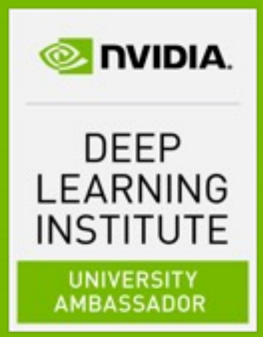
# Introducing GPUs Architecture - Important Aspects -

Carlos Jaime BARRIOS HERNANDEZ, PhD.



# About Top500 List -2022

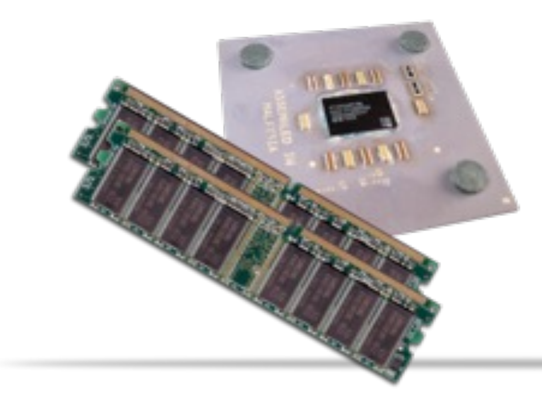
www.top500.org

| Rank | System | Cores | Rmax (PFlop/s) | Rpeak (PFlop/s) | Power (kW) |
|---|---|---|---|---|---|
| 1 | **Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE<br>DOE/SC/Oak Ridge National Laboratory<br>United States | 8,730,112 | 1,102.00 | 1,685.65 | 21,100 |
| 2 | **Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu<br>RIKEN Center for Computational Science<br>Japan | 7,630,848 | 442.01 | 537.21 | 29,899 |
| 3 | **LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE<br>EuroHPC/CSC<br>Finland | 2,220,288 | 309.10 | 428.70 | 6,016 |
| 4 | **Leonardo** - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband, Atos<br>EuroHPC/CINECA<br>Italy | 1,463,616 | 174.70 | 255.75 | 5,610 |
| 5 | **Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM<br>DOE/SC/Oak Ridge National Laboratory<br>United States | 2,414,592 | 148.60 | 200.79 | 10,096 |
| 6 | **Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox<br>DOE/NNSA/LLNL<br>United States | 1,572,480 | 94.64 | 125.71 | 7,438 |
| 7 | **Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC<br>National Supercomputing Center in Wuxi<br>China | 10,649,600 | 93.01 | 125.44 | 15,371 |
| 8 | **Perlmutter** - HPE Cray EX235n, AMD EPYC 7763 64C 2.45GHz, NVIDIA A100 SXM4 40 GB, Slingshot-10, HPE<br>DOE/SC/LBNL/NERSC<br>United States | 761,856 | 70.87 | 93.75 | 2,589 |
| 9 | **Selene** - NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband, Nvidia<br>NVIDIA Corporation<br>United States | 555,520 | 63.46 | 79.22 | 2,646 |
| 10 | **Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000, NUDT<br>National Super Computer Center in Guangzhou<br>China | 4,981,760 | 61.44 | 100.68 | 18,482 |

| Rank | System | Cores | Rmax (PFlop/s) | Rpeak (PFlop/s) | Power (kW) |
|---|---|---|---|---|---|
| 33 | **Pégaso** - Supermicro A+ Server 4124GO-NART+, AMD EPYC 7513 32C 2.6GHz, NVIDIA A100, Infiniband HDR, Atos<br>Petróleo Brasileiro S.A<br>Brazil | 233,856 | 19.07 | 42.00 | 1,033 |
| 65 | **Dragão** - Supermicro SYS-4029GP-TVRT, Xeon Gold 6230R 26C 2.1GHz, NVIDIA Tesla V100, Infiniband EDR, Atos<br>Petróleo Brasileiro S.A<br>Brazil | 188,224 | 8.98 | 14.01 | 943 |
| 126 | **Atlas** - Bull 4029GP-TVRT, Xeon Gold 6240 18C 2.6GHz, NVIDIA Tesla V100, Infiniband EDR, Atos<br>Petróleo Brasileiro S.A<br>Brazil | 91,936 | 4.38 | 8.85 | 547 |
| 147 | **IARA** - NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100 SXM4 40 GB, Infiniband, Nvidia<br>SiDi<br>Brazil | 24,800 | 3.66 | 4.13 | |
| 155 | **NOBZ1** - ThinkSystem C2397, Xeon Platinum 8280 28C 2.7GHz, Broadcom, Lenovo<br>Software Company MBZ<br>Brazil | 80,640 | 3.55 | 6.97 | |
| 179 | **Fênix** - Bull 4029GP-TVRT, Xeon Gold 5122 4C 3.6GHz, NVIDIA Tesla V100, Infiniband EDR, Atos<br>Petróleo Brasileiro S.A<br>Brazil | 60,480 | 3.16 | 5.37 | 390 |
| 387 | **A16A** - ThinkSystem C0366, Xeon Gold 6252 24C 2.1GHz, 100G Ethernet, Lenovo<br>Software Company MBZ<br>Brazil | 61,440 | 2.09 | 4.13 | |
| 462 | **Santos Dumont (SDumont)** - Bull Sequana X1000, Xeon Gold 6252 24C 2.1GHz, Mellanox InfiniBand EDR, NVIDIA Tesla V100 SXM2, Atos<br>Laboratório Nacional de Computação Científica<br>Brazil | 33,856 | 1.85 | 2.73 | |

# Architecture Terminology

- Heterogeneous Computing
  - *Host* The CPU and its memory (host memory)
  - *Device* The GPU and its memory (device memory)

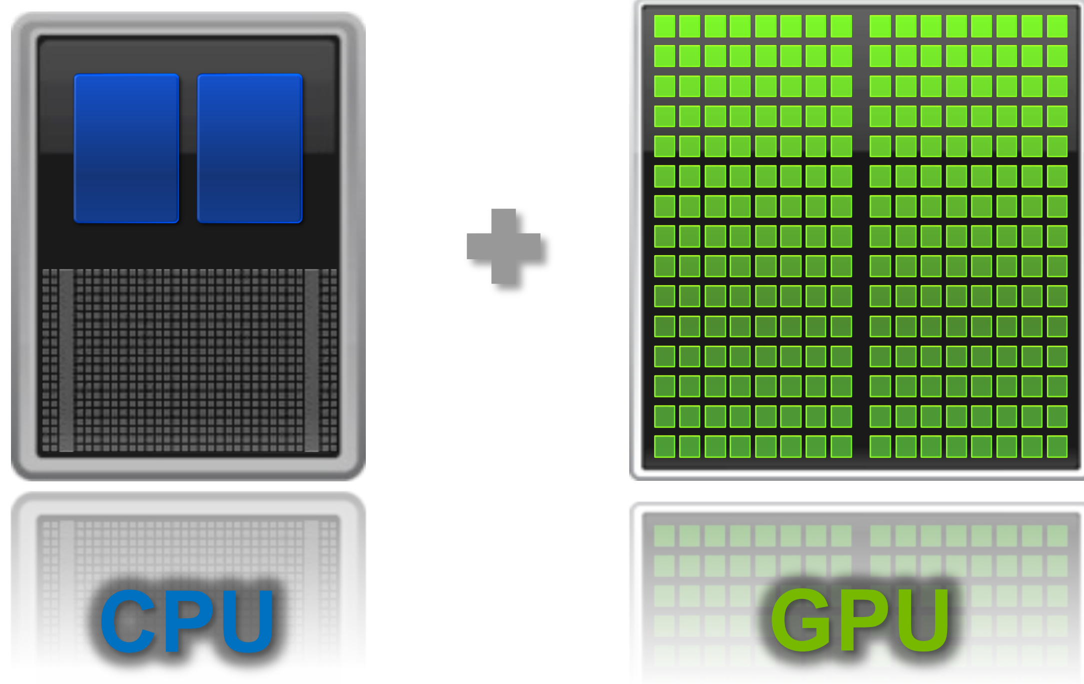


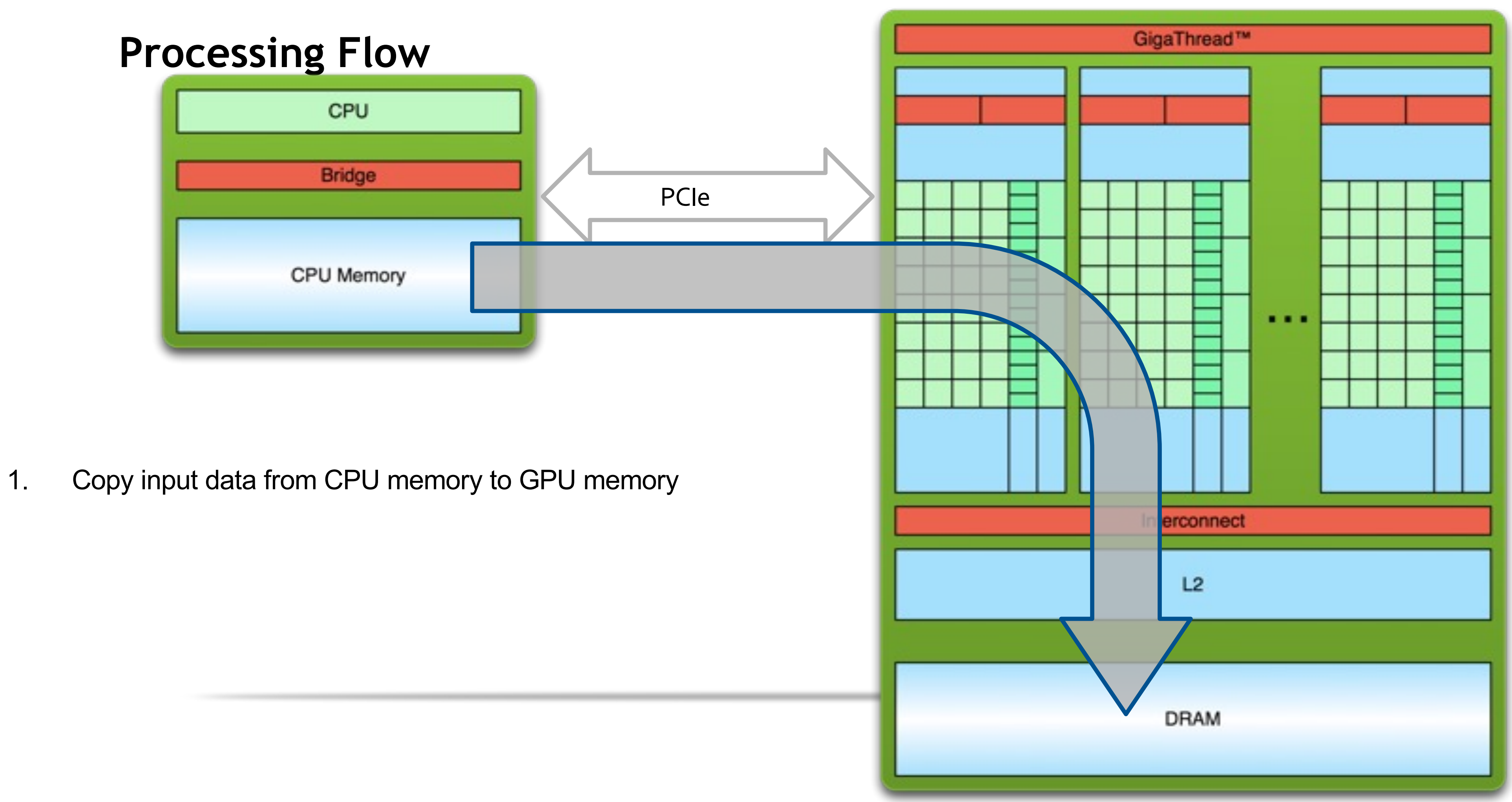
Host

Device

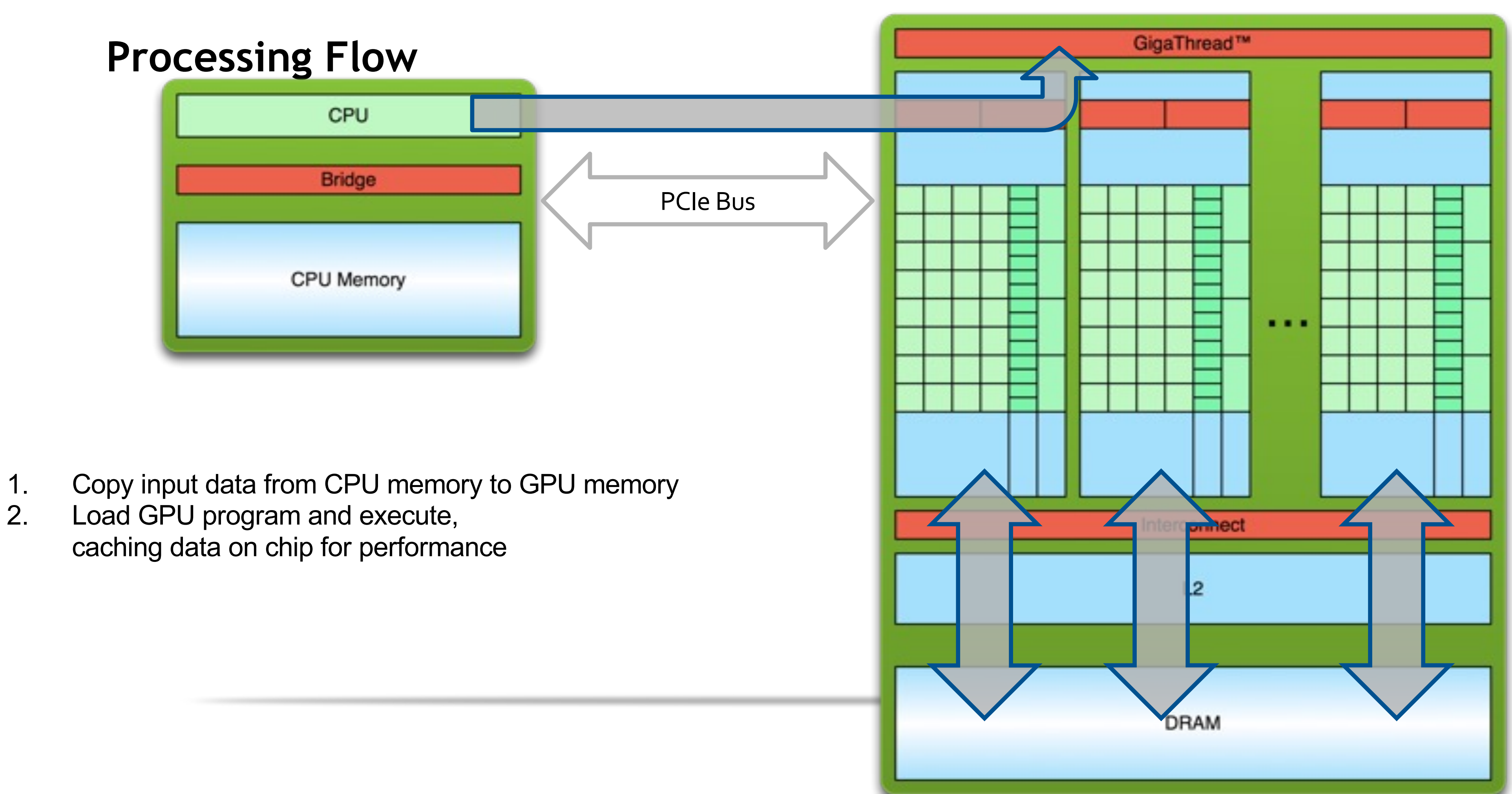
# GPGPU Accelerate Computing

*Latency Processor + Throughput processor*

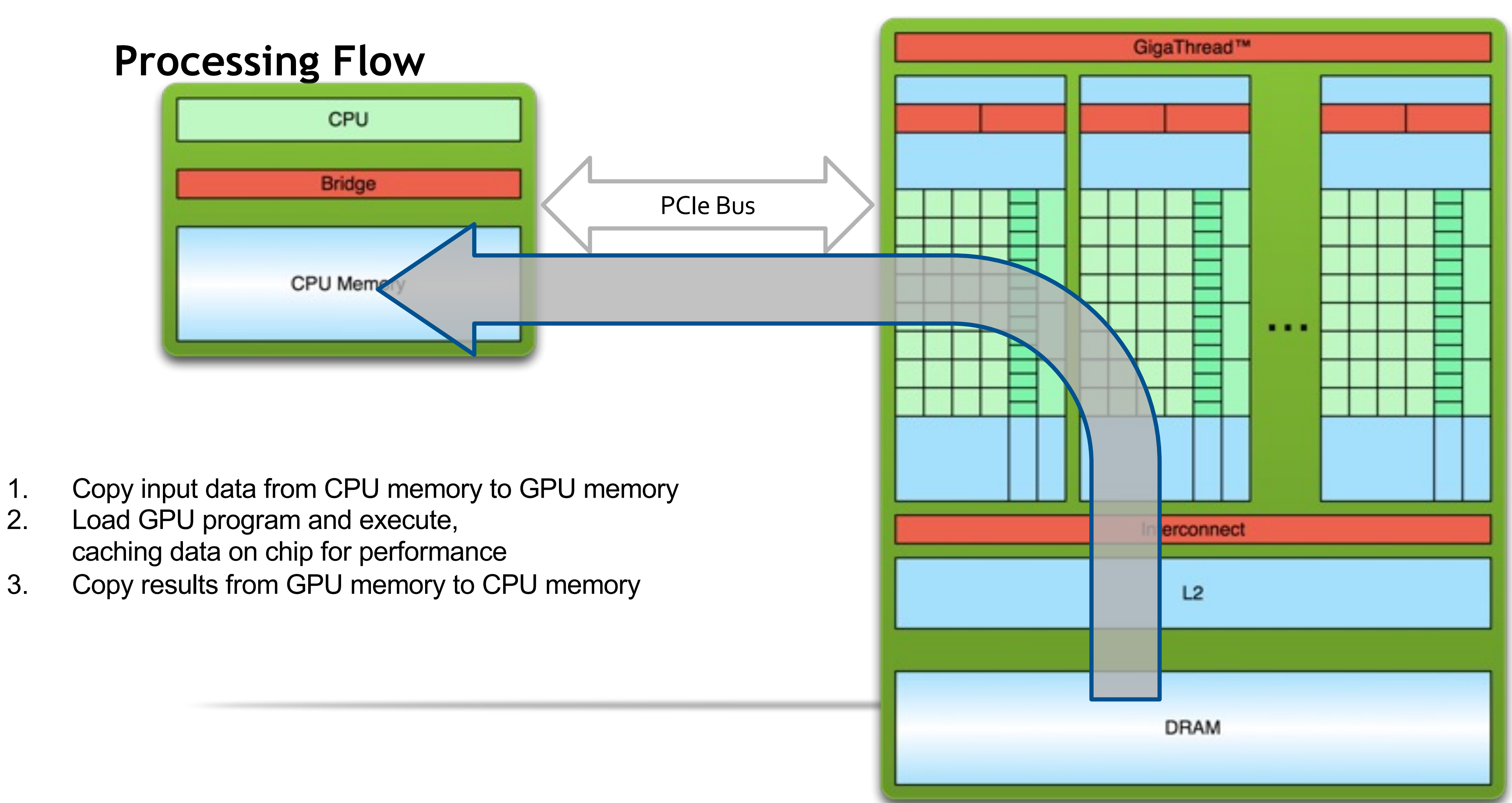



Processing Flow
CPU
Bridge
CPU Memory
PCIe
GigaThread™
Interconnect
L2
DRAM
1. Copy input data from CPU memory to GPU memory

# Processing Flow



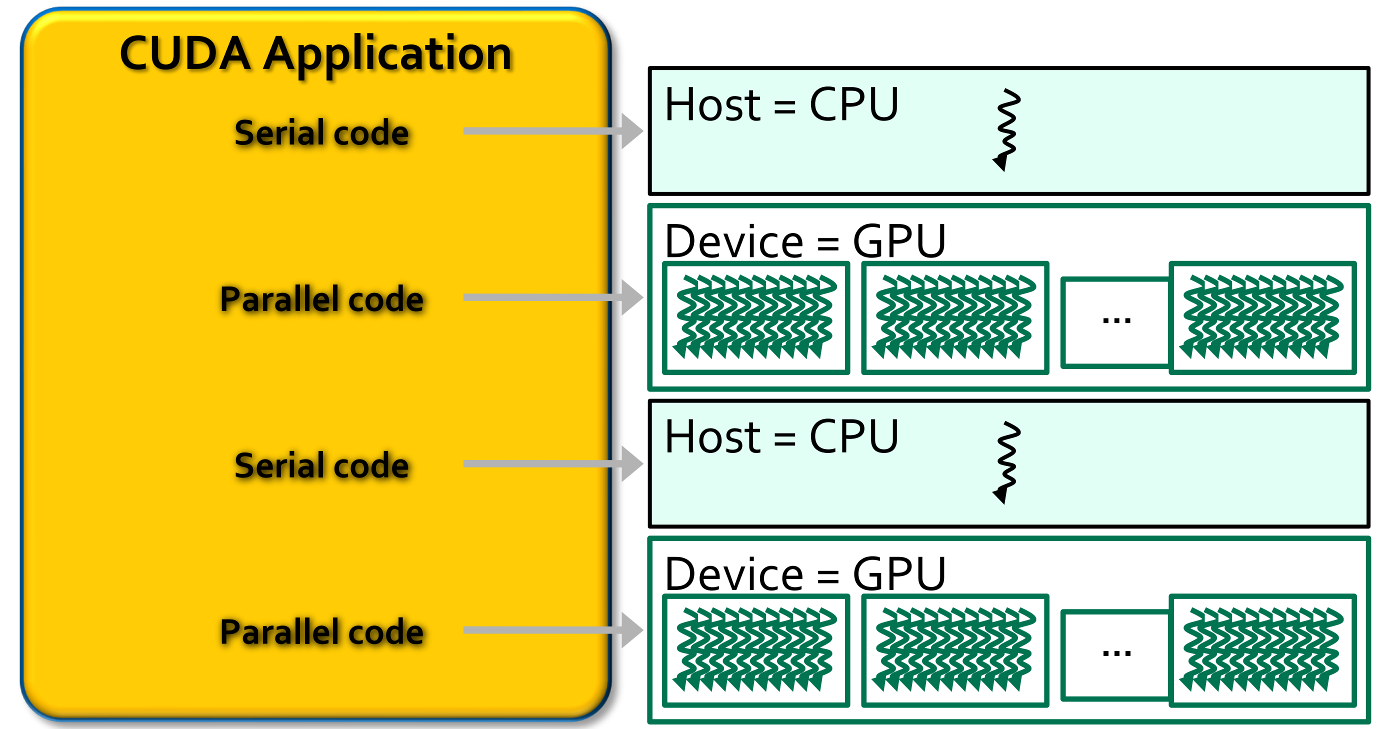

1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute,
   caching data on chip for performance


Processing Flow
CPU
Bridge
CPU Memory
PCIe Bus
GigaThread™
Interconnect
L2
DRAM
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute,
caching data on chip for performance
3. Copy results from GPU memory to CPU memory

# 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 |

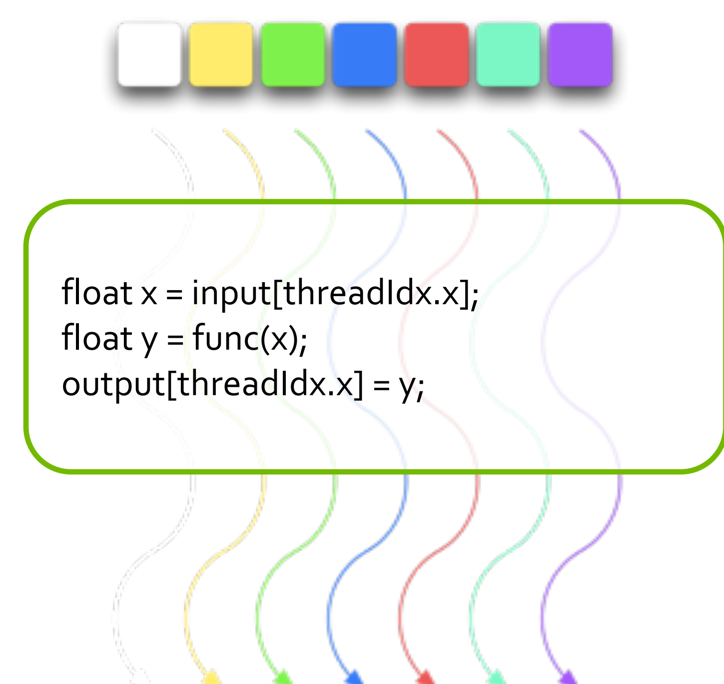
# CUDA Kernels: Parallel Threads

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

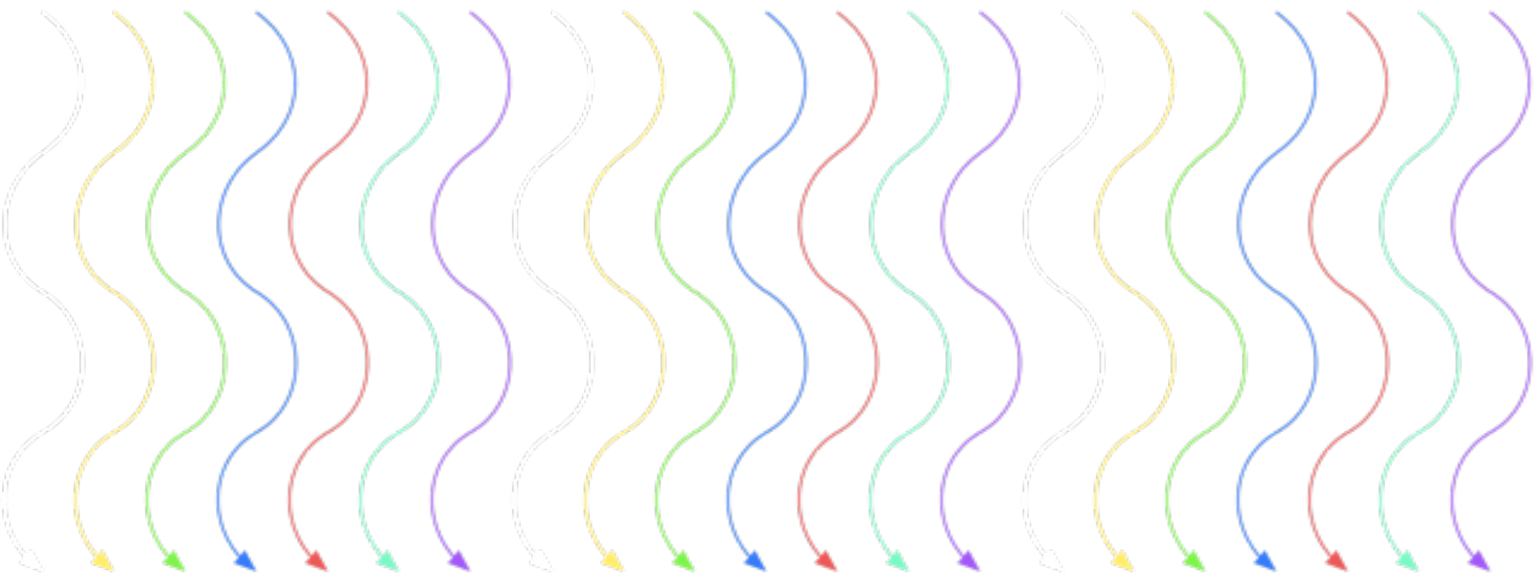
All threads execute the same code, can take different paths

Each thread has an ID

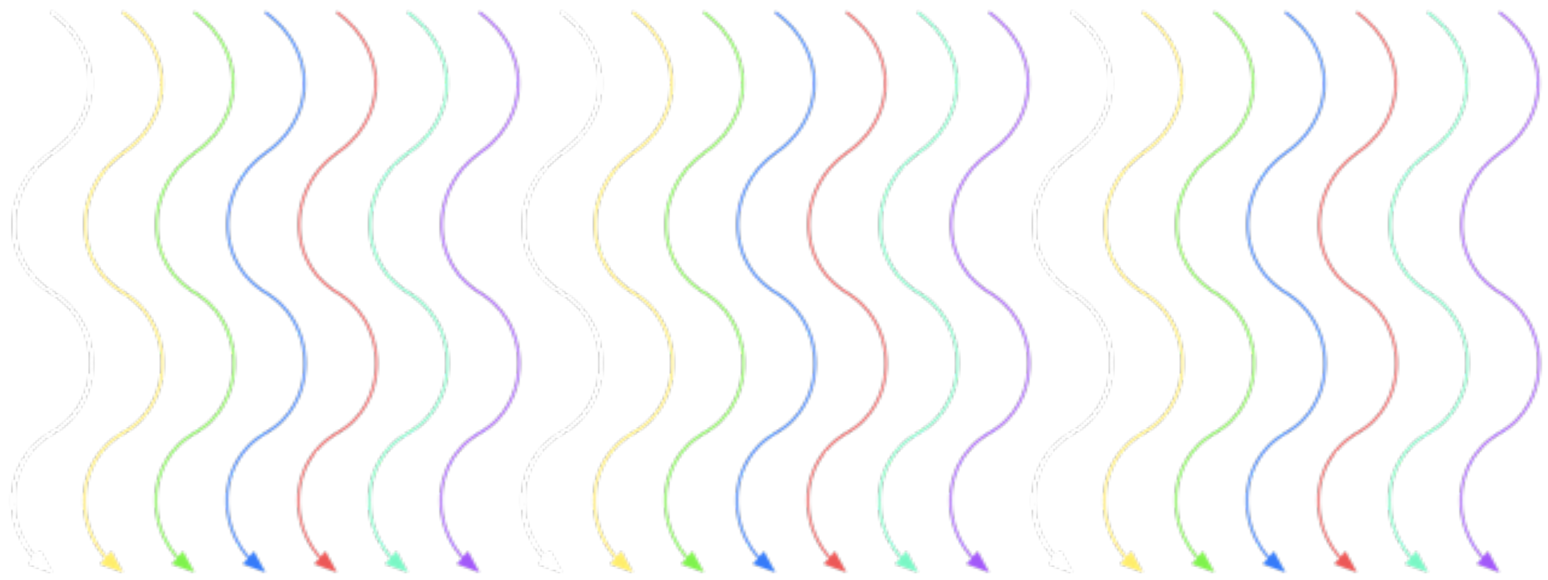
- Select input/output data
- Control decisions



# 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

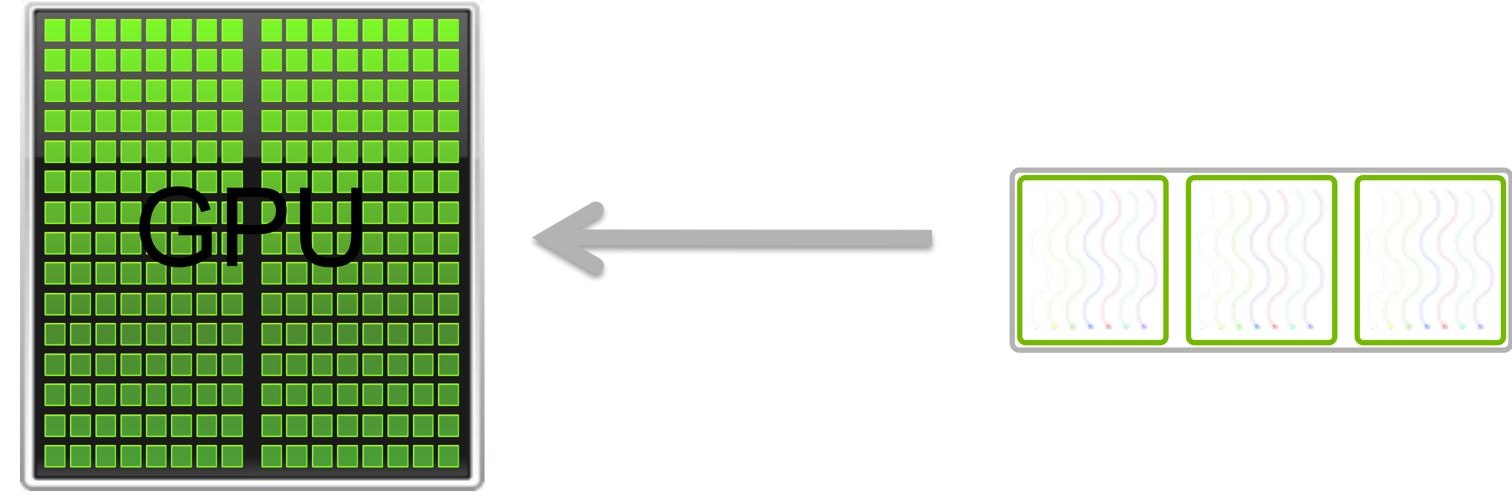
# CUDA Kernels: Subdivide into Blocks

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

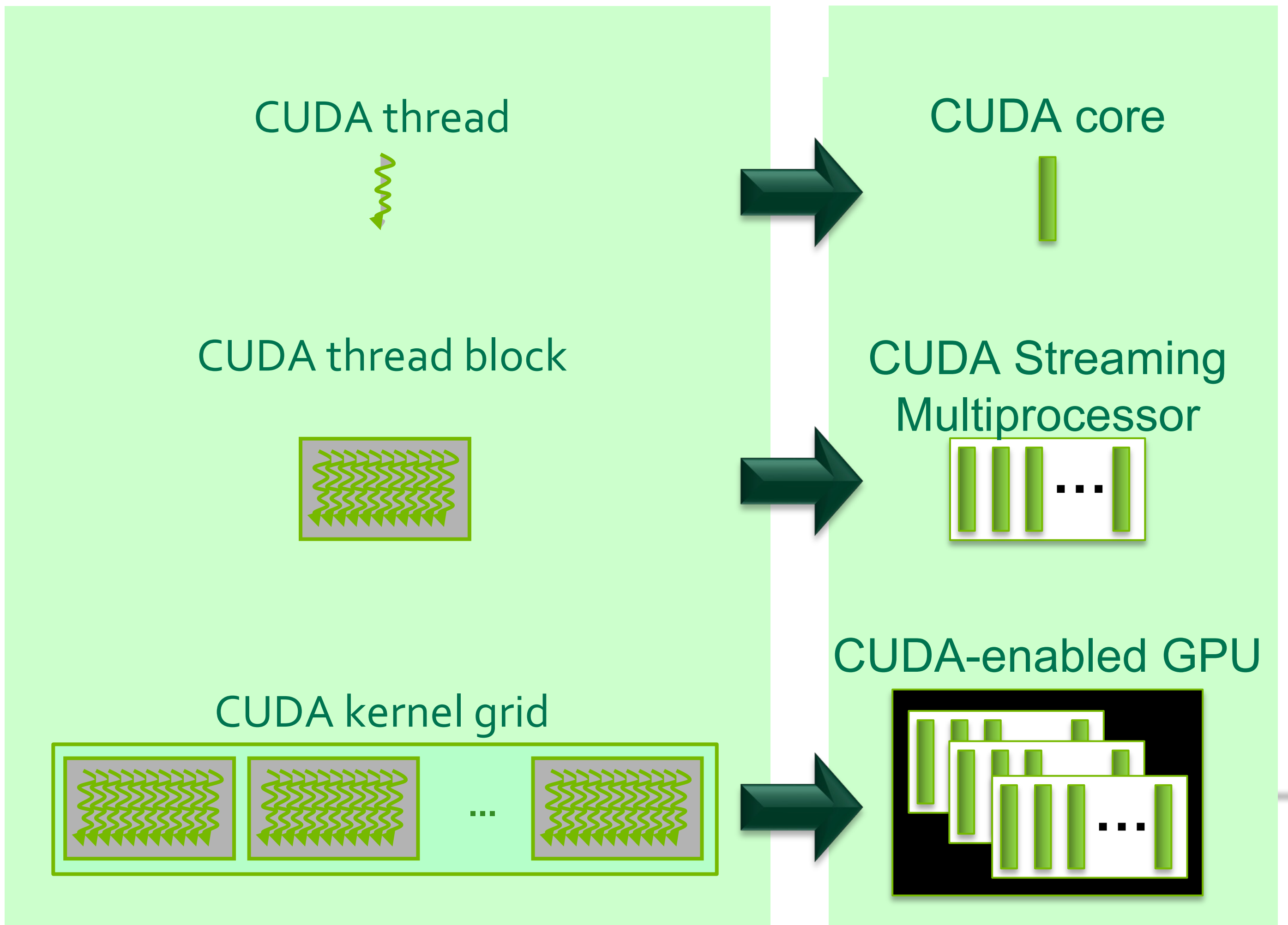



Threads are grouped into blocks

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:

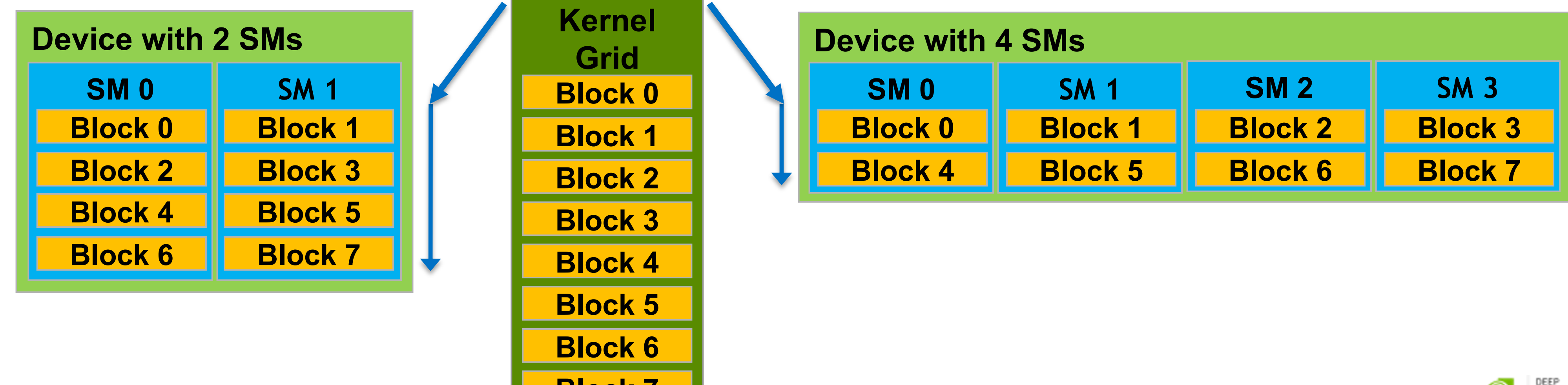
- Cooperatively load/store blocks of memory that they all use
- Share results with each other or cooperate to produce a single result
- Synchronize with each other

# 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

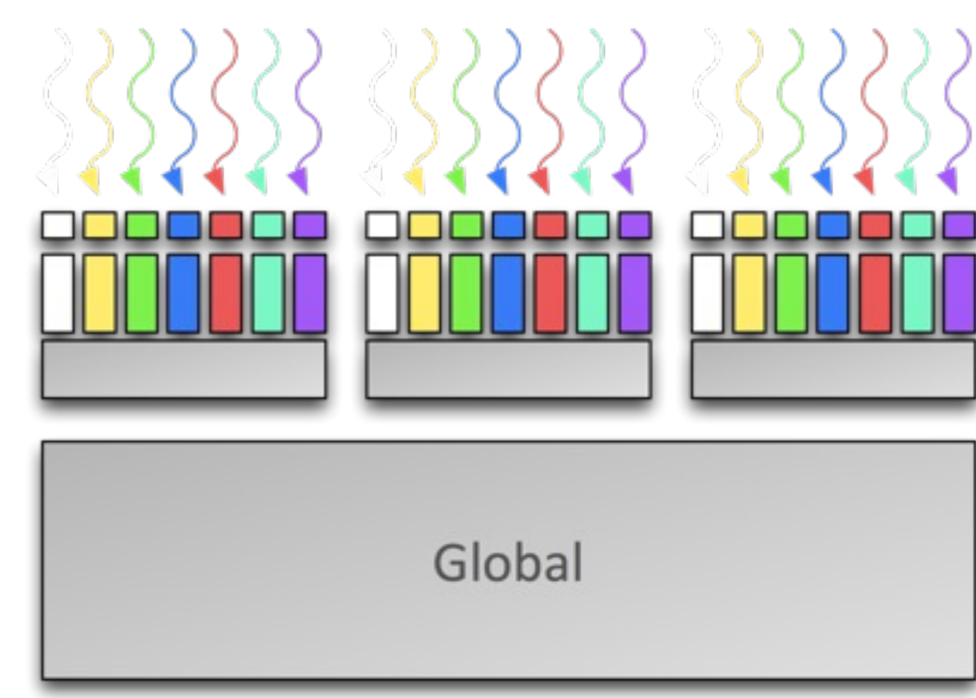
Local memory

Block of threads:

Shared memory

All blocks:

Global memory



# 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

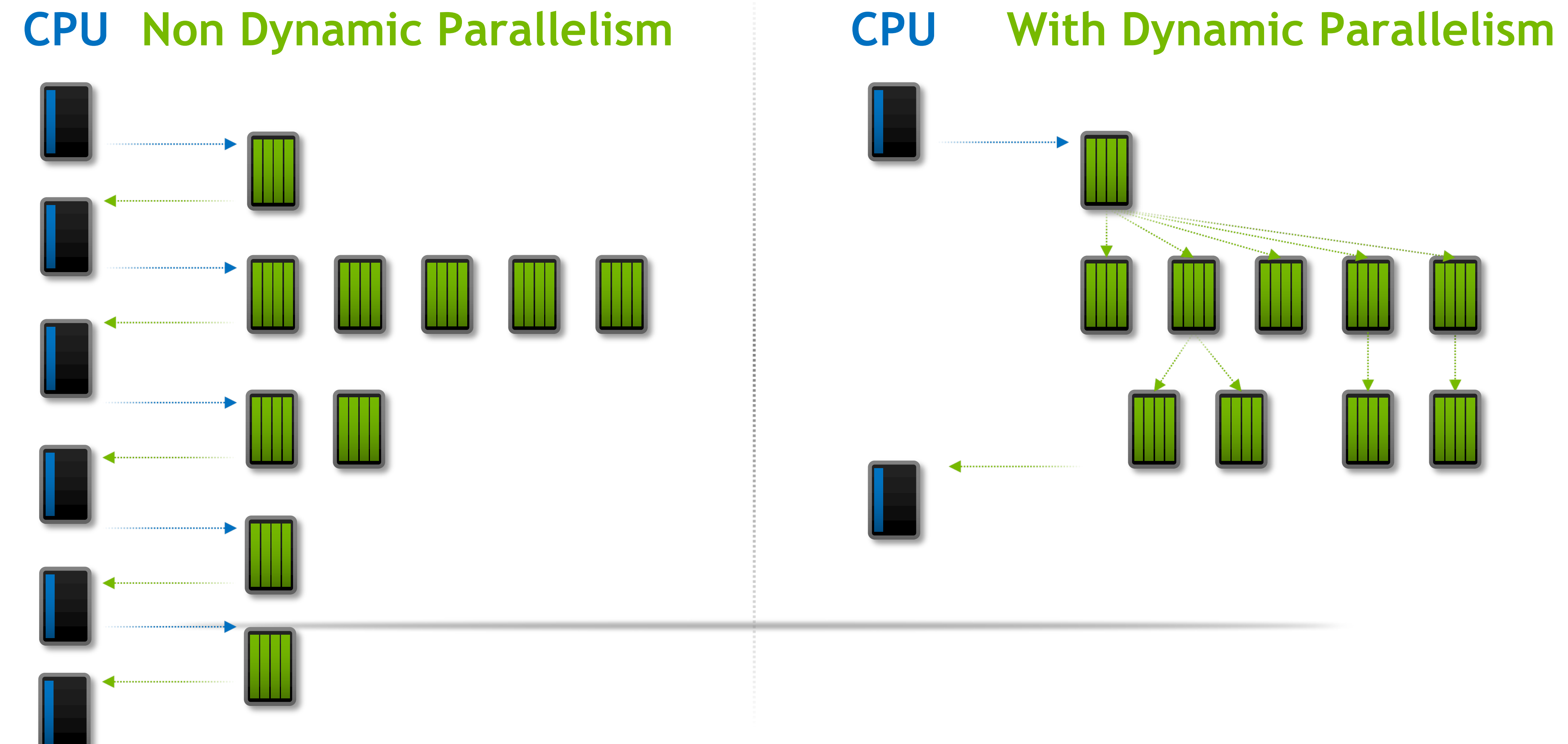
## *Copperhead: Parallel Python*

CU

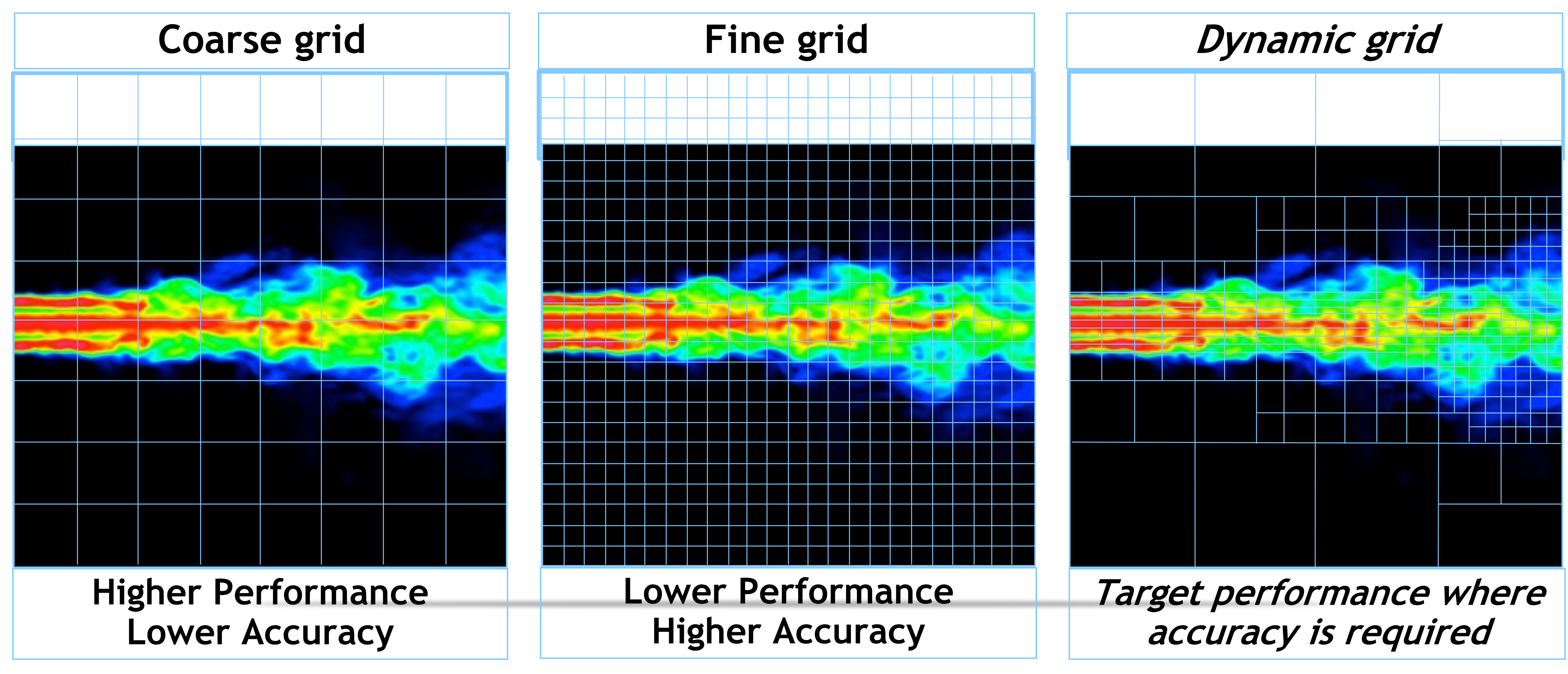
```
from copperhead import *
import numpy as np

@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

# Dynamic Parallelism



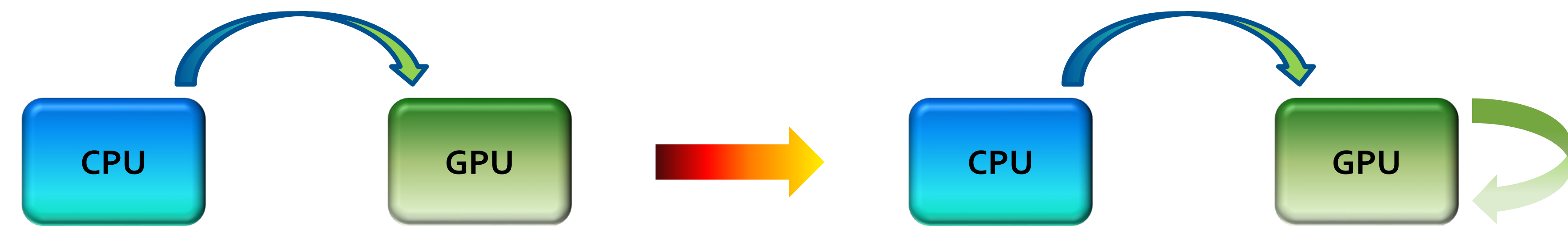
# Dynamic Work Generation



# 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);
}
```

CPU

GPU

A

B

C

X

Y

Z

# 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

NVIDIA DEEP LEARNING INSTITUTE

# 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

NVIDIA DEEP LEARNING INSTITUTE

# 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

# 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.

# Working at home!

Follow the next tutorial using HPC facilities in the university:

https://cuda-tutorial.readthedocs.io/en/latest/

DEEP
LEARNING
INSTITUTE

www.nvidia.com/dli