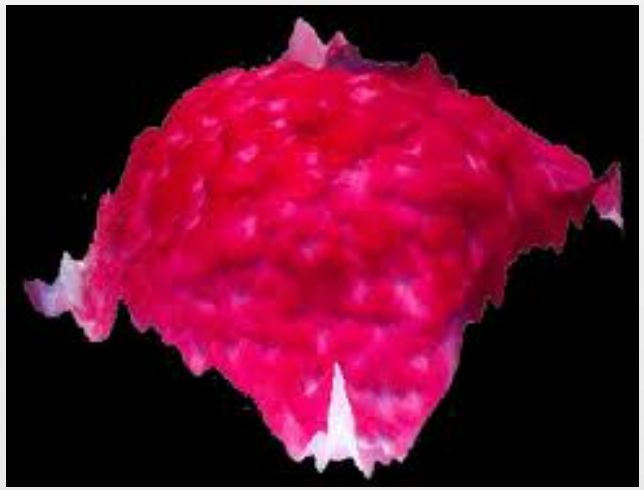
GPU-accelerated Programming with CUDA
An Introduction

Carlos J. Barrios Hernández., PhD
@carlosjaimebh
@SC3UIS

Super Computación y
Cálculo Científico UIS

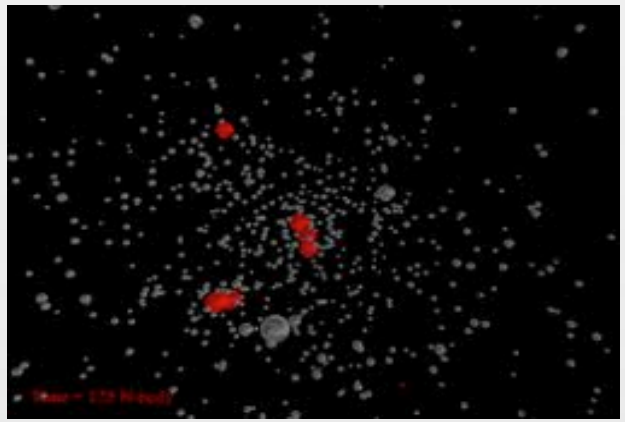
3 in House Examples

Microscopy Using EDF



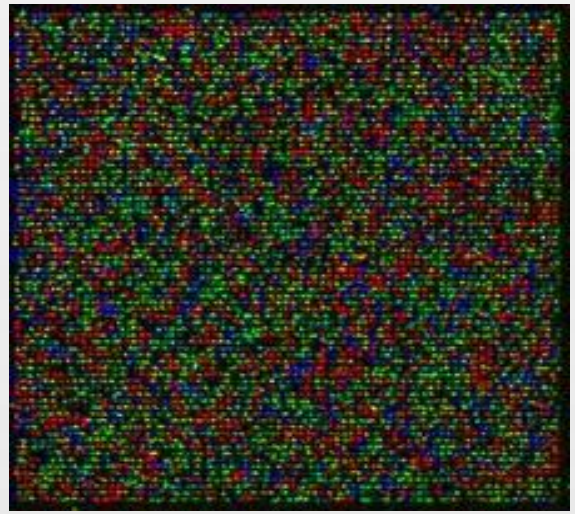
5x in 5 Hours

N-Bodies Based in Montecarlo



12x in 1 Hour

Iinfluenza MetaGenomics



125x in 2 Hours

CUDA C

```
void saxpy(int n, float a,
           float *x, float *y)
{
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}

int N = 1<<20;



// Perform SAXPY on 1M elements
saxpy(N, 2.0, x, y);
```

```
__global__
void saxpy(int n, float a,
           float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

int N = 1<<20;
cudaMemcpy(d_x, x, N, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N, cudaMemcpyHostToDevice);

// Perform SAXPY on 1M elements
saxpy<<<4096,256>>>(N, 2.0, d_x, d_y);

cudaMemcpy(y, d_y, N, cudaMemcpyDeviceToHost);
```

http://developer.nvidia.com/cuda-toolkit

Thrust C++ Template Library

Serial C++ Code

with STL and Boost

```
int N = 1<<20;
std::vector<float> x(N), y(N);

...



// Perform SAXPY on 1M elements
std::transform(x.begin(), x.end(),
               y.begin(), y.end(),
               2.0f * _1 + _2);
```

www.boost.org/libs/lambda

Parallel C++ Code

```
int N = 1<<20;
thrust::host_vector<float> x(N), y(N);

...

thrust::device_vector<float> d_x = x;
thrust::device_vector<float> d_y = y;


// Perform SAXPY on 1M elements
thrust::transform(d_x.begin(), d_x.end(),
                  d_y.begin(), d_y.begin(),
                  2.0f * _1 + _2);
```

http://thrust.github.com

CUDA Fortran

Standard Fortran

```
module mymodule contains
  subroutine saxpy(n, a, x, y)
    real :: x(:), y(:), a
    integer :: n, i
    do i=1,n
      y(i) = a*x(i)+y(i)
    enddo
  end subroutine saxpy
end module mymodule

program main
  use mymodule
  real :: x(2**20), y(2**20)
  x = 1.0, y = 2.0

  ! Perform SAXPY on 1M elements
  call saxpy(2**20, 2.0, x, y)

end program main
```

Parallel Fortran

```
module mymodule contains
  attributes(global) subroutine saxpy(n, a, x, y)
    real :: x(:), y(:), a
    integer :: n, i
    attributes(value) :: a, n
    i = threadIdx%x+(blockIdx%x-1)*blockDim%x
    if (i<=n) y(i) = a*x(i)+y(i)
  end subroutine saxpy
end module mymodule

program main
  use cudafor; use mymodule
  real, device :: x_d(2**20), y_d(2**20)
  x_d = 1.0, y_d = 2.0

  ! Perform SAXPY on 1M elements
  call saxpy<<<4096,256>>>(2**20, 2.0, x_d, y_d)

end program main
```

http://developer.nvidia.com/cuda-fortran

Python

Standard Python

```
import numpy as np


def saxpy(a, x, y):
  return [a * xi + yi
          for xi, yi in zip(x, y)]

x = np.arange(2**20, dtype=np.float32)
y = np.arange(2**20, dtype=np.float32)


cpu_result = saxpy(2.0, x, y)
```

http://numpy.scipy.org

Copperhead: Parallel Python

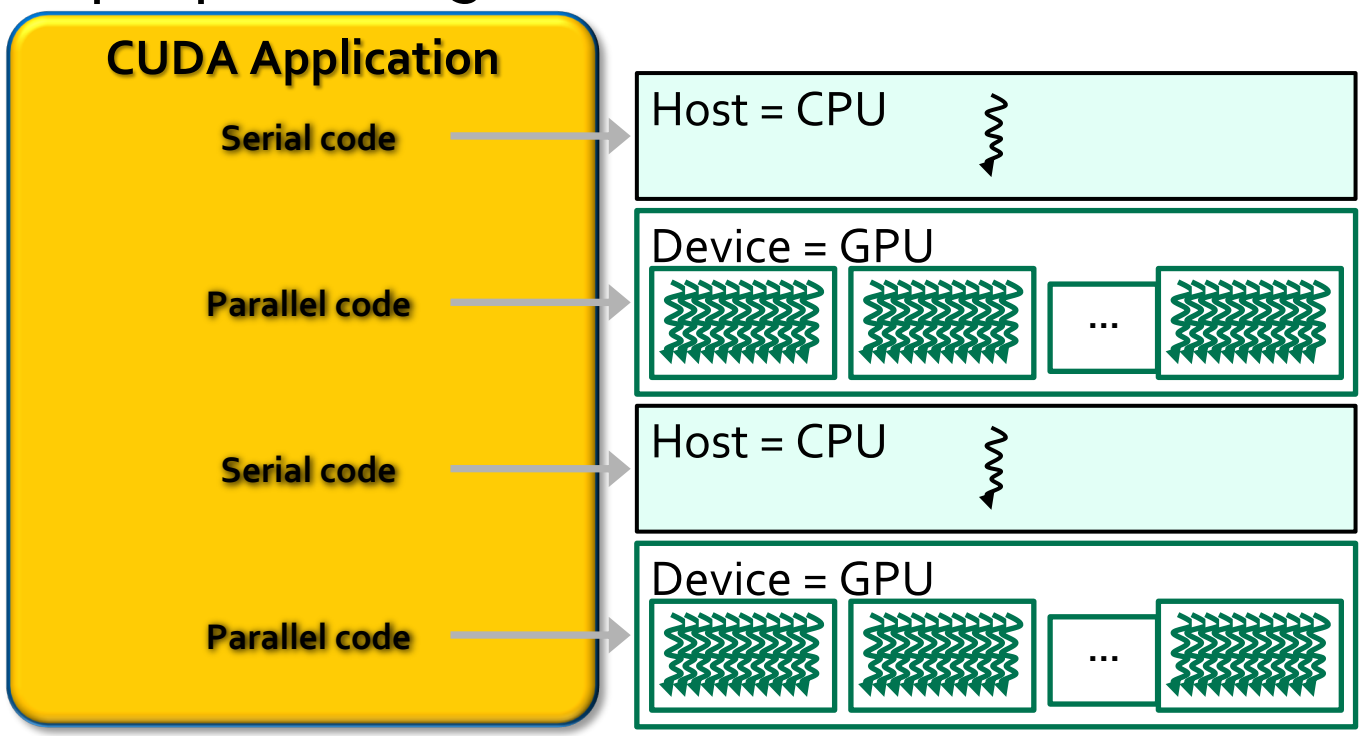
```
from copperhead import *
import numpy as np

@cu
def saxpy(a, x, y):
  return [a * xi + yi
          for xi, yi in zip(x, y)]

x = np.arange(2**20, dtype=np.float32)
y = np.arange(2**20, dtype=np.float32)

with places.gpu0:
  gpu_result = saxpy(2.0, x, y)

with places.openmp:
  cpu_result = saxpy(2.0, x, y)
```

http://copperhead.github.com

Anatomy of a CUDA Application

Serial code executes in a Host (CPU) thread

Parallel code executes in many Device (GPU) threads across multiple processing elements



CUDA Kernels

Parallel portion of application: execute as a kernel

- Entire GPU executes kernel, many threads

CUDA threads:

- Lightweight
- Fast switching
- 1000s execute simultaneously

CPU	Host	Executes functions
GPU	Device	Executes kernels

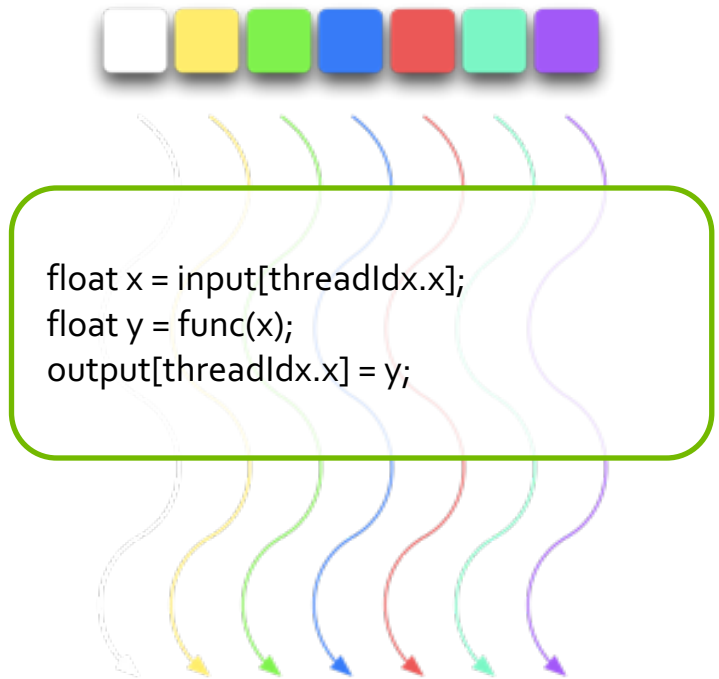
CUDA Kernels: Parallel Threads

A kernel is a function executed on the GPU as an array of threads in parallel

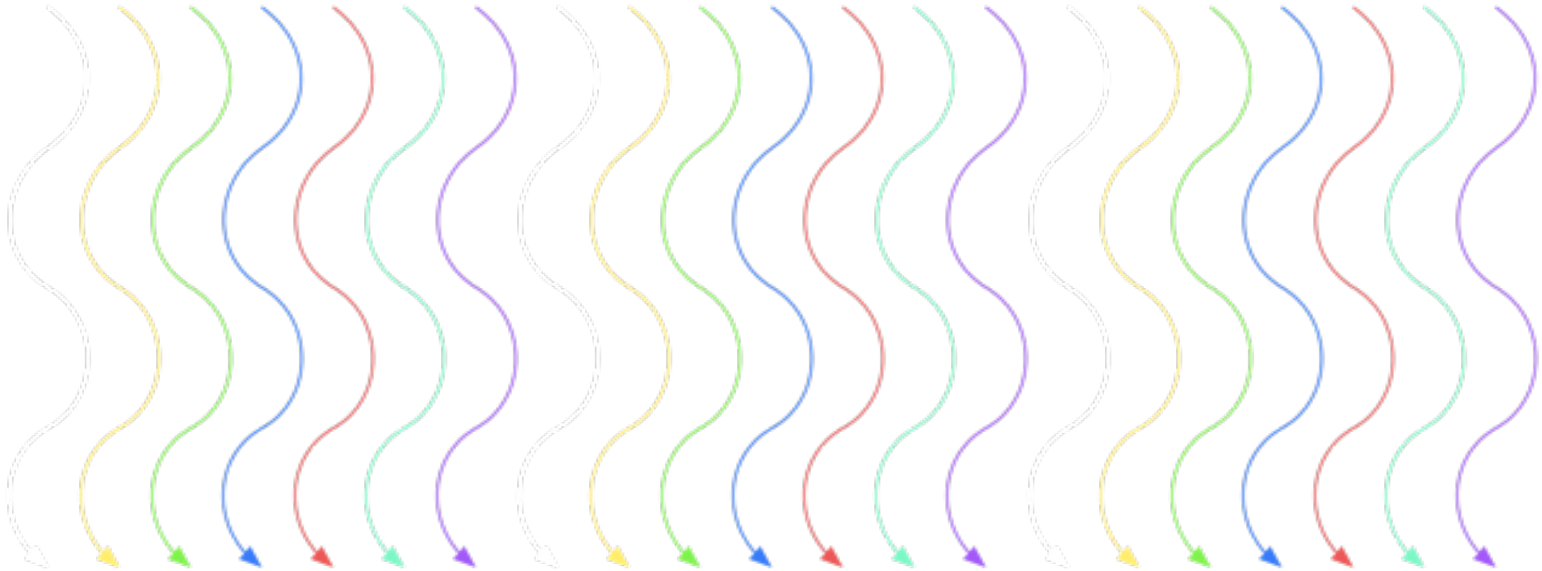
All threads execute the same code, can take different paths

Each thread has an ID

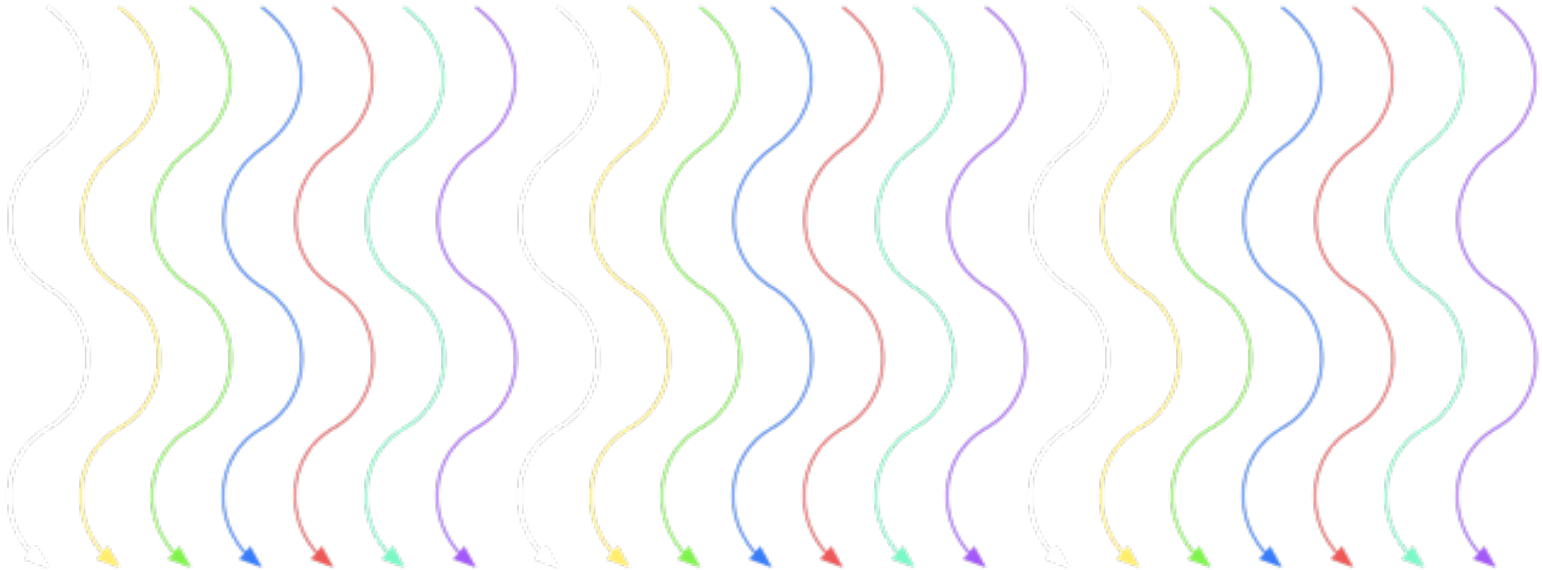
- Select input/output data
- Control decisions



CUDA Kernels: Subdivide into Blocks

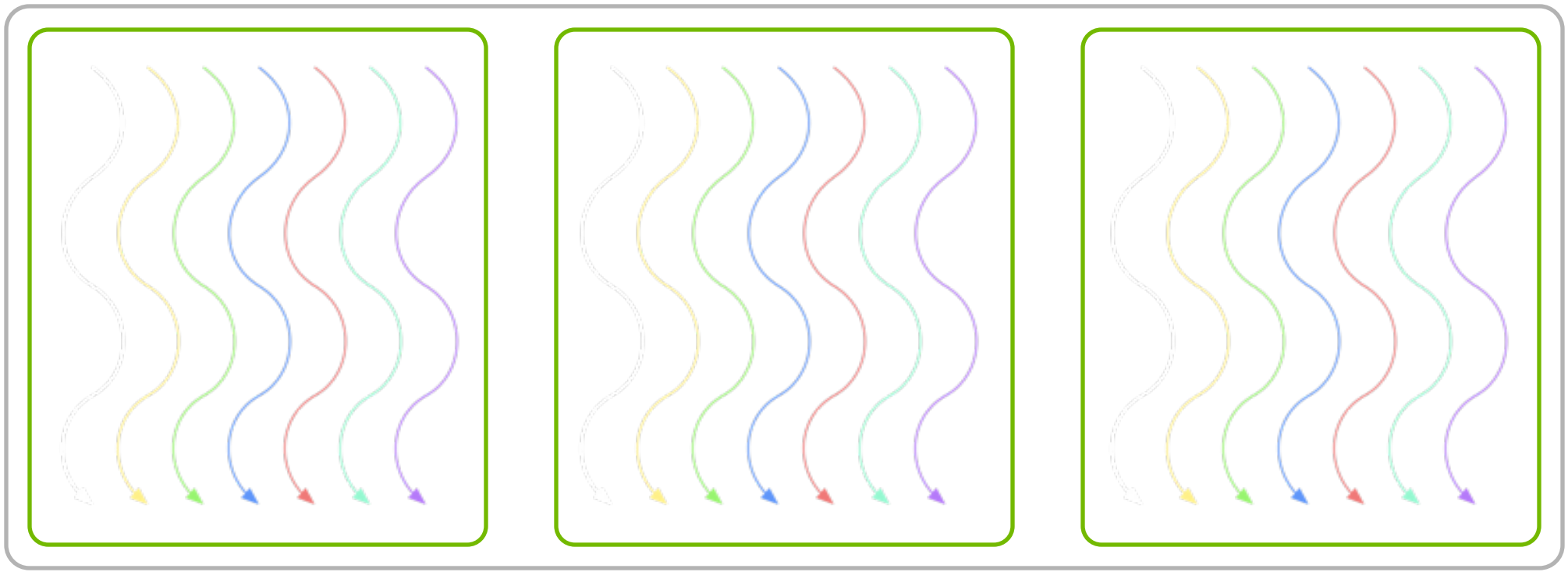


CUDA Kernels: Subdivide into Blocks



Threads are grouped into blocks

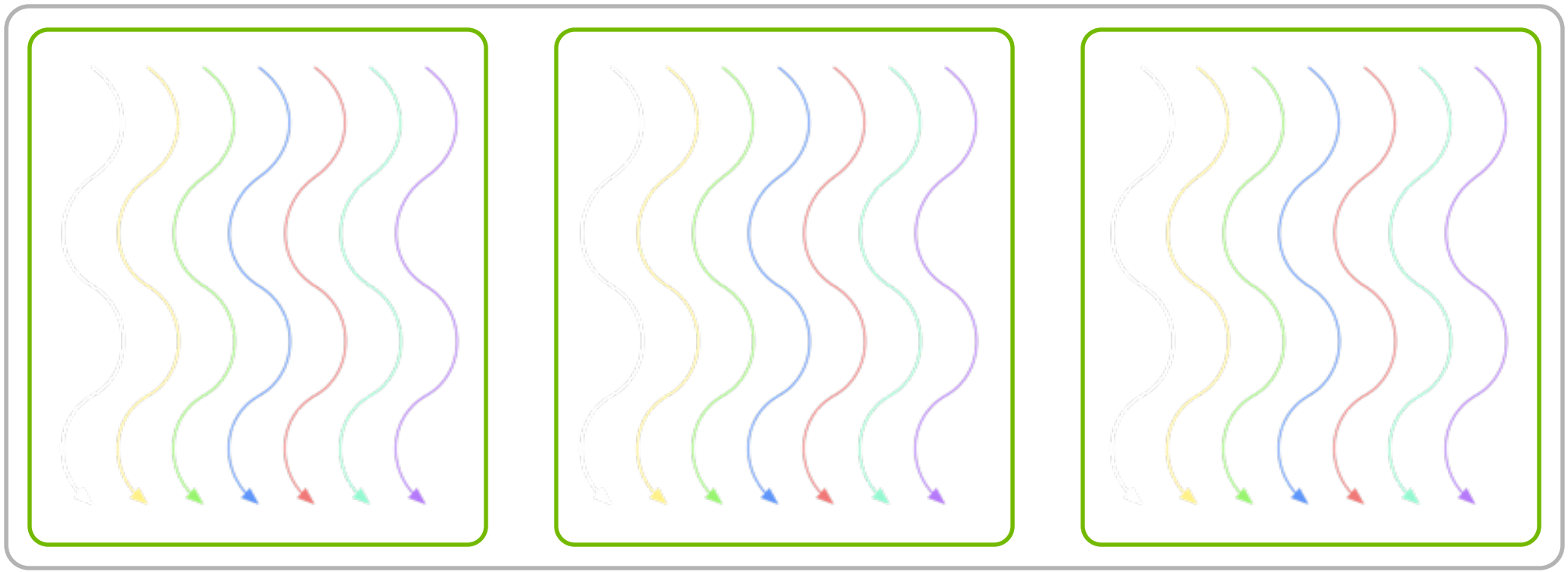
CUDA Kernels: Subdivide into Blocks



Threads are grouped into blocks

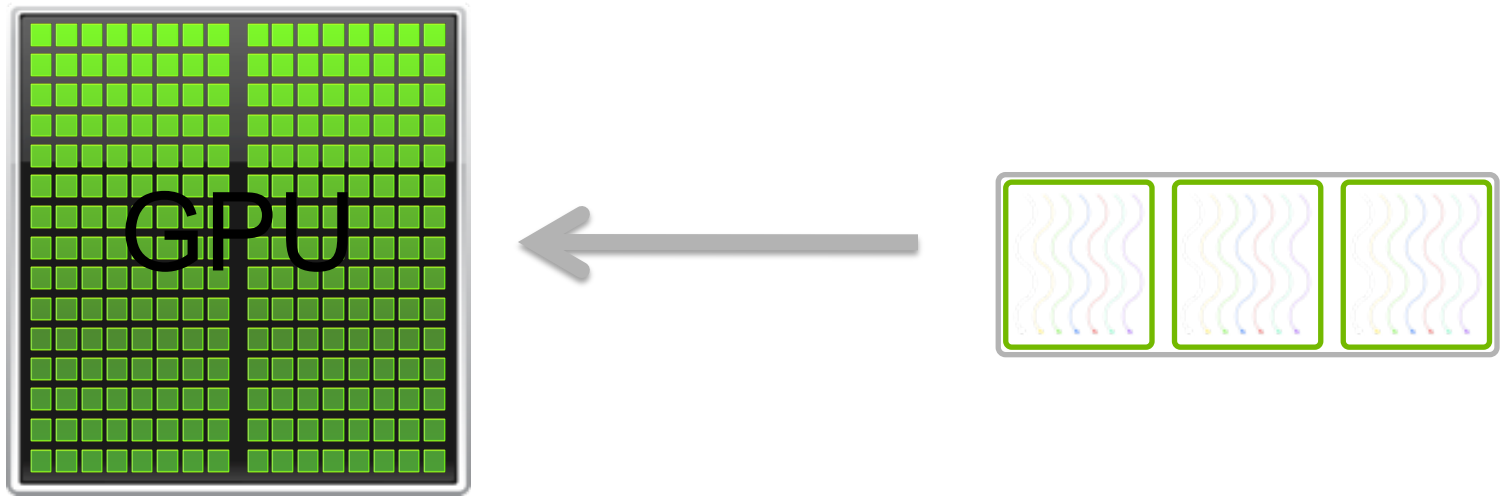
Blocks are grouped into a grid

CUDA Kernels: Subdivide into Blocks



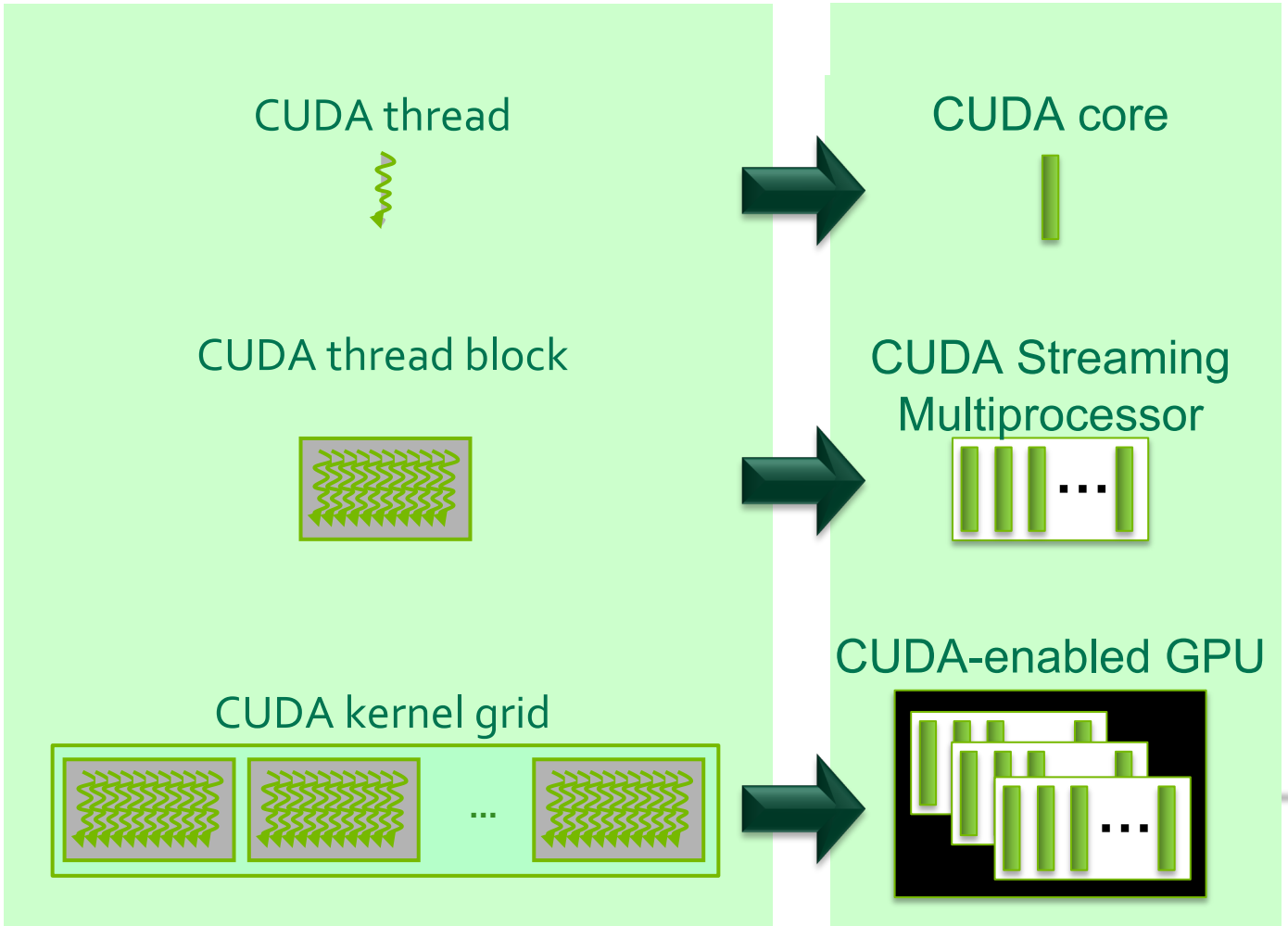
Threads are grouped into blocks
Blocks are grouped into a grid
A kernel is executed as a grid of blocks of threads

CUDA Kernels: Subdivide into Blocks



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

Kernel Execution



- Each thread is executed by a core
- Each block is executed by one SM and does not migrate
- Several concurrent blocks can reside on one SM depending on the blocks' memory requirements and the SM's memory resources
- Each kernel is executed on one device
- Multiple kernels can execute on a device at one time

Thread blocks allow cooperation

Threads may need to cooperate:

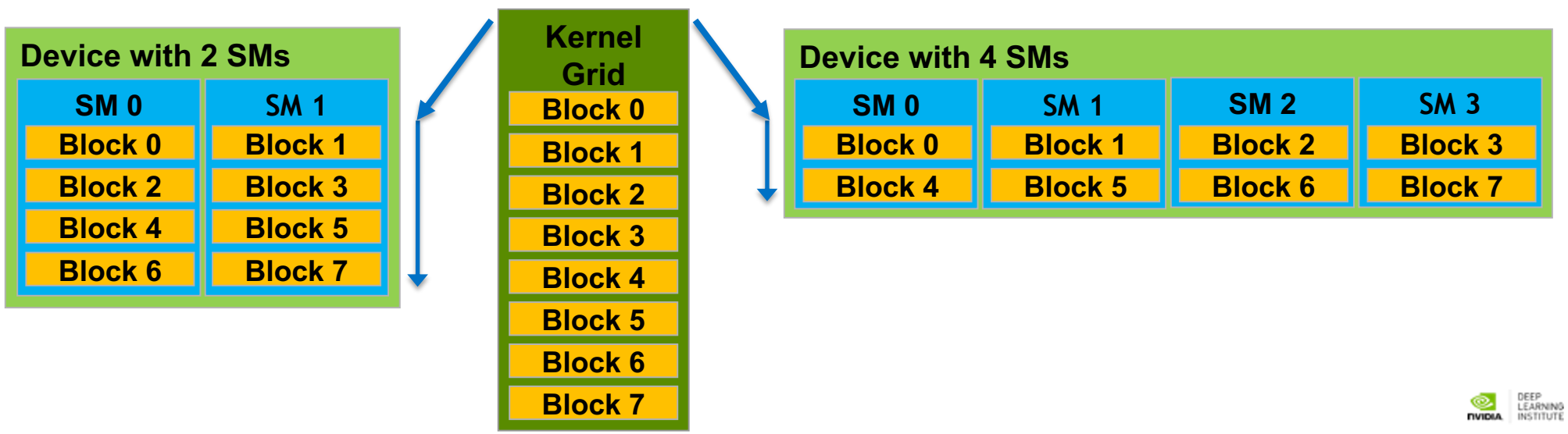
- Cooperatively load/store blocks of memory that they all use
- Share results with each other or cooperate to produce a single result
- Synchronize with each other

Thread blocks allow scalability

Blocks can execute in any order, concurrently or sequentially

This independence between blocks gives scalability:

A kernel scales across any number of SMs



Memory hierarchy

Thread:

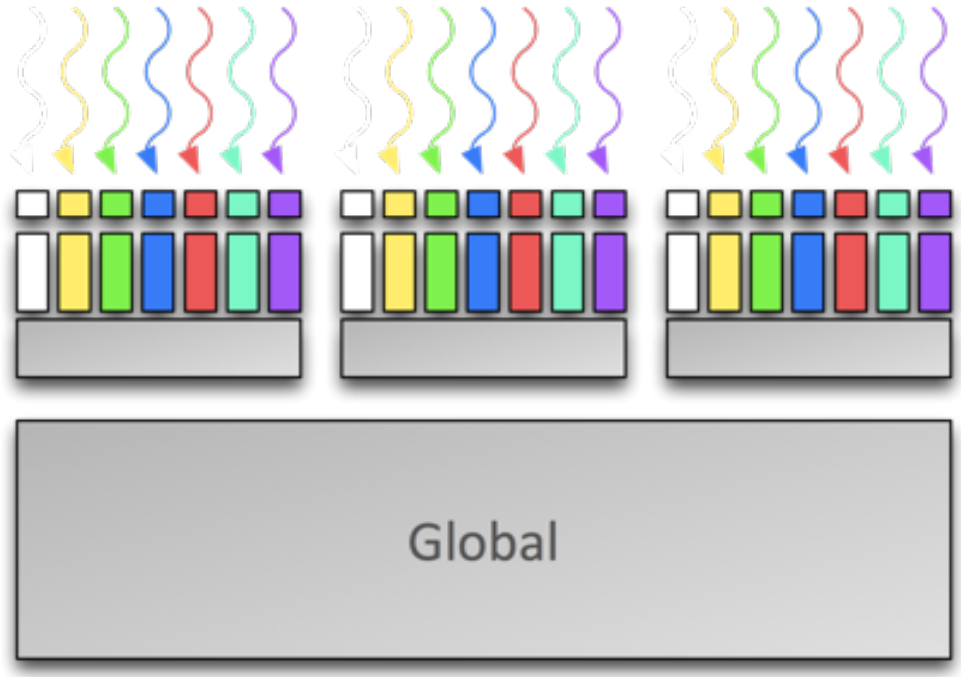
- Registers
- Local memory

Block of threads:

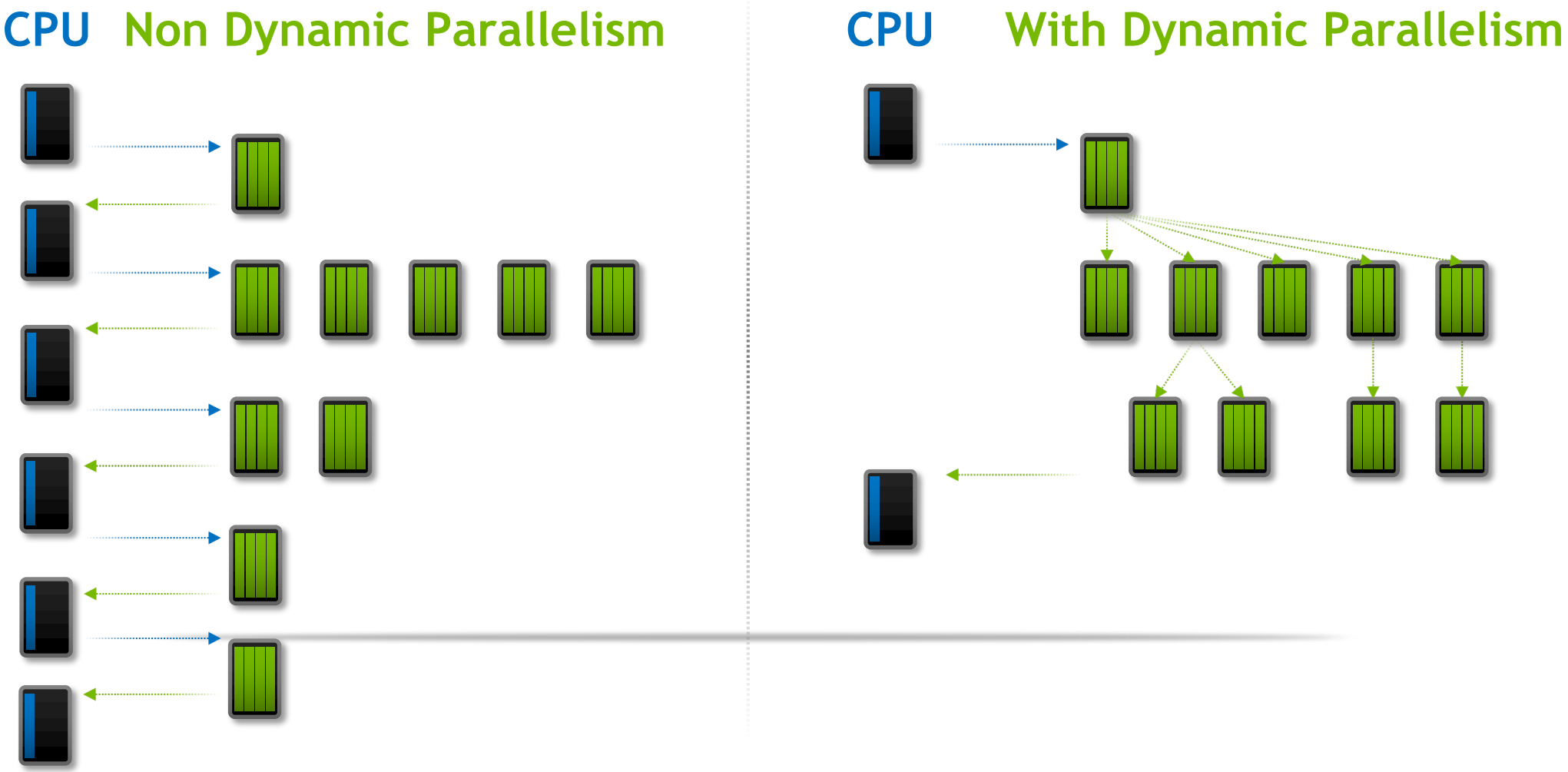
- Shared memory

All blocks:

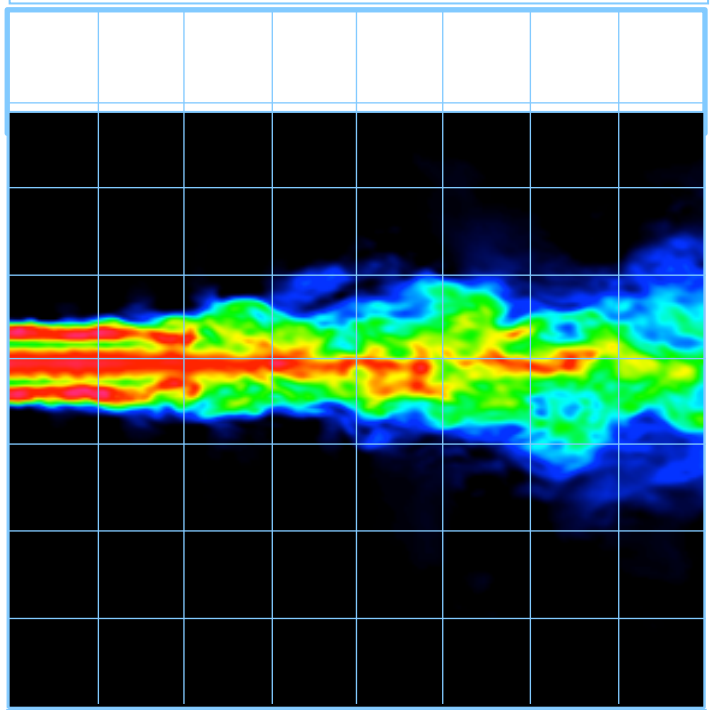
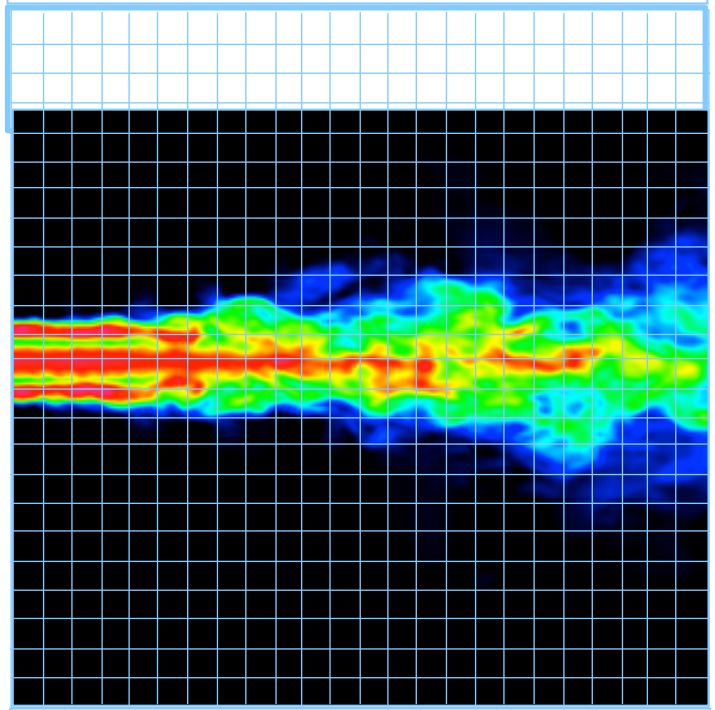
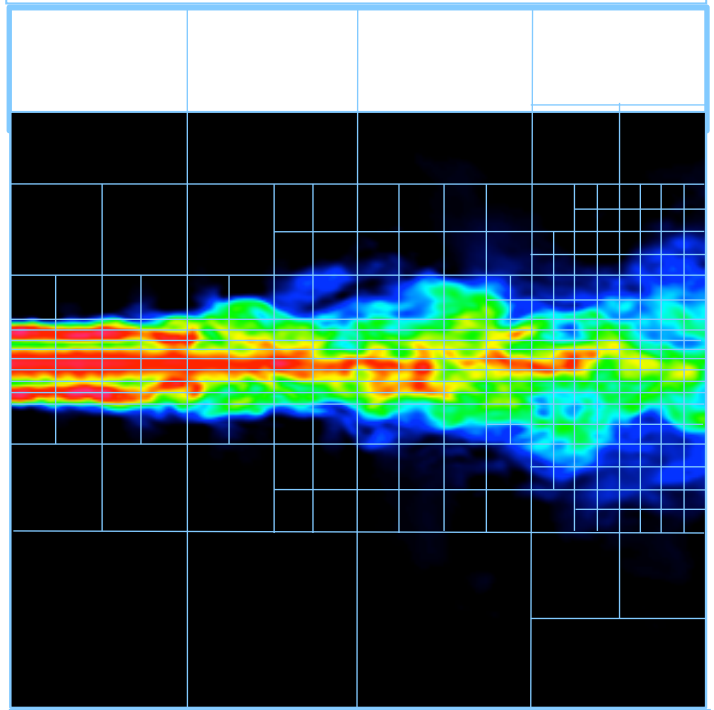
- Global memory



Dynamic Parallelism



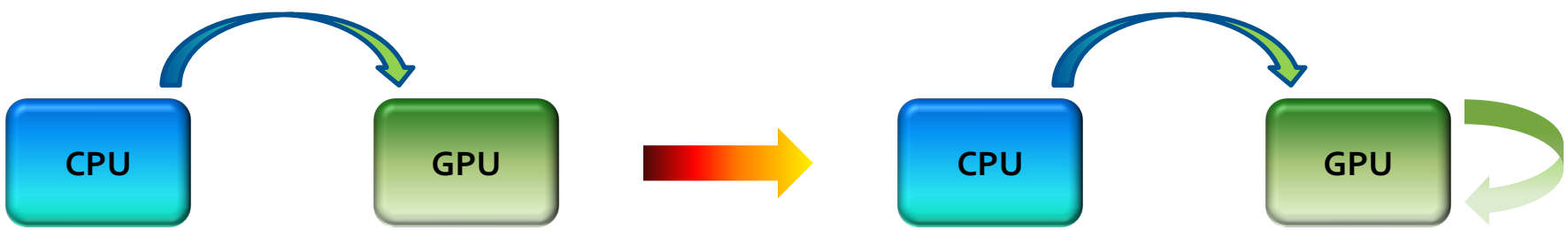
Dynamic Work Generation

Coarse grid	Fine grid	*Dynamic grid*
		
Higher Performance Lower Accuracy	Lower Performance Higher Accuracy	*Target performance where accuracy is required*

What is Dynamic Parallelism?

The ability to launch new kernels from the GPU

- Dynamically - based on run-time data
- Simultaneously - from multiple threads at once
- Independently - each thread can launch a different grid



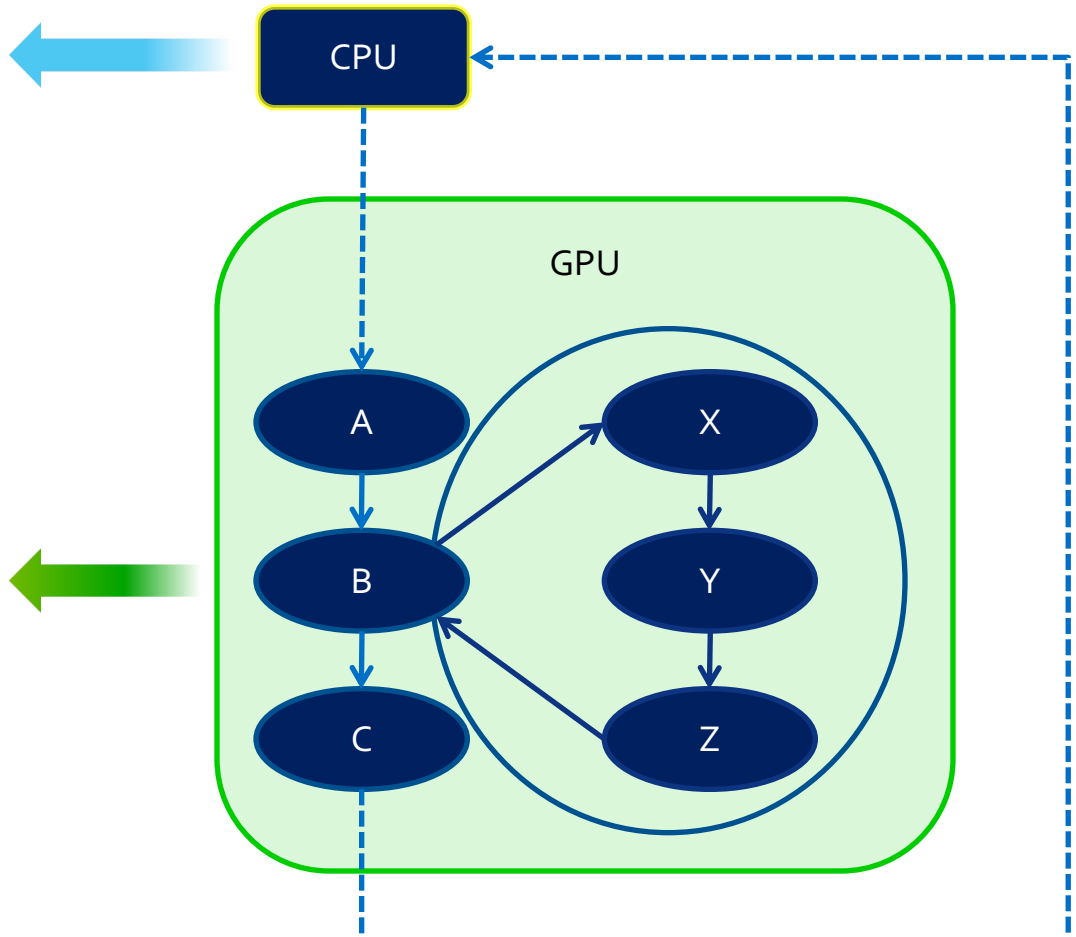
Fermi: Only CPU can generate GPU work

Kepler: GPU can generate work for itself

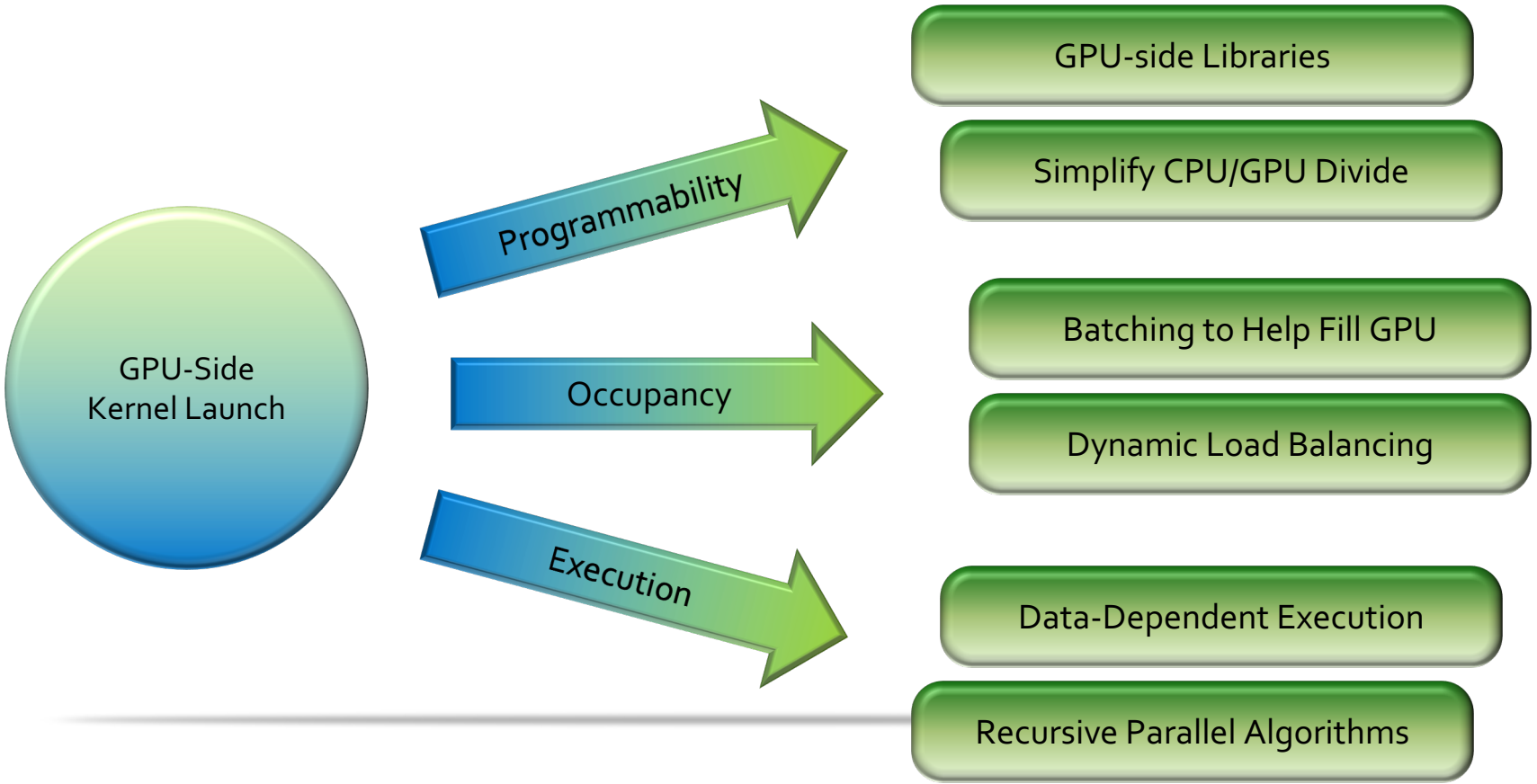
Familiar Programming Model

```
int main() {
    float *data;
    setup(data);

    A <<< ... >>> (data);
    B <<< ... >>> (data);
    C <<< ... >>> (data);

    cudaDeviceSynchronize();
    return 0;
}
```

```
__global__ void B(float *data)
{
    do_stuff(data);

    X <<< ... >>> (data);
    Y <<< ... >>> (data);
    Z <<< ... >>> (data);
    cudaDeviceSynchronize();

    do_more_stuff(data);
}
```



Challenges for the State of Art Program Model



Compiling a CUDA™ code

Using nvcc™ compilator

Visit this site and run the examples (after this sesion):

https://docs.nvidia.com/cuda/cuda-samples/index.html

Typical compiling

nvcc mycudacode.cu

Specific compilation

nvcc -(args) mycudacude.cu - (extensions)

NVCC Compiler

- NVIDIA provides a CUDA-C compiler
  - nvcc
- NVCC compiles device code then forwards code on to the host compiler (e.g. g++)
- Can be used to compile & link host only applications

Example 1: Hello World

```
int main() {
  printf("Hello World!\n");
  return 0;
}
```

Instructions:

1. Build and run the hello world code
2. Modify Makefile to use nvcc instead of g++
3. Rebuild and run

CUDA Example 1: Hello World

```
__global__ void mykernel(void) {
}

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

Instructions:

1. Add kernel and kernel launch to main.cu
2. Try to build

CUDA Example 1: Build Considerations

- Build failed
  - Nvcc only parses .cu files for CUDA
- Fixes:
  - Rename main.cc to main.cu

  OR
  - nvcc -x cu
    - Treat all input files as .cu files

Instructions:

1. Rename main.cc to main.cu
2. Rebuild and Run

Hello World! with Device Code

```
__global__ void mykernel(void) {
}

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

Output:

```
$ nvcc main.cu
$ ./a.out
Hello World!
```

- `mykernel()` does nothing, somewhat anticlimactic!)

The Real and Complet « Hello World » in CUDA

```
// This is the REAL "hello world" for CUDA!
// It takes the string "Hello ", prints it, then passes it to CUDA with an array
// of offsets. Then the offsets are added in parallel to produce the string "World!"
// By Ingemar Ragnemalm 2010

#include <stdio.h>

const int N = 7;
const int blocksize = 7;

__global__
void hello(char *a, int *b)
{
  a[threadIdx.x] += b[threadIdx.x];
}

int main()
{
  char a[N] = "Hello ";
  int b[N] = {15, 10, 6, 0, -11, 1, 0};

  char *ad;
  int *bd;
  const int csize = N*sizeof(char);
  const int isize = N*sizeof(int);

  printf("%s", a);

  cudaMalloc( (void**)&ad, csize );
  cudaMalloc( (void**)&bd, isize );
  cudaMemcpy( ad, a, csize, cudaMemcpyHostToDevice );
  cudaMemcpy( bd, b, isize, cudaMemcpyHostToDevice );

  dim3 dimBlock( blocksize, 1 );
  dim3 dimGrid( 1, 1 );
  hello<<<dimGrid, dimBlock>>>(ad, bd);
  cudaMemcpy( a, ad, csize, cudaMemcpyDeviceToHost );
  cudaFree( ad );

  printf("%s\n", a);
  return EXIT_SUCCESS;
}
```

Compiler Flags

- Remember there are two compilers being used
  - NVCC: Device code
  - Host Compiler: C/C++ code
- NVCC supports some host compiler flags
  - If flag is unsupported, use -Xcompiler to forward to host
    - e.g. -Xcompiler -fopenmp
- Debugging Flags
  - -g: Include host debugging symbols
  - -G: Include device debugging symbols
  - -lineinfo: Include line information with symbols

CUDA-MEMCHECK

- Memory debugging tool
  - No recompilation necessary

    %> cuda-memcheck ./exe
- Can detect the following errors
  - Memory leaks
  - Memory errors (OOB, misaligned access, illegal instruction, etc)
  - Race conditions
  - Illegal Barriers
  - Uninitialized Memory
- For line numbers use the following compiler flags:
  - -Xcompiler -rdynamic -lineinfo

http://docs.nvidia.com/cuda/cuda-memcheck

Example 2: CUDA-MEMCHECK

Instructions:

1. Build & Run "Hello World"

Output should be the numbers 0-9

Do you get the correct results?

2. Run with cuda-memcheck

```
%> cuda-memcheck ./a.out
```

3. Rebuild & Run with cuda-memcheck
4. Fix the illegal write

http://docs.nvidia.com/cuda/cuda-memcheck

NVIDIA-SMI

- nvidia-smi : The NVIDIA System Management Interface (nvidia-smi) is a command line utility, based on top of the NVIDIA Management Library (NVML), intended to aid in the management and monitoring of NVIDIA GPU devices.

- Explore the site: http://nvidia.custhelp.com/app/answers/detail/a_id/3751/~/useful-nvidia-smi-queries and follow the instructions for the commands and see the information in the selected node of the practice.

NVIDIA-SMI

- nvidia-smi : The NVIDIA System Management Interface (nvidia-smi) is a command line utility, based on top of the NVIDIA Management Library (NVML), intended to aid in the management and monitoring of NVIDIA GPU devices.

- Explore the site: http://nvidia.custhelp.com/app/answers/detail/a_id/3751/~/useful-nvidia-smi-queries and follow the instructions for the commands and see the information in the selected node of the practice.

Practice Time

- nvidia-smi : The NVIDIA System Management Interface (nvidia-smi) is a command line utility, based on top of the NVIDIA Management Library (NVML), intended to aid in the management and monitoring of NVIDIA GPU devices.

- Explore the site: http://nvidia.custhelp.com/app/answers/detail/a_id/3751/~/useful-nvidia-smi-queries and follow the instructions for the commands and see the information in the selected node of the practice.

- Explore, compile and run the 5 simple examples in the folder SCCAMP.

Thank you!
@carlosjaimebh

www.nvidia.com/dli