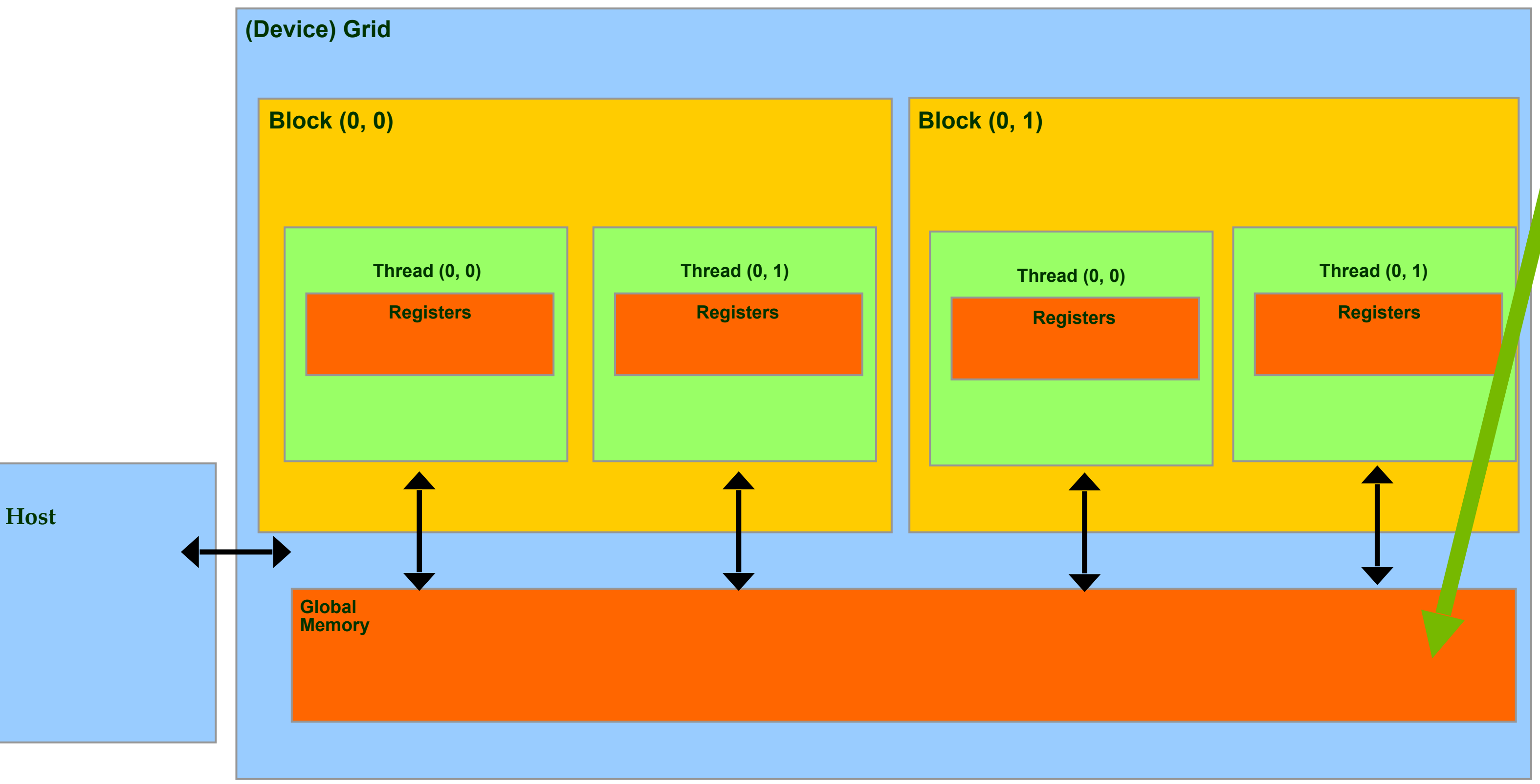


Partial Overview of CUDA Memories



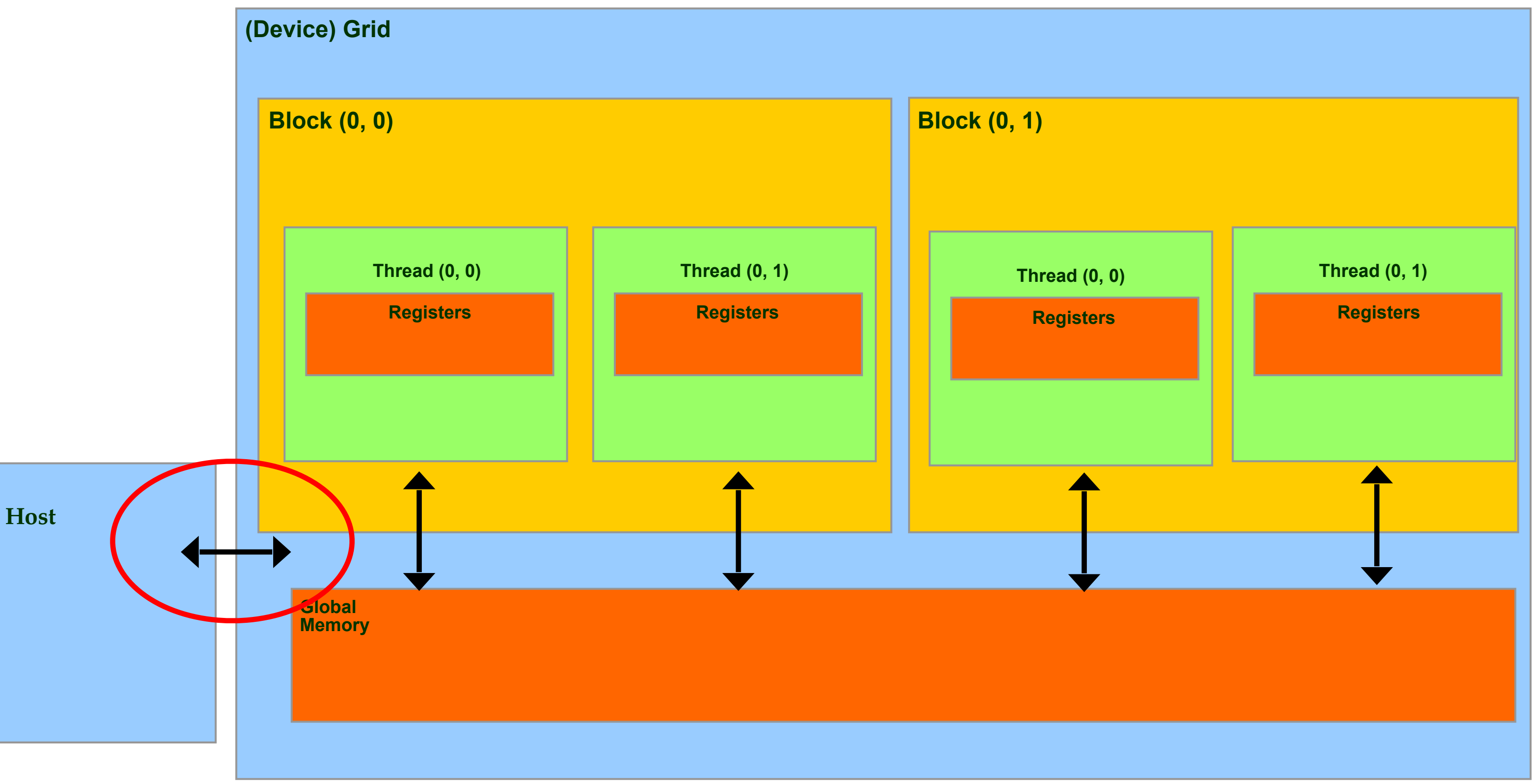
- Device code can:
  - R/W per-thread registers
  - R/W all-shared global memory
- Host code can
  - Transfer data to/ from per grid global memory

Device Global Memory & Data Transfers



- cudaMalloc()
- Allocates an object in the device global memory
- Two parameters
- **Address of a pointe**r to the allocated object
- **Size of** allocated object in terms of bytes
- cudaFree()
- Frees object from device global memory
- One parameter
- **Pointer** to freed object

Device Global Memory & Data Transfers…



- cudaMemcpy()
- memory data transfer
- Requires four parameters
- Pointer to destination
- Pointer to source
- Number of bytes copied
- Type/Direction of transfer

- Blocking API

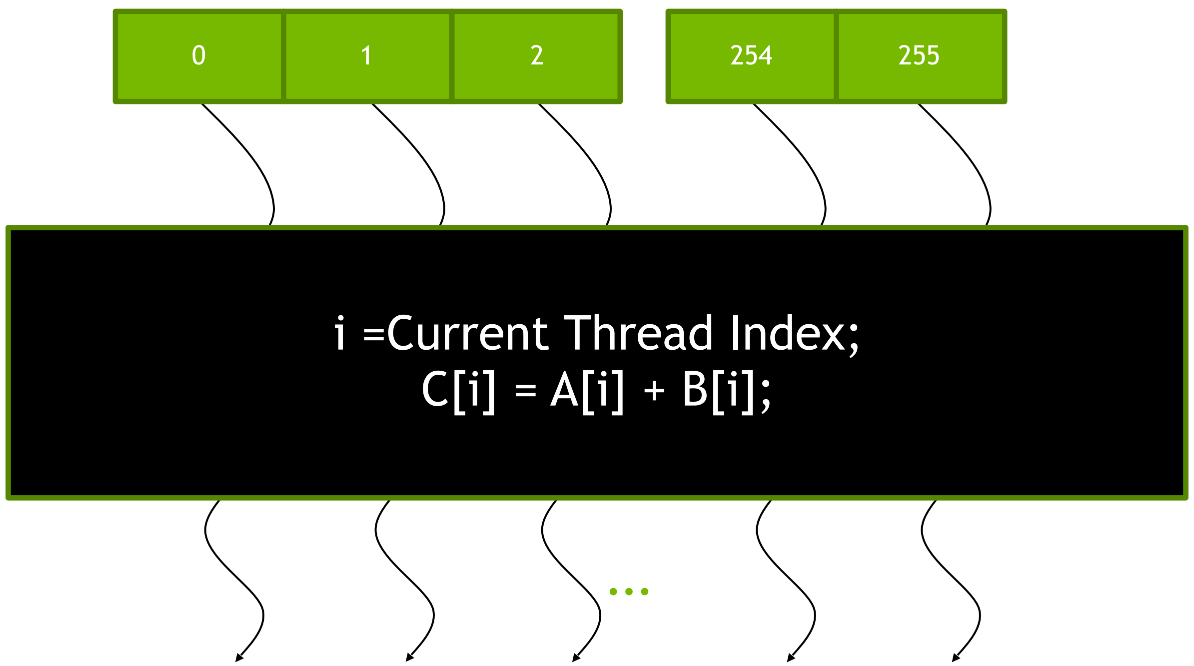
- Four symbolic predefined constants
  - cudaMemcpyHostToHost
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice
  - (cudaMemcpyDefault)

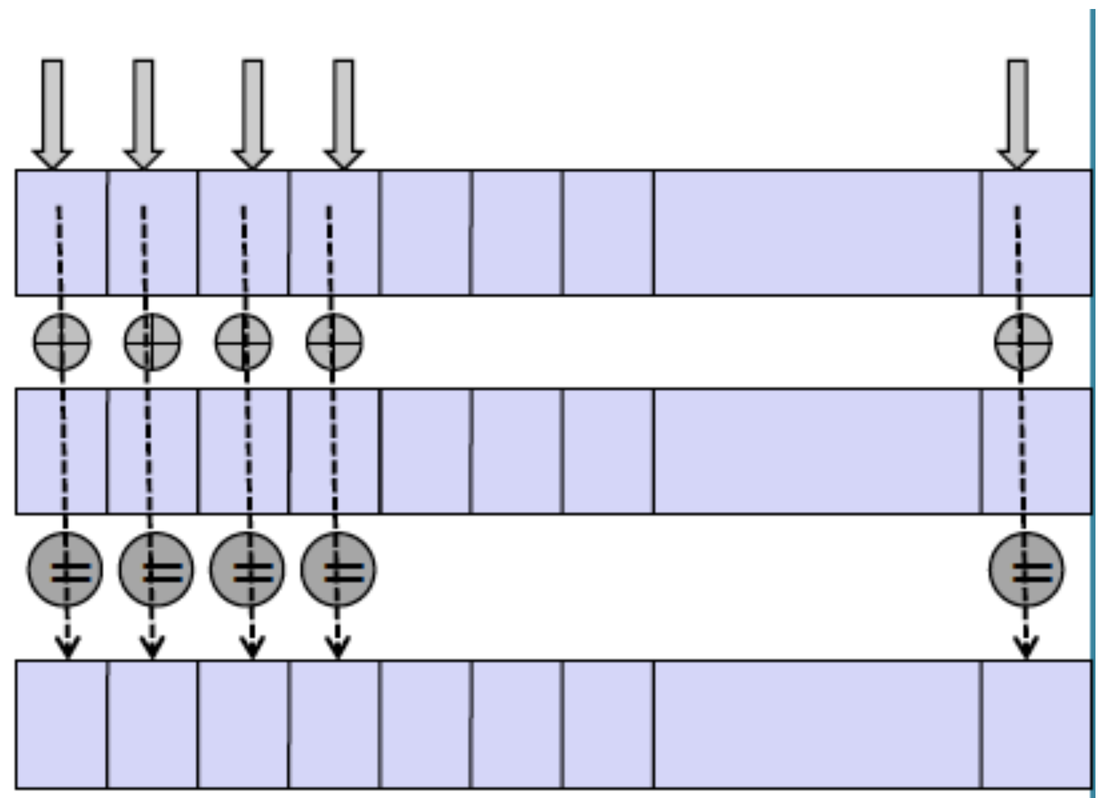
Parallel Vector Addition (VERSION 2

```
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
   int size = n * sizeof(float); float *d_A, *d_B, *d_C;

   cudaMalloc((void **) &d_A, size);
   cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_C, size);

    // Kernel invocation code – to be shown later

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

Kernel Functions & Threading

- A CUDA kernel is executed by a grid (array) of threads
  - All threads in a grid run the same kernel code (Single Program Multiple Data)
  - Each thread has indexes that it uses to compute memory addresses and make control decisions





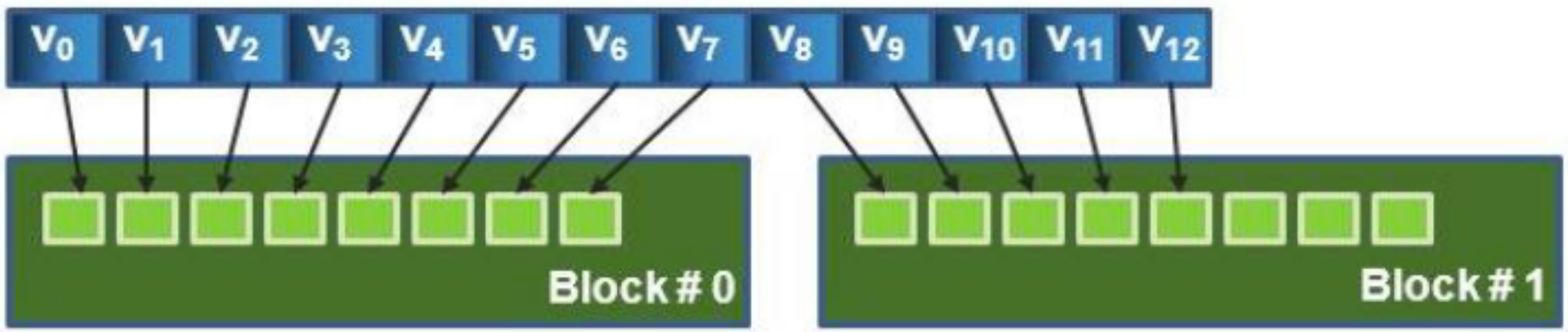
Kernel Functions & Threading…

```
__global__ void vecAddKernel(float* A, float* B, float* C, int n)
{
        int i = threadIdx.x;
        if(i < n)
                C[i] = A[i] + B[i];
}
```

	Executed on the :	**Only callable from the :**
__device__ float DeviceFunc()	**Device**	**Device**
__global__ void KernelFunc()	**Device**	**Host**
__host__ float HostFunc()	**Host**	**Host**

CUDA extensions to C functional declaration

Kernel Functions & Threading…




```
__global__ void vecAddKernel(float* A, float* B, float* C, int n)
{
        int i = threadIdx.x;
        if(i < n)
                C[i] = A[i] + B[i];
}
```

Kernel Launch

- When the host code invokes a kernel, it sets the grid and thread block dimensions via execution configuration parameters.
- Two struct variables of type dim3 are declared, The first is for describing the configuration of grid, the second variable describes the configuration of the block.
- Kernel launch statement provides the dimensions of the grid in terms of number of blocks and the dimensions of the blocks in terms of number of threads.

CUDA execution configuration parameters Examples

dim3 numberOfBlocks(8);

dim3 numberOfThreads(4);

Kernel call : gauss<<<numberOfBlocks,numberOfThreads>>>();

vecAddKernel<<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);

- If there are 1000 data elements, we launch ceil(1000/256.0) = 4 thread blocks
- It will launch 4 * 256(threads in each block) = 1024 threads
- Number of thread blocks depends on the length of the input data (***n***). For 1D vector,
  - If n = 750, 3 thread blocks
  - If n = 4000, 16 thread blocks

Parallel Vector Addition Final Version

```
#include<cuda.h>
void vecAdd(float* A, float* B, float* C, int n)
{
        int size = n * sizeof(float);
        float *d_A, *d_B, *d_C;
        cudaMalloc((void**) &d_A,  size);
        cudaMemcpy(d_A,  A, size, cudaMemcpyHostToDevice);
        cudaMalloc((void**) &d_B,  size);
        cudaMemcpy(d_B,  B, size, cudaMemcpyHostToDevice);
        cudaMalloc((void**) &d_C,  size);
        vecAddKernel<<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);
        cudaMemcpy(C,  d_C, size, cudaMemcpyDeviceToHost);
        //Free device memory
        cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}

__global__ void vecAddKernel(float* A, float* B, float* C, int N)
{
        int i = blockDim.x * blockIdx.x+ threadIdx.x;
        if(i < N)
                C[i] = A[i] + b[i];
}
```

Error Handling in CUDA

- CUDA API functions return flags that indicate whether an error has occurred when they served the request
- Most errors are due to inappropriate argument values
- Every CUDA call (except kernel launches) return an error code of type **cudaError_t** (enum that contains all possible error codes)
- No error = "**cudaSuccess**" otherwise an error code.
- Example:
  - **cudaError_t   cudaMalloc(…);**
  - returns **cudaSuccess** or **cudaErrorMemoryAllocation**

Error Handling in CUDA…

- Human-readable error obtained from:

```
char*  cudaGetErrorString(cudaError_t  error);
```

  - returns an error message string that can then be printed out.

- A call to cudaMalloc():

```
cudaMalloc((void**) &d_A,  size);
```

- Surround the call with code that tests for error conditions and prints out error messages

```
cudaError_t err  = cudaMalloc((void**) &d_A,  size);
if (err != cudaSuccess) {
    printf("%s in %s at line %d\n", cudaGetErrorString(err), __FILE__, __LINE__);
exit(EXIT_FAILURE);
}
```

Error Handling in CUDA…

- Use macro substitution:

```
#define HANDLE_ERROR( err ) (HandleError( err, __FILE__, __LINE__ ))

static void HandleError( cudaError_t err, const char *file, int line ) {
    if (err != cudaSuccess) {
        printf( "%s in %s at line %d\n", cudaGetErrorString( err ), file, line );
        exit( EXIT_FAILURE );
    }
}
```

- which works with any CUDA call that returns an error code:

```
HANDLE_ERROR( cudaMalloc( … ) );
```

Error Handling in CUDA…

Possible return values from CUDA APIs

cudaSuccess

cudaErrorMemoryAllocation

cudaErrorInitializationError

cudaErrorLaunchFailure

cudaErrorInvalidDevice

cudaErrorInvalidValue

cudaErrorInvalidHostPointer

cudaErrorInvalidDevicePointer

cudaErrorInvalidMemcpyDirection

cudaErrorStartupFailure

cudaErrorDevicesUnavailable

cudaErrorDuplicateVariableName

……..

Demo

- Vector Addition
- Vector Addition with Error Handling
- PyCUDA Vector Addition