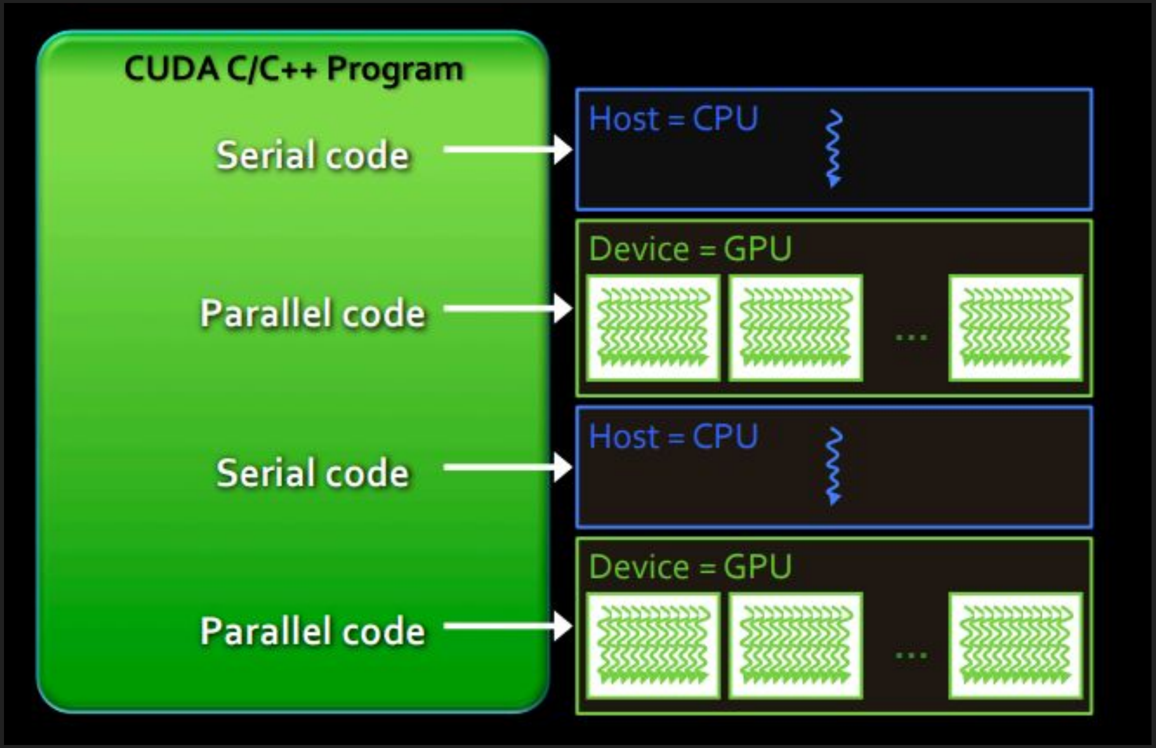
Introduction to CUDA

CUDA

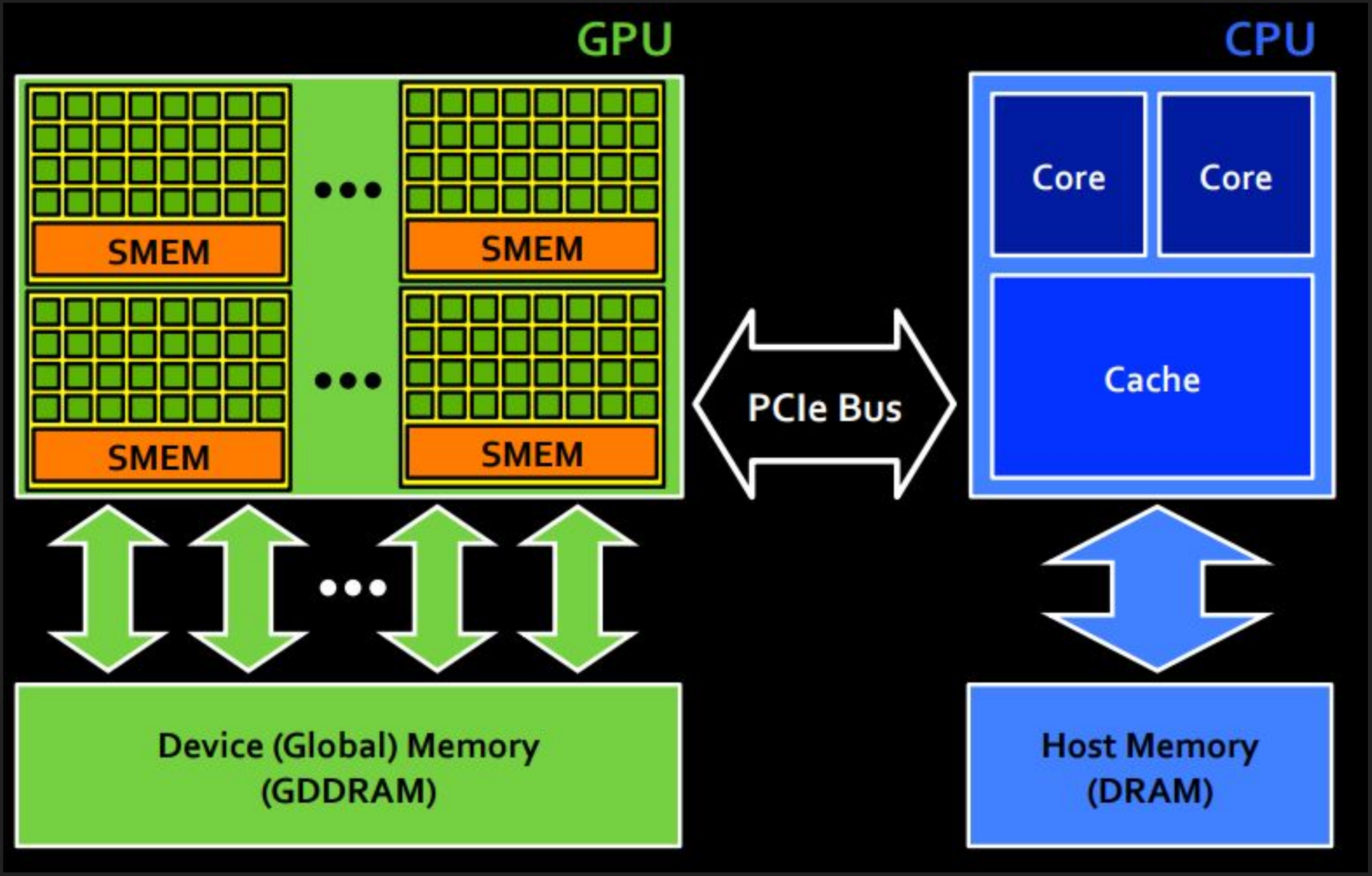
- Programing system for machines with GPUs
  - Programming Language
  - Compilers
  - Runtime Environments
  - Drivers
  - Hardware

Behavior of CUDA program

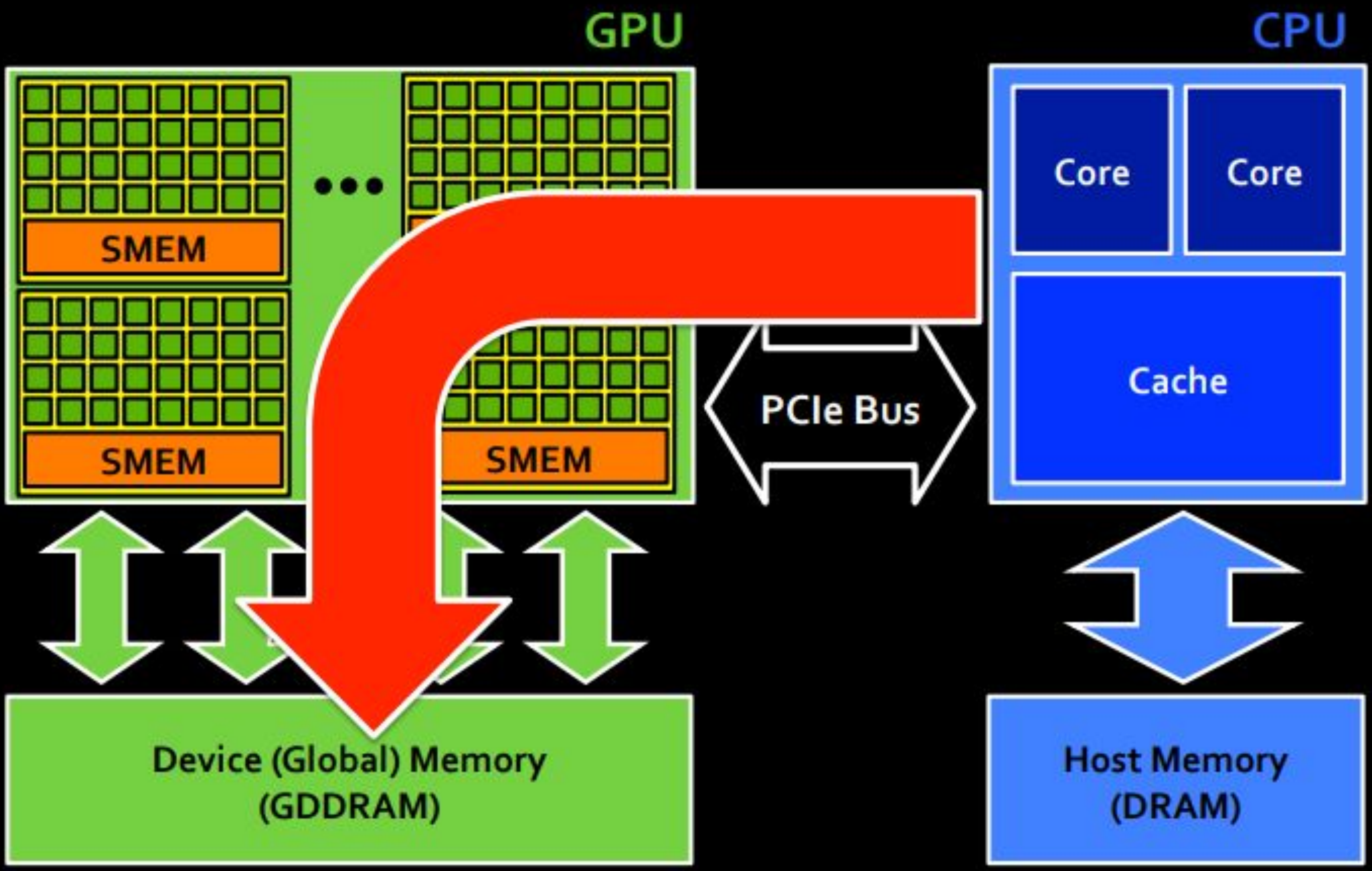
- Serial code executes in Host (CPU) thread
- Parallel code executes in many concurrent Device (GPU) threads across multiple parallel processing elements



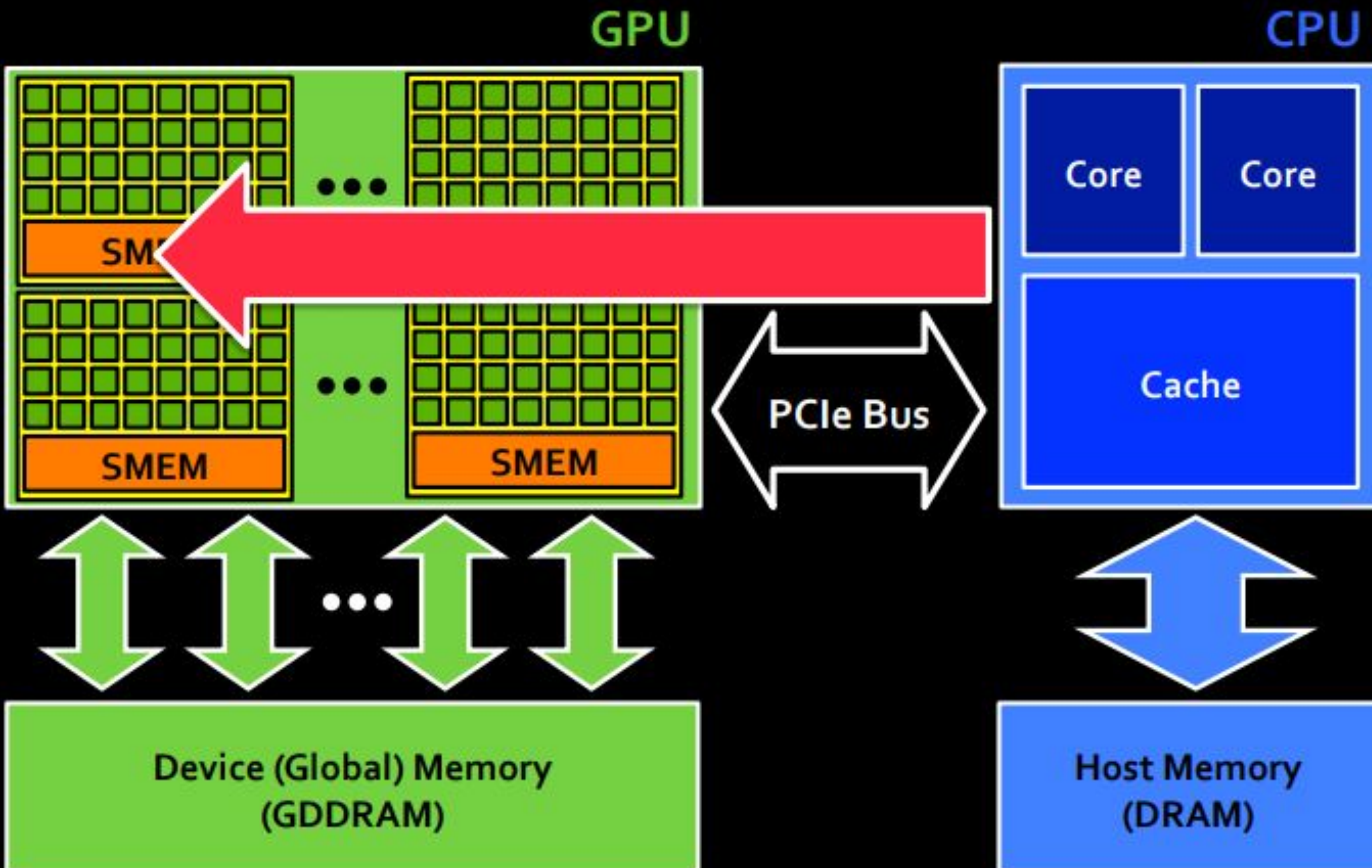
Execution flow



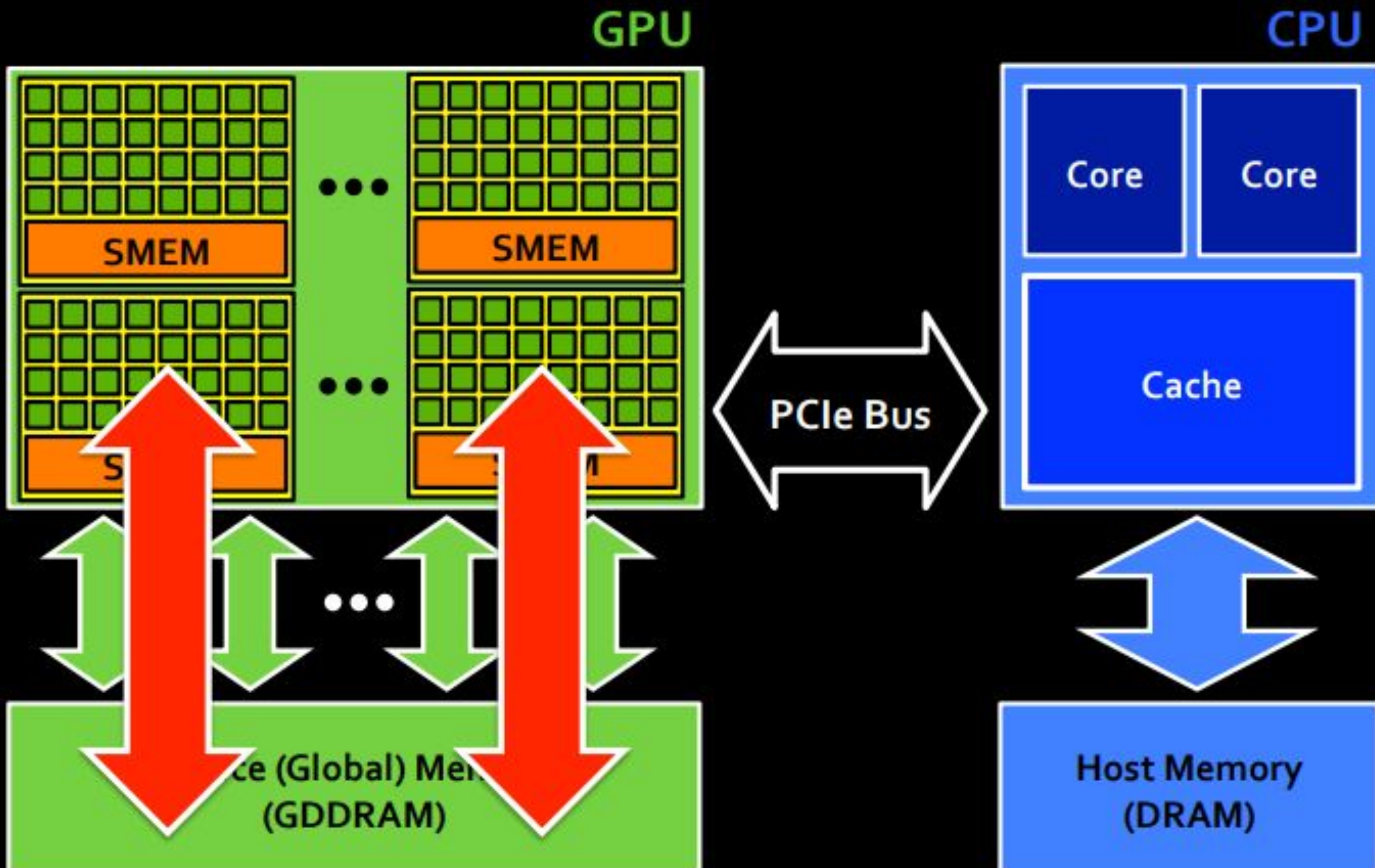
Step 1 – copy data to GPU memory



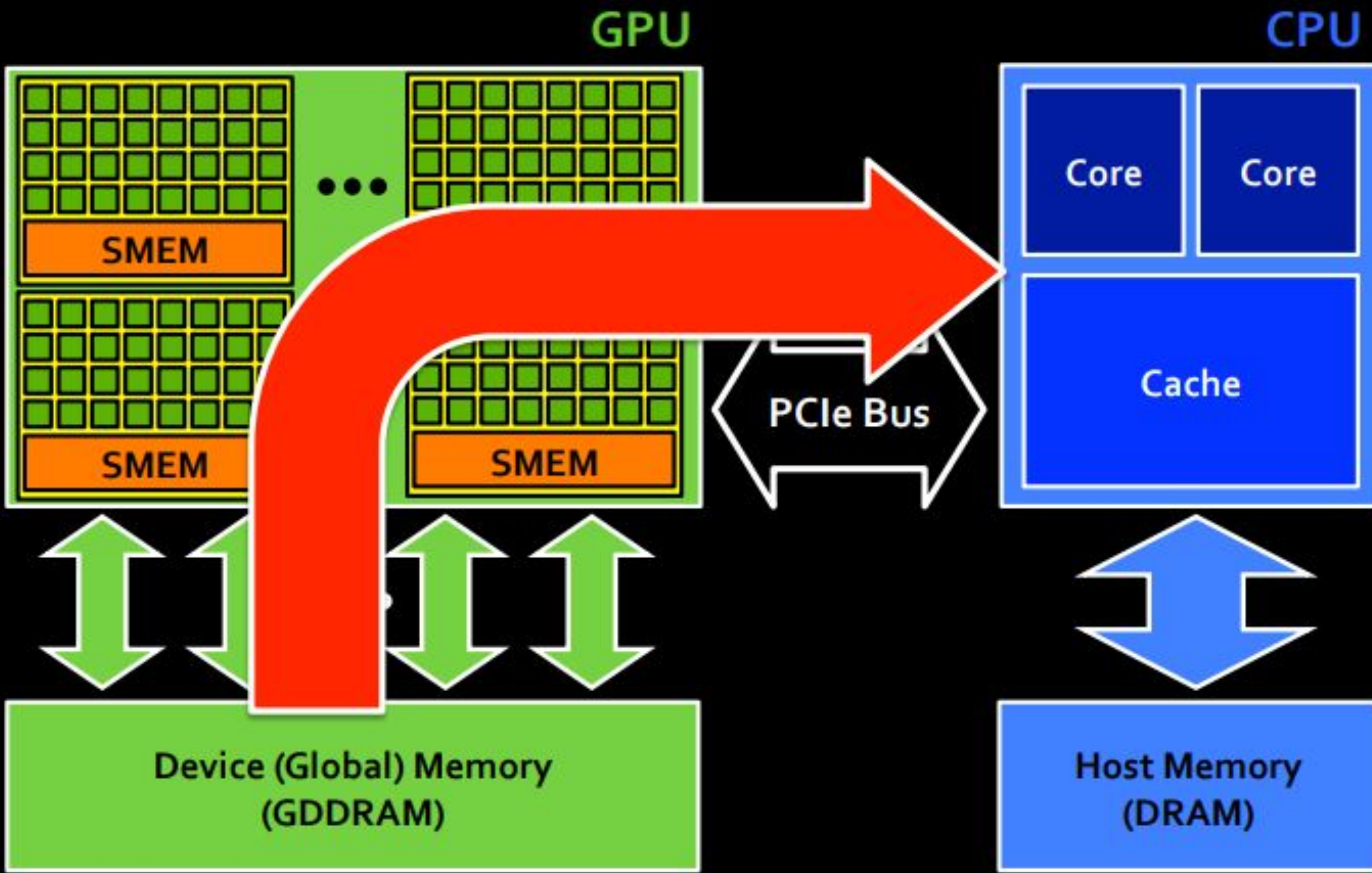
Step 2 – launch kernel on GPU



Step 3 – execute kernel on GPU



Step 4 – copy data to CPU memory

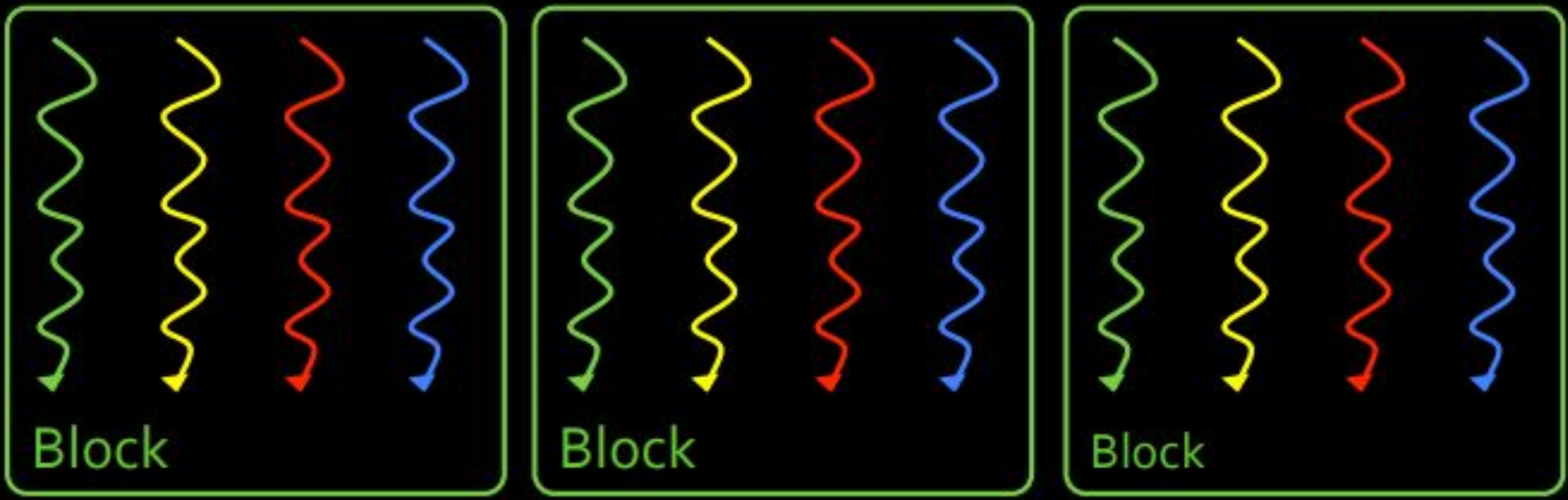


CUDA ARCHITECTURE

CUDA Thread Organization

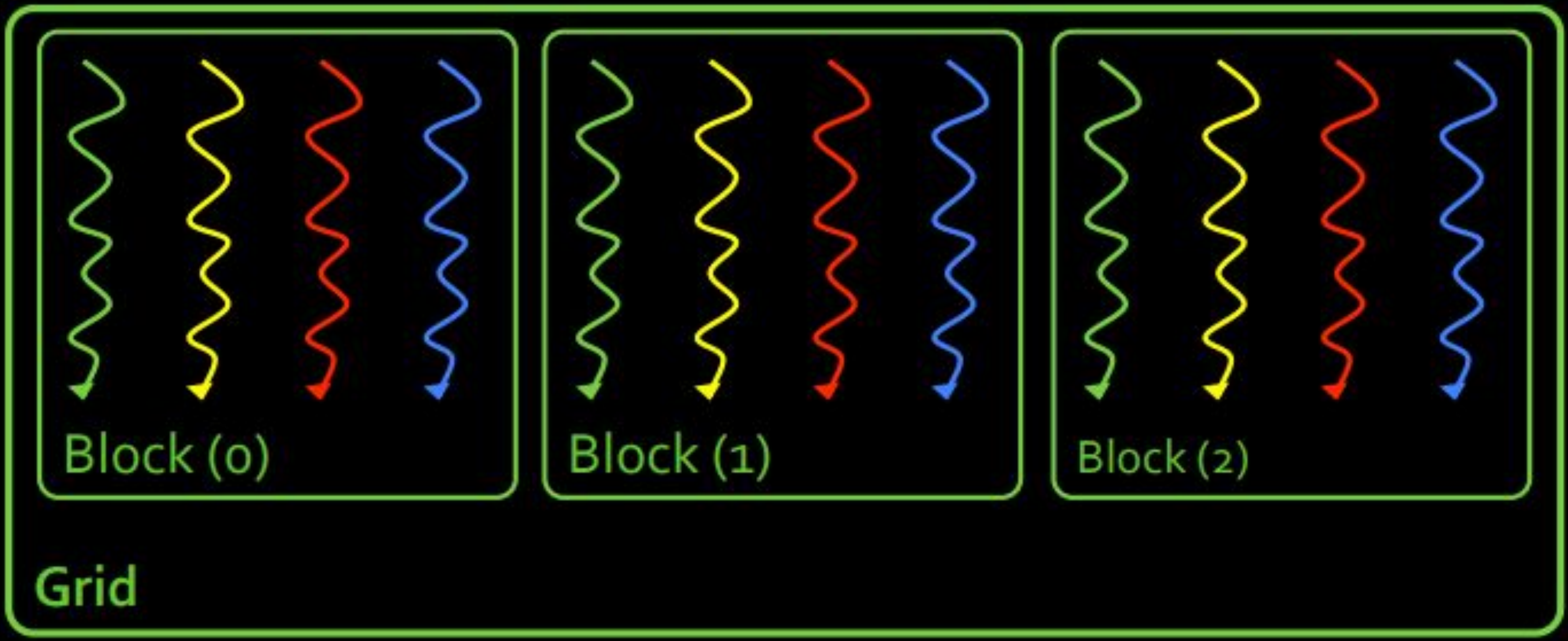
- GPUs can handle thousands of concurrent threads
- CUDA programming model supports even more
  - Allows a kernel launch to specify more threads than the GPU can execute concurrently
  - Helps to amortize kernel launch times

Blocks of threads



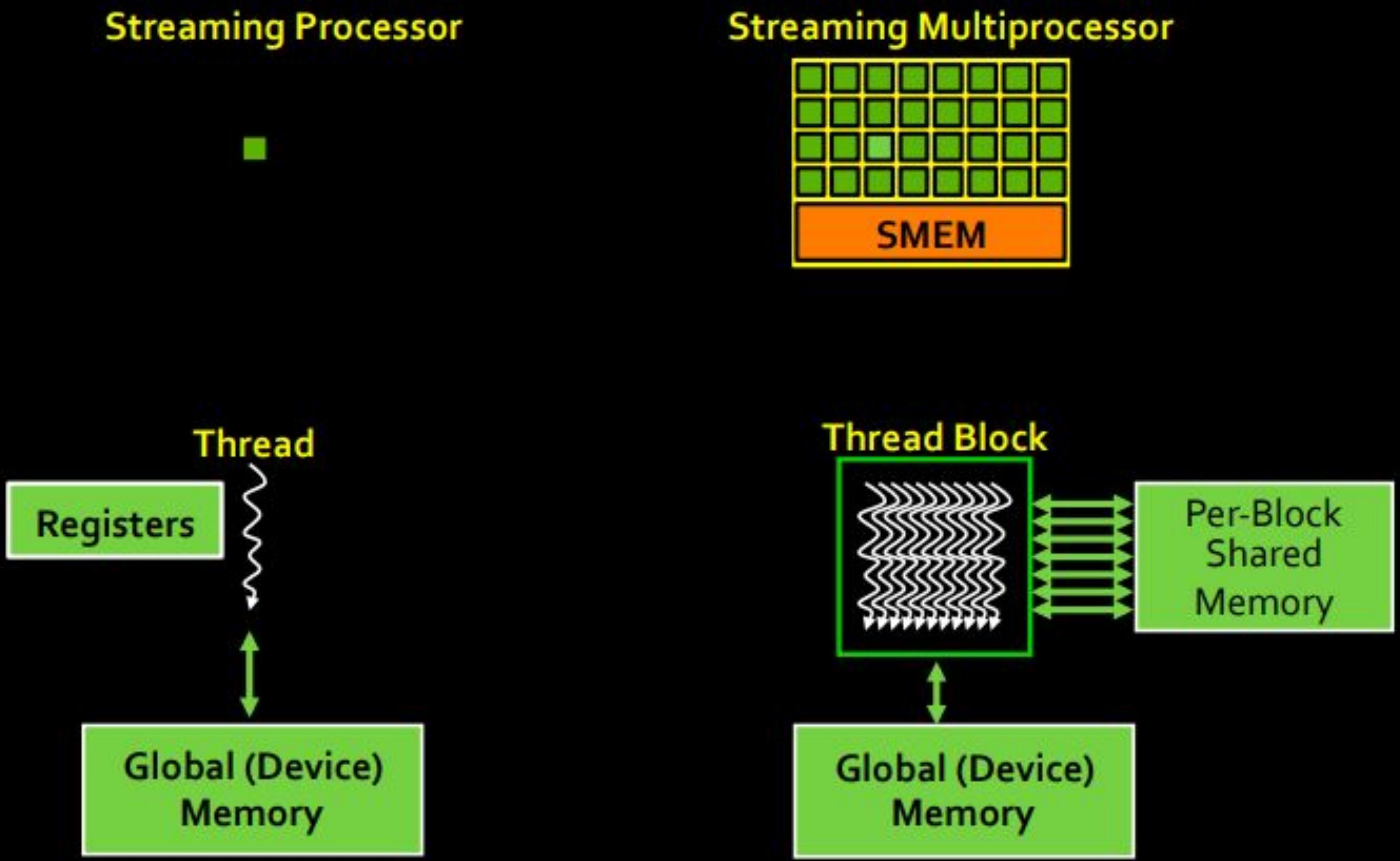
- Threads are grouped into **blocks**

Grids of blocks

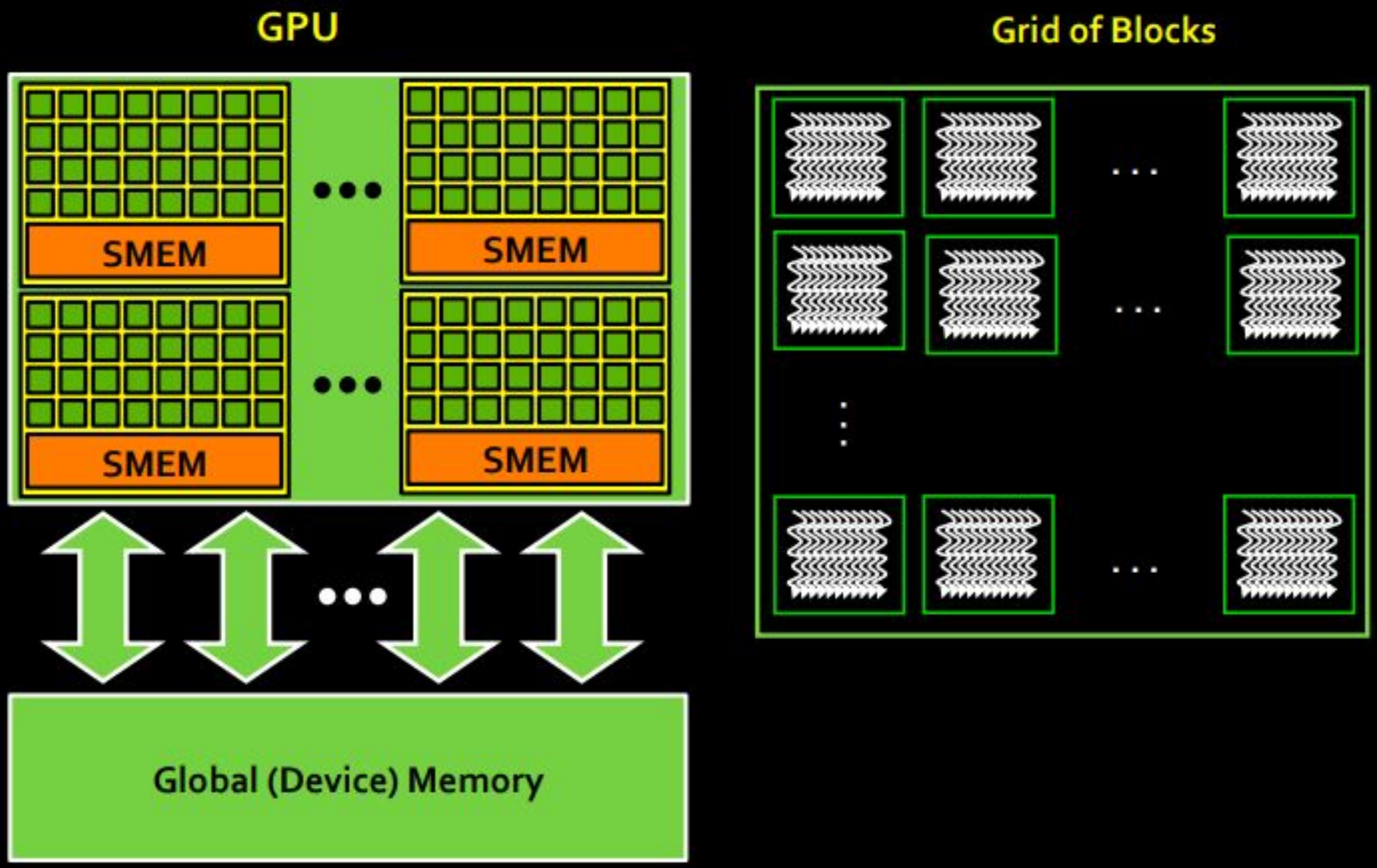


- Threads are grouped into blocks
- Blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads

Blocks execute on Streaming Multiprocessors

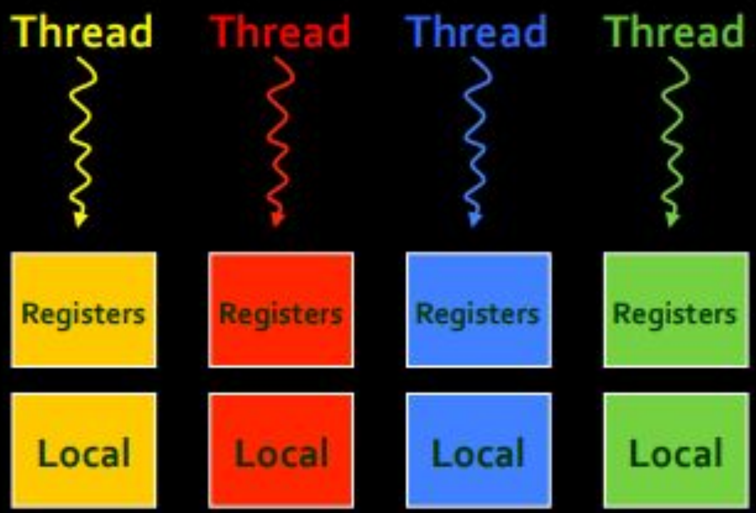


Grids of blocks executes across GPU



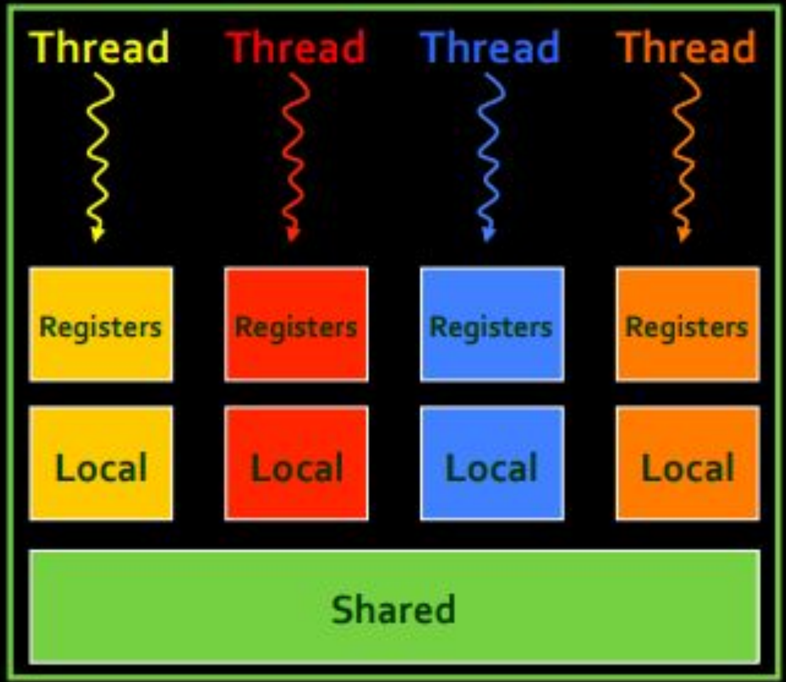
CUDA Memory Hierarchy

- Thread
  - Registers
  - Local memory



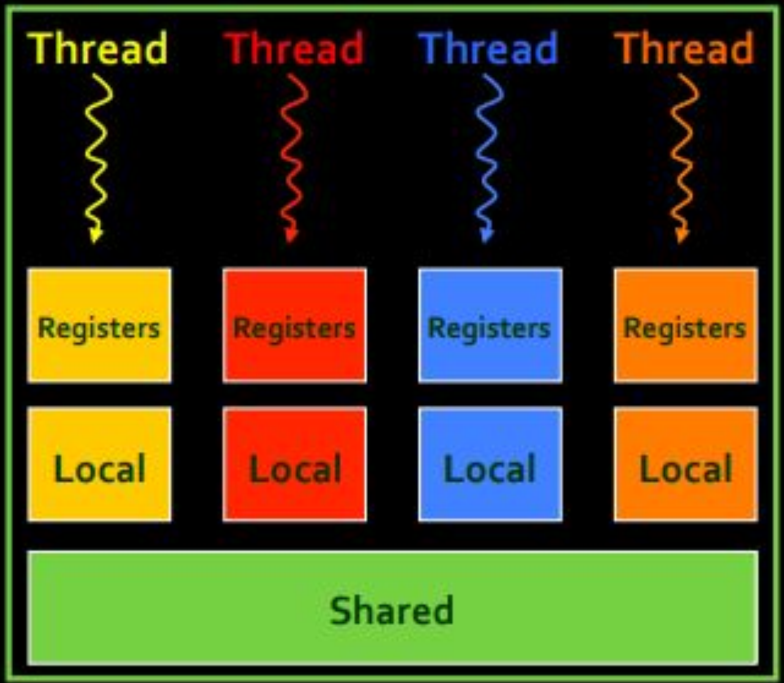
CUDA Memory Hierarchy

- Thread
  - Registers
  - Local memory
- Thread Block
  - Shared memory



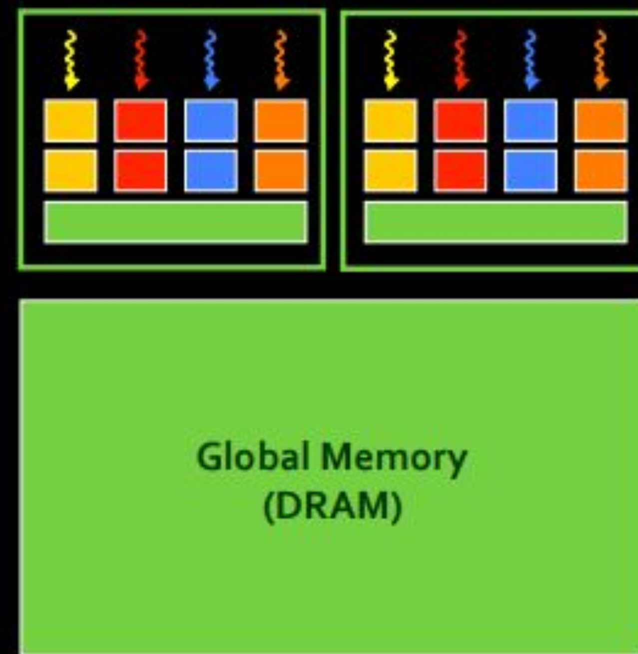
CUDA Memory Hierarchy

- Thread
  - Registers
  - Local memory
- Thread Block
  - Shared memory



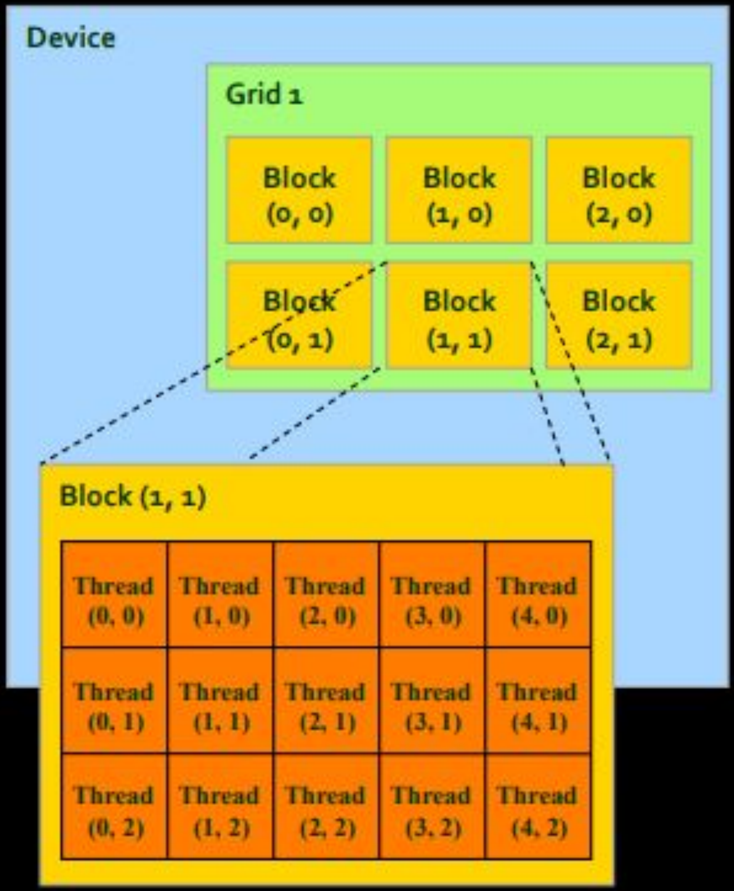
CUDA Memory Hierarchy

- Thread
  - Registers
  - Local memory
- Thread Block
  - Shared memory
- All Thread Blocks
  - Global Memory



Thread and Block ID and Dimensions

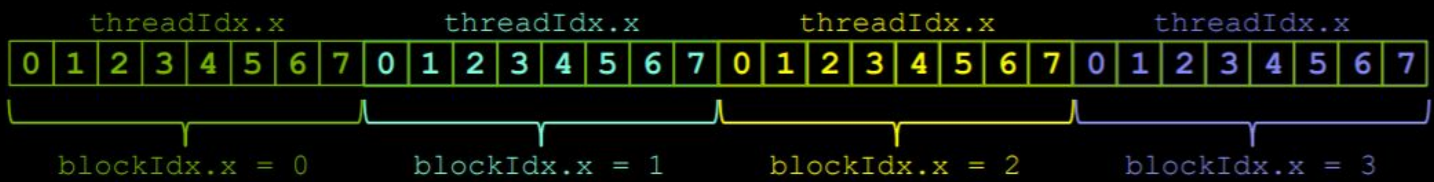
- Threads
  — 3D IDs, unique within a block
- Thread Blocks
  — 2D IDs, unique within a grid
- Dimensions set at launch
  — Can be unique for each grid
- Built-in variables
  — `threadIdx`, `blockIdx`
  — `blockDim`, `gridDim`
- Programmers usually select dimensions that simplify the mapping of the application data to CUDA threads



Indexing Arrays With Threads And Blocks

- No longer as simple as just using `threadIdx.x` or `blockIdx.x` as indices

- To index array with 1 thread per entry (using 8 threads/block)

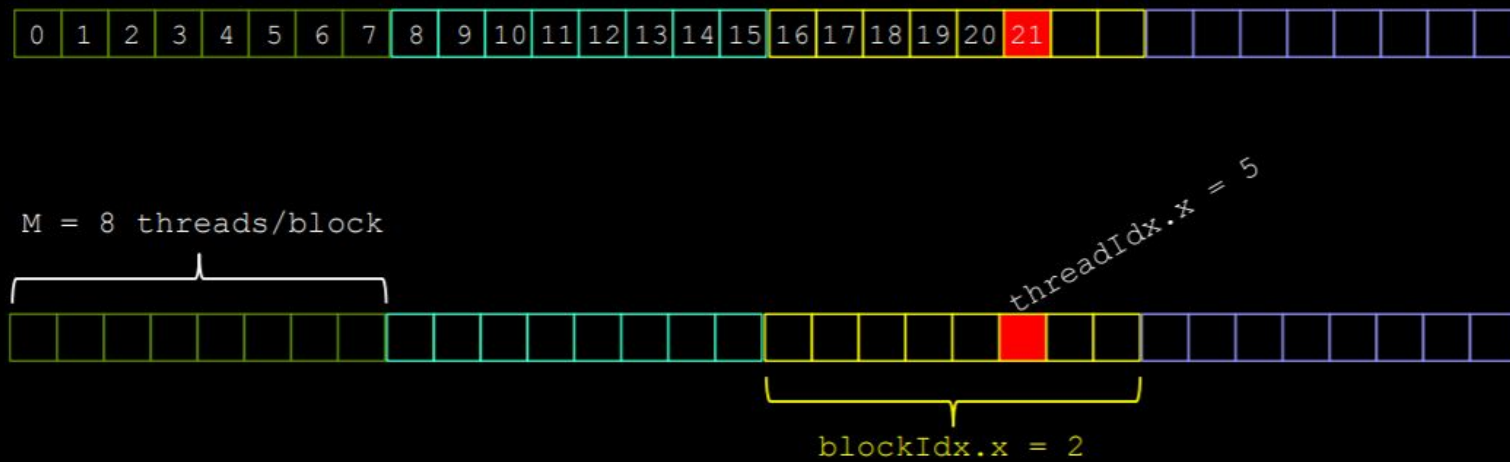


- If we have `M` threads/block, a unique array index for each entry given by

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

int index =         x    +      y     * width;
```

Indexing Arrays: Example

- In this example, the red entry would have an index of 21:




```
int index = threadIdx.x + blockIdx.x * M;
          =      5      +      2      * 8;
          = 21;
```

CUDA C

```
#include <stdio.h>

__global__ void print_kernel() {
    printf("Hello from block %d, thread %d\n", blockIdx.x, threadIdx.x);
}

int main() {
    print_kernel<<<10, 10>>>();
    cudaDeviceSynchronize();
}
```

```
Hello from block 1, thread 0
Hello from block 1, thread 1
Hello from block 1, thread 2
Hello from block 1, thread 3
Hello from block 1, thread 4
Hello from block 1, thread 5
....
Hello from block 8, thread 3
Hello from block 8, thread 4
Hello from block 8, thread 5
Hello from block 8, thread 6
Hello from block 8, thread 7
Hello from block 8, thread 8
Hello from block 8, thread 9
```

Parallel Programming in CUDA C

- With `add()` running in parallel…let's do vector addition
- Terminology: Each parallel invocation of `add()` referred to as a *block*
- Kernel can refer to its block's index with the variable `blockIdx.x`
- Each block adds a value from `a[]` and `b[]`, storing the result in `c[]`:

```
__global__ void add( int *a, int *b, int *c ) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- By using `blockIdx.x` to index arrays, each block handles different indices

Parallel Programming in CUDA C

- We write this code:

```
__global__ void add( int *a, int *b, int *c ) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- This is what runs in parallel on the device:

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];
```

Parallel Addition: main()

```
#define N  512
int main( void ) {
    int *a, *b, *c;                    // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c;        // device copies of a, b, c
    int size = N * sizeof( int );      // we need space for 512 integers

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, size );

    a = (int*)malloc( size );
    b = (int*)malloc( size );
    c = (int*)malloc( size );

    random_ints( a, N );
    random_ints( b, N );
```

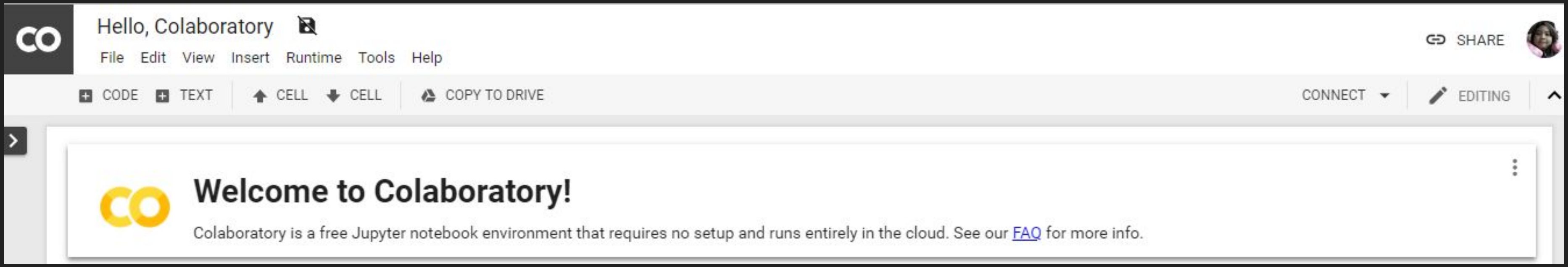
Parallel Addition: `main()` (cont)

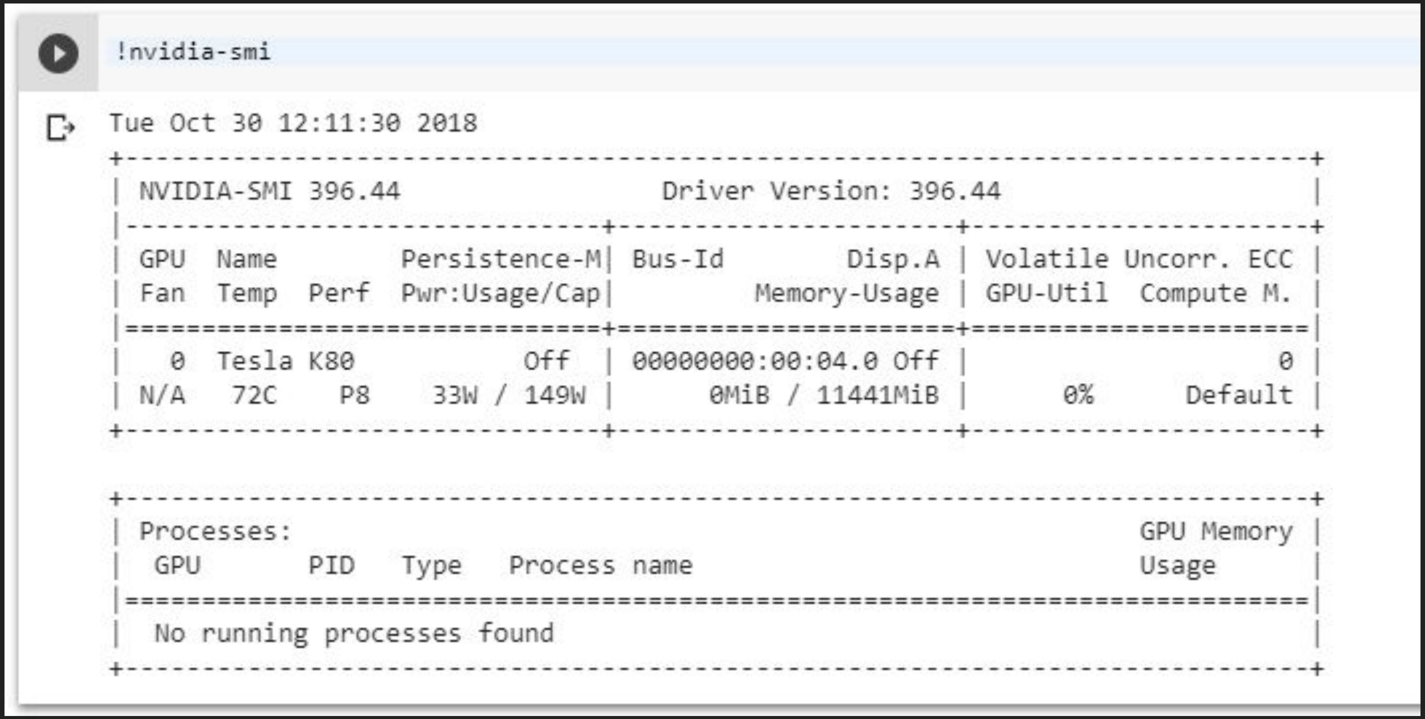
```
    // copy inputs to device
    cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );

    // launch add() kernel with N parallel blocks
    add<<< N, 1 >>>( dev_a, dev_b, dev_c );

    // copy device result back to host copy of c
    cudaMemcpy( c, dev_c, size, cudaMemcpyDeviceToHost );

    free( a ); free( b ); free( c );
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
    return 0;
}
```

CUDA using Python

- Anaconda/Python 3.6.1/Jupyter notebook
- CUDA Toolkit
- Numba package

CUDA using Python





```
!nvidia-smi
```

```
Tue Oct 30 12:11:30 2018
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 396.44                 Driver Version: 396.44                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla K80           Off  | 00000000:00:04.0 Off |                    0 |
| N/A   72C    P8    33W / 149W |      0MiB / 11441MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

CUDA using Python

```
!apt-get install nvidia-cuda-toolkit
```

```
  libvdpau-va-gl1 nvidia-vdpau-driver nvidia-legacy-340xx-vdpau-driver
  mesa-utils
Recommended packages:
  libnvcuvid1
The following NEW packages will be installed:
  adwaita-icon-theme at-spi2-core ca-certificates-java cpp-6
  dconf-gsettings-backend dconf-service fontconfig fonts-dejavu-core
```

```
!nvcc --version
```

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2018 NVIDIA Corporation
Built on Tue_Jun_12_23:07:04_CDT_2018
Cuda compilation tools, release 9.2, V9.2.148
```

```
!pip3 install numba
```

```
Collecting numba
  Downloading https://files.pythonhosted.org/packages/42/45/8d5fc45e5f760ac65906ba48dec98e99e7920c96783ac7248c5e31c9464e/numba-0.40.1-cp36-cp36m-manylinux1_x8
    100% |████████████████████████████████| 3.2MB 8.3MB/s
Requirement already satisfied: numpy in /usr/local/lib/python3.6/dist-packages (from numba) (1.14.6)
Collecting llvmlite>=0.25.0dev0 (from numba)
  Downloading https://files.pythonhosted.org/packages/34/fb/f9c2e9e0ef2b54c52f0b727cf6af75b68c3d7ddb6d88c8d557b1b16bc1ab/llvmlite-0.25.0-cp36-cp36m-manylinux1
    100% |████████████████████████████████| 16.1MB 2.8MB/s
Installing collected packages: llvmlite, numba
Successfully installed llvmlite-0.25.0 numba-0.40.1
```

Vector add GPU

```python
from __future__ import print_function
from timeit import default_timer as time
import numpy as np
from numba import cuda

@cuda.jit('(f4[:], f4[:], f4[:])')
def cuda_sum(a, b, c):
    i = cuda.grid(1)
    c[i] = a[i] + b[i]

griddim = 50, 1
blockdim = 32, 1, 1
N = griddim[0] * blockdim[0]
print("N", N)
cuda_sum_configured = cuda_sum.configure(griddim, blockdim)
a = np.array(np.random.random(N), dtype=np.float32)
b = np.array(np.random.random(N), dtype=np.float32)
c = np.empty_like(a)

ts = time()
cuda_sum_configured(a, b, c)
te = time()
print(te - ts)


assert (a + b == c).all()
print(c)
```

http://numba.pydata.org/numba-doc/0.13/CUDAJit.html

Vector add CPU

```
from timeit import default_timer as time
import numpy as np
N = 1600
def cpu_sum(a, b, c):
    for i in range(0, N):
      c[i] = a[i] + b[i]


a = np.array(np.random.random(N), dtype=np.float32)
b = np.array(np.random.random(N), dtype=np.float32)
c = np.empty_like(a)

ts = time()
cpu_sum(a, b, c)
te = time()
print(te - ts)

print(c)
```