Massive Parallel Processing Opportunities for Data Analytics and Visualization Using CUDA

Carlos Jaime Barrios Hernandez, PhD.
@carlosjaimebh
Supercomputación y Cálculo Cientifico
Universidad Industrial de Santander
http://www.sc3.uis.edu.co @sc3uis

NVIDIA CUDA TEACHING CENTER
NVIDIA CUDA RESEARCH CENTER

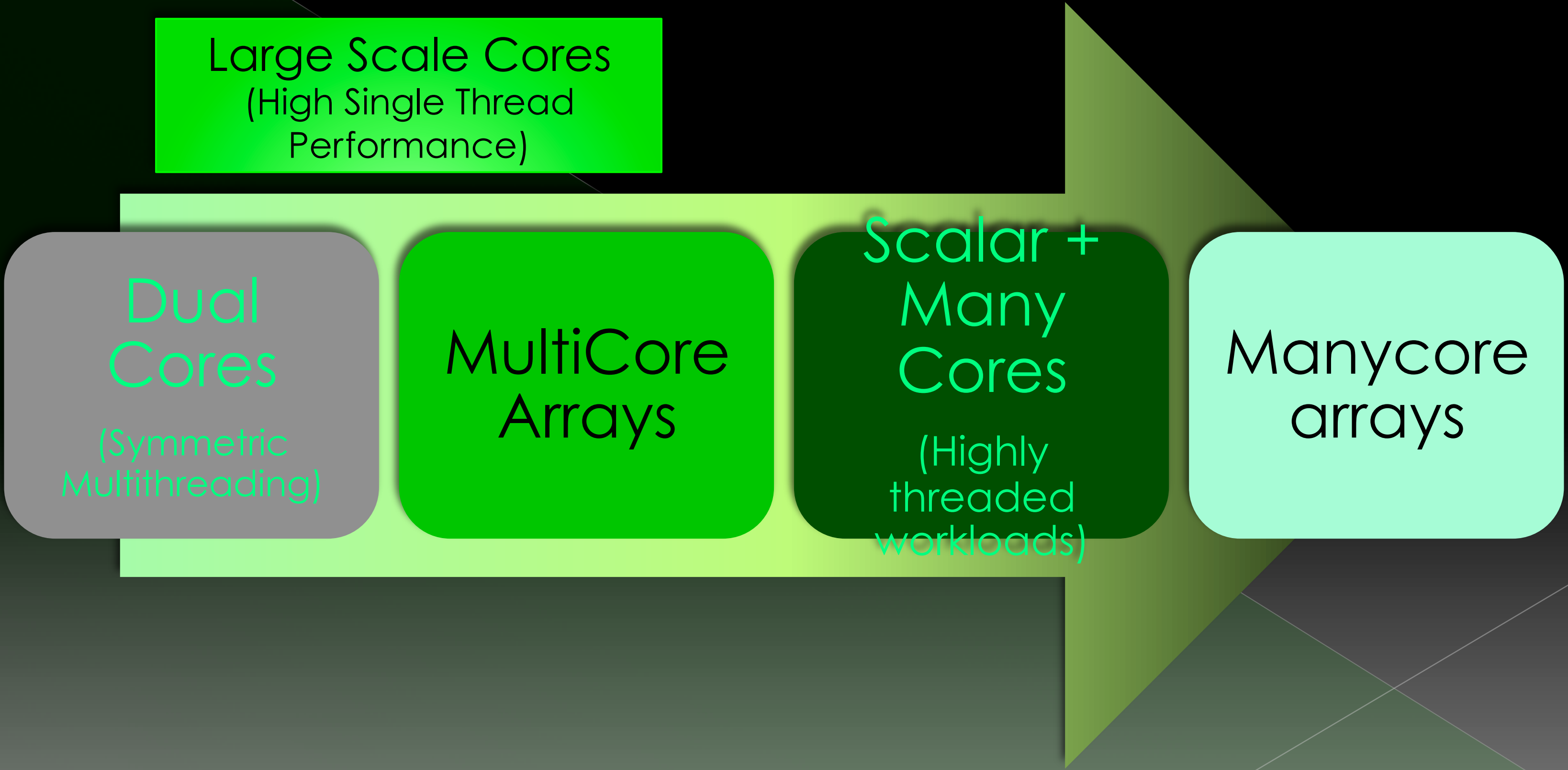
Universidad Industrial de Santander
CONSTRUIMOS FUTURO

Super Computacion y Calculo Cientifico UIS

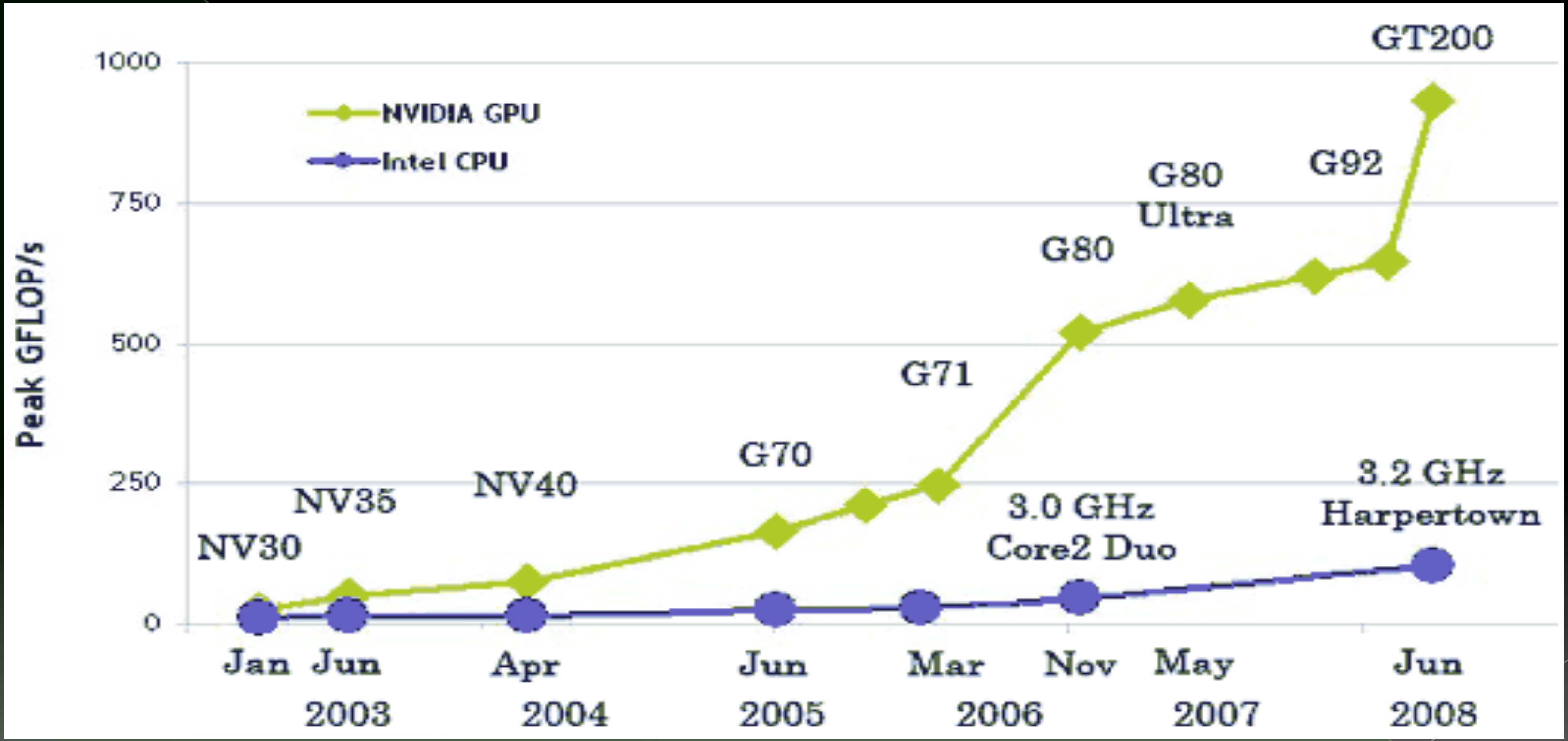
Prerequisites

- You need experience with C
- You don’t need GPU experience
- You don’t need parallel programming experience (It’s not really true, if you have is better!!!)
- You don’t need graphics experience

Evolution of Configurable Architecture

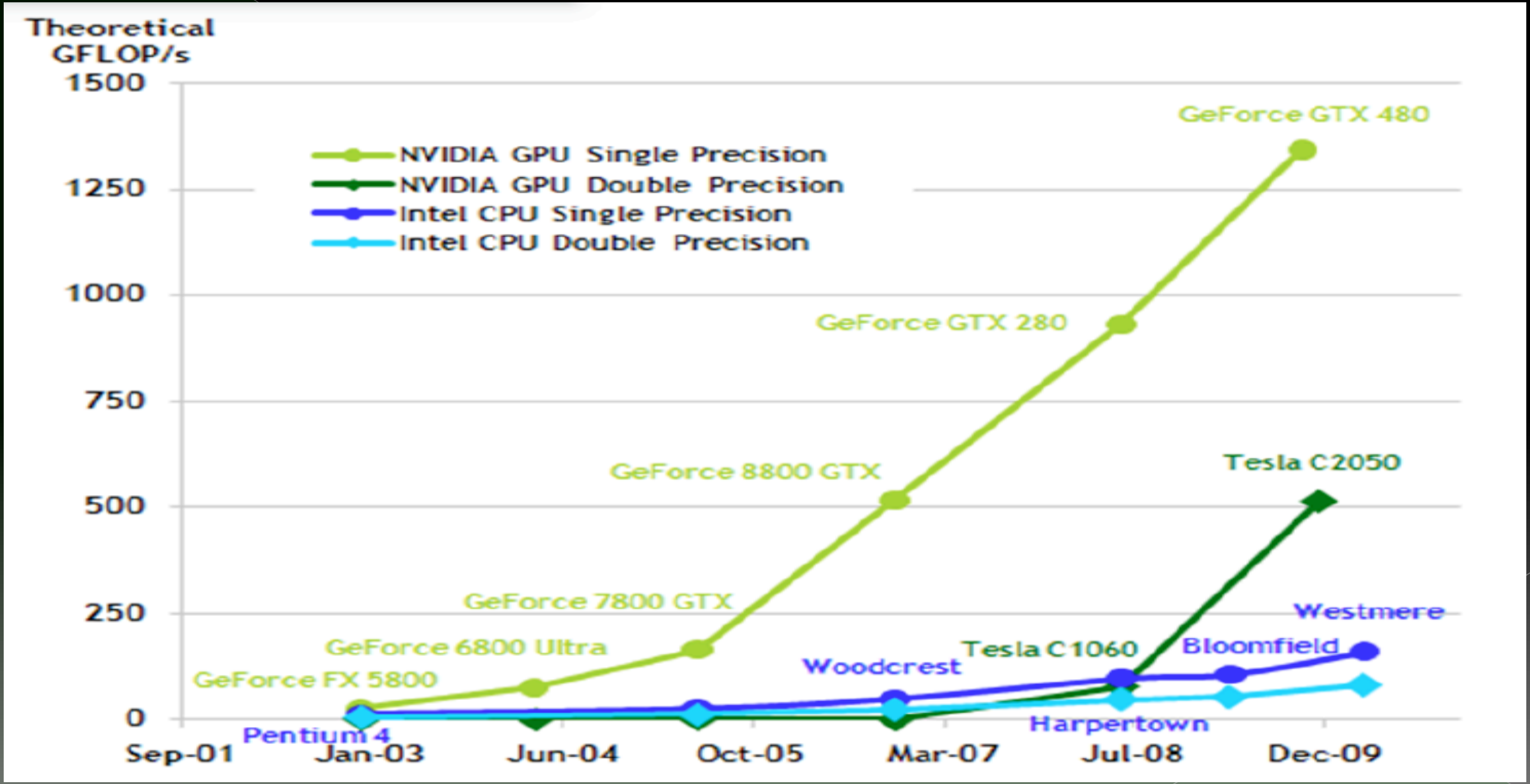

Large Scale Cores
(High Single Thread Performance)
Dual Cores
(Symmetric Multithreading)
MultiCore Arrays
Scalar + Many Cores
(Highly threaded workloads)
Manycore arrays

GPUs vs CPUs Performance



http://ixbtlabs.com/articles3/video/cuda-1-p1.ht

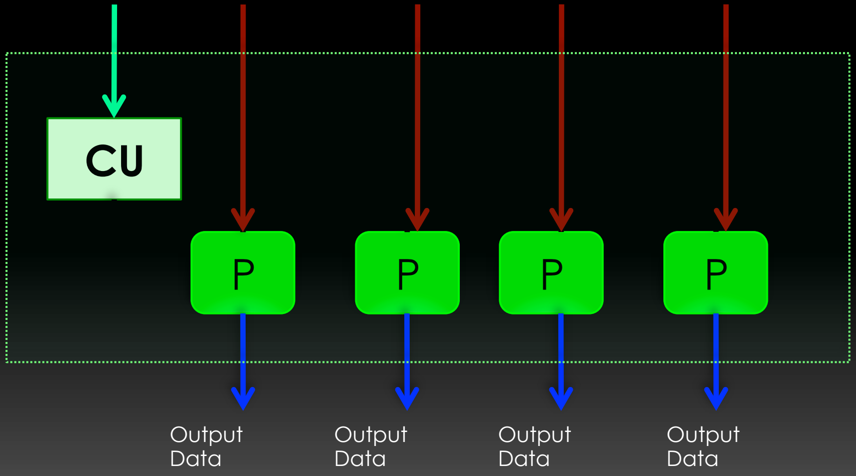
GPUs vs CPUs


Theoretical GFLOP/s
1500
1250
1000
750
500
250
0
Sep-01
Jan-03
Jun-04
Oct-05
Mar-07
Jul-08
Dec-09
NVIDIA GPU Single Precision
NVIDIA GPU Double Precision
Intel CPU Single Precision
Intel CPU Double Precision
GeForce GTX 480
GeForce GTX 280
GeForce 8800 GTX
GeForce 7800 GTX
GeForce 6800 Ultra
GeForce FX 5800
Tesla C2050
Tesla C1060
Westmere
Bloomfield
Woodcrest
Harpertown
Pentium 4

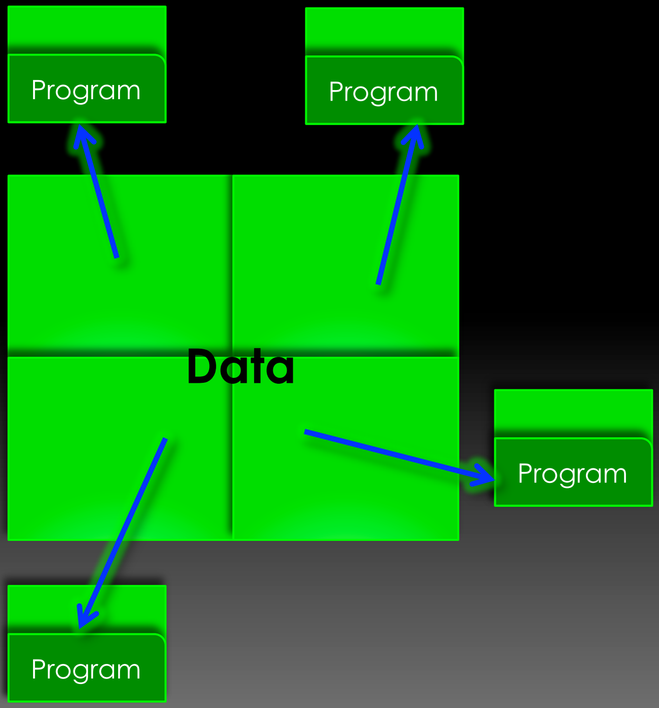
Architectural Systems Facts

SPMD: Parallel Processing Units execute the same program on multiple parts of data

SIMD: All processors units are executing the same instructions in any instant.



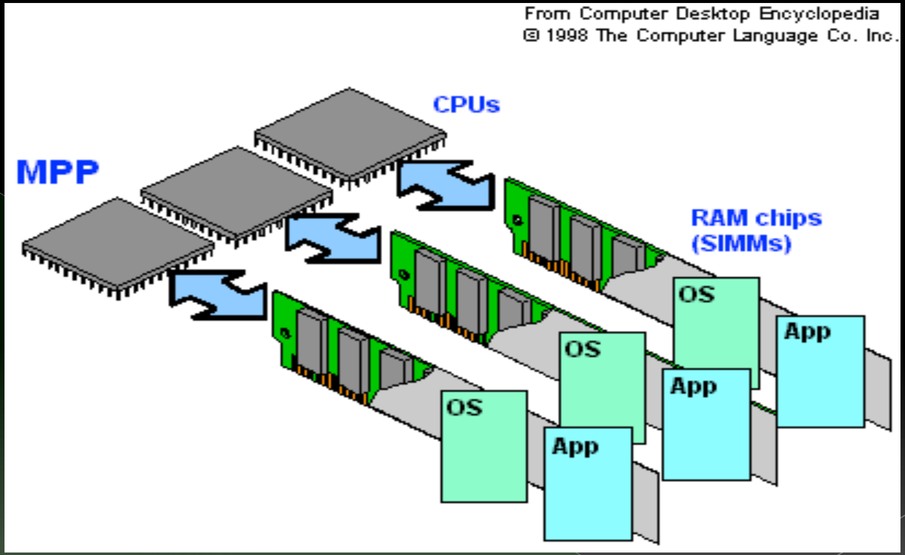
SIMD



Processor

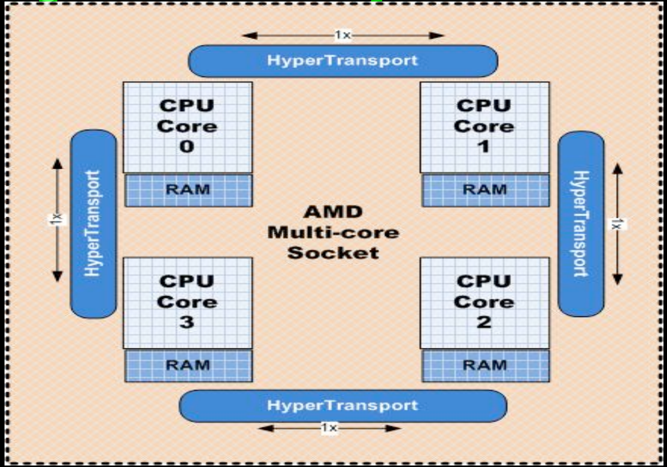
Massive Parallel Processing (MPP)

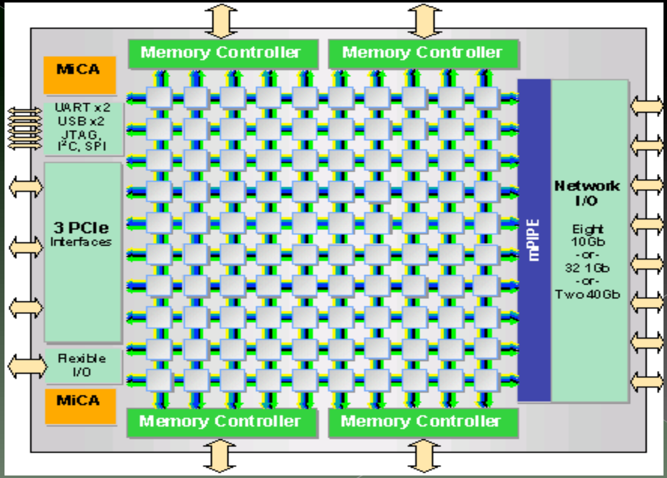
- Computer system with many independent arithmetic units or entire microprocessors, that run in parallel
- MPPA is a MIMD (Multiple Instruction streams, Multiple Data) architecture, with distributed memory accessed locally, not shared globally


From Computer Desktop Encyclopedia
© 1998 The Computer Language Co. Inc.
CPUs
MPP
RAM chips (SIMMs)
OS
App
OS
App
OS
App

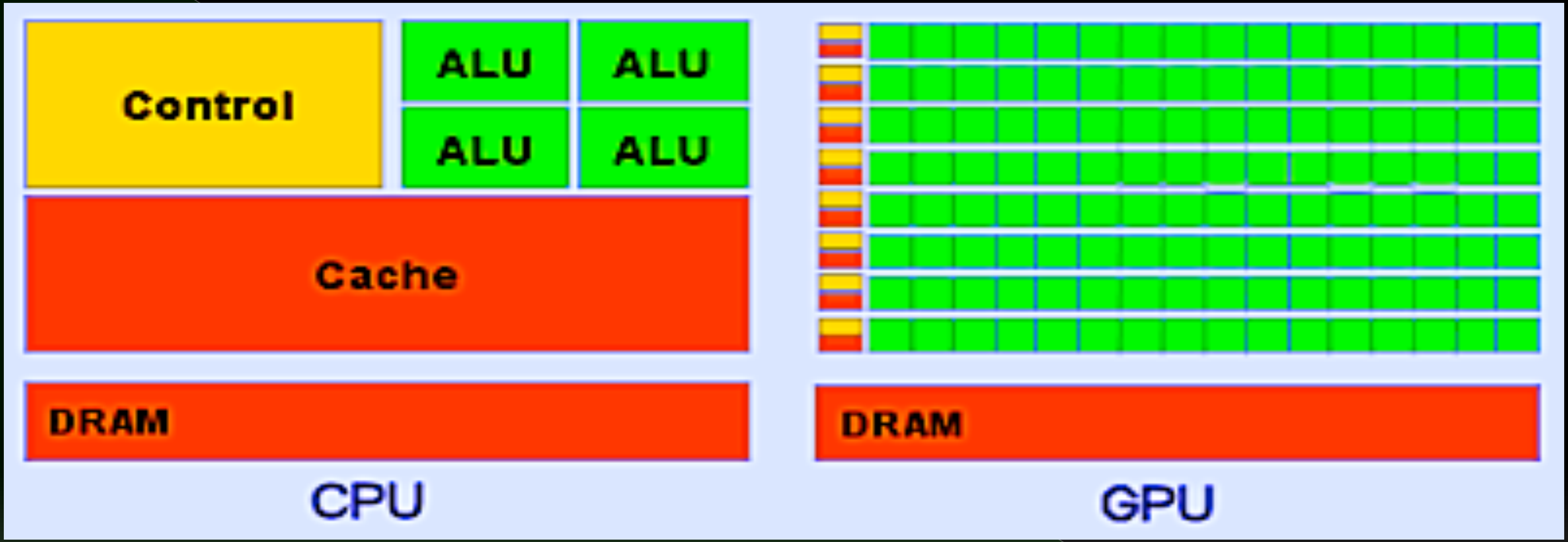
Microprocessor Trajectory

- Multicore: Execution speed of sequential programs while moving into multiple cores.
- Many-core: Execution throughput of parallel applications.

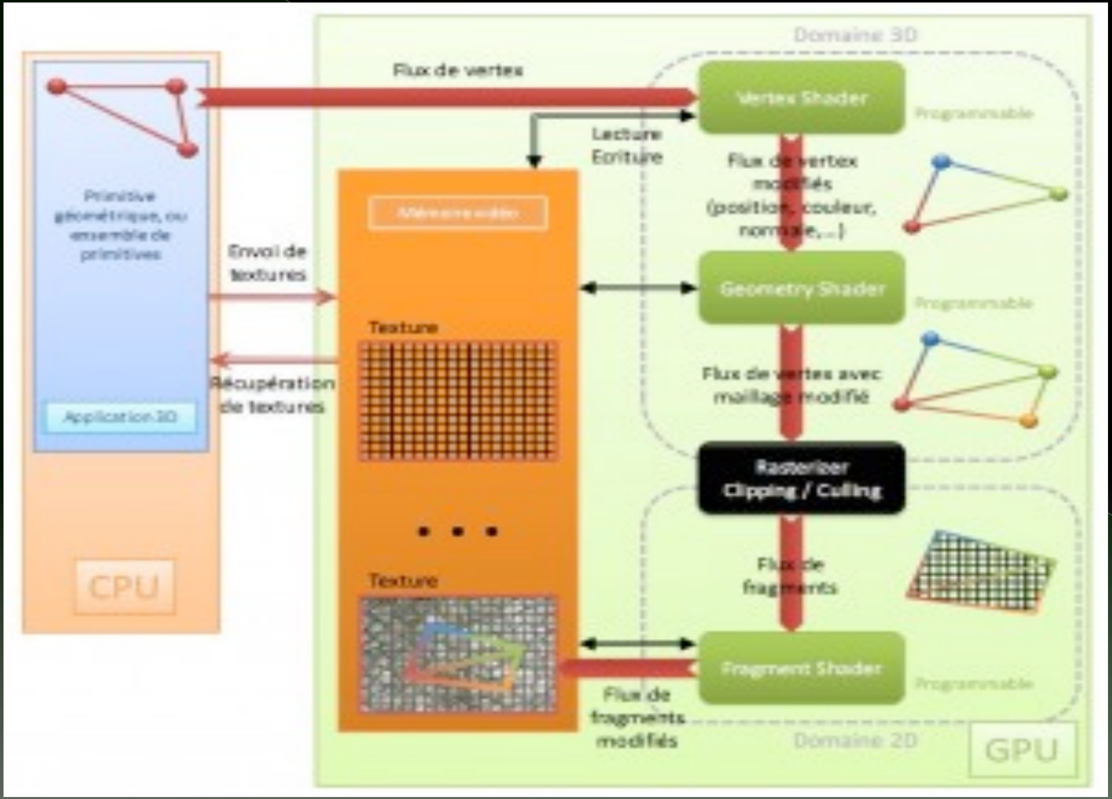


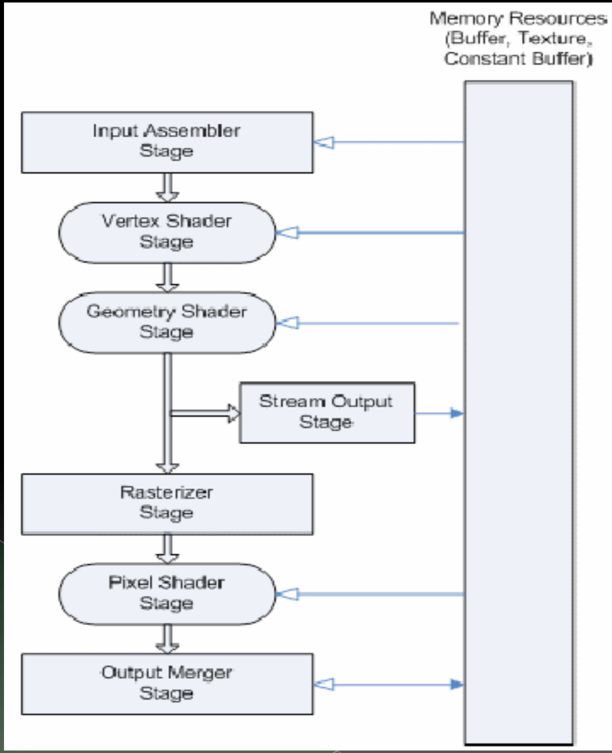


CPUs and GPUs Design



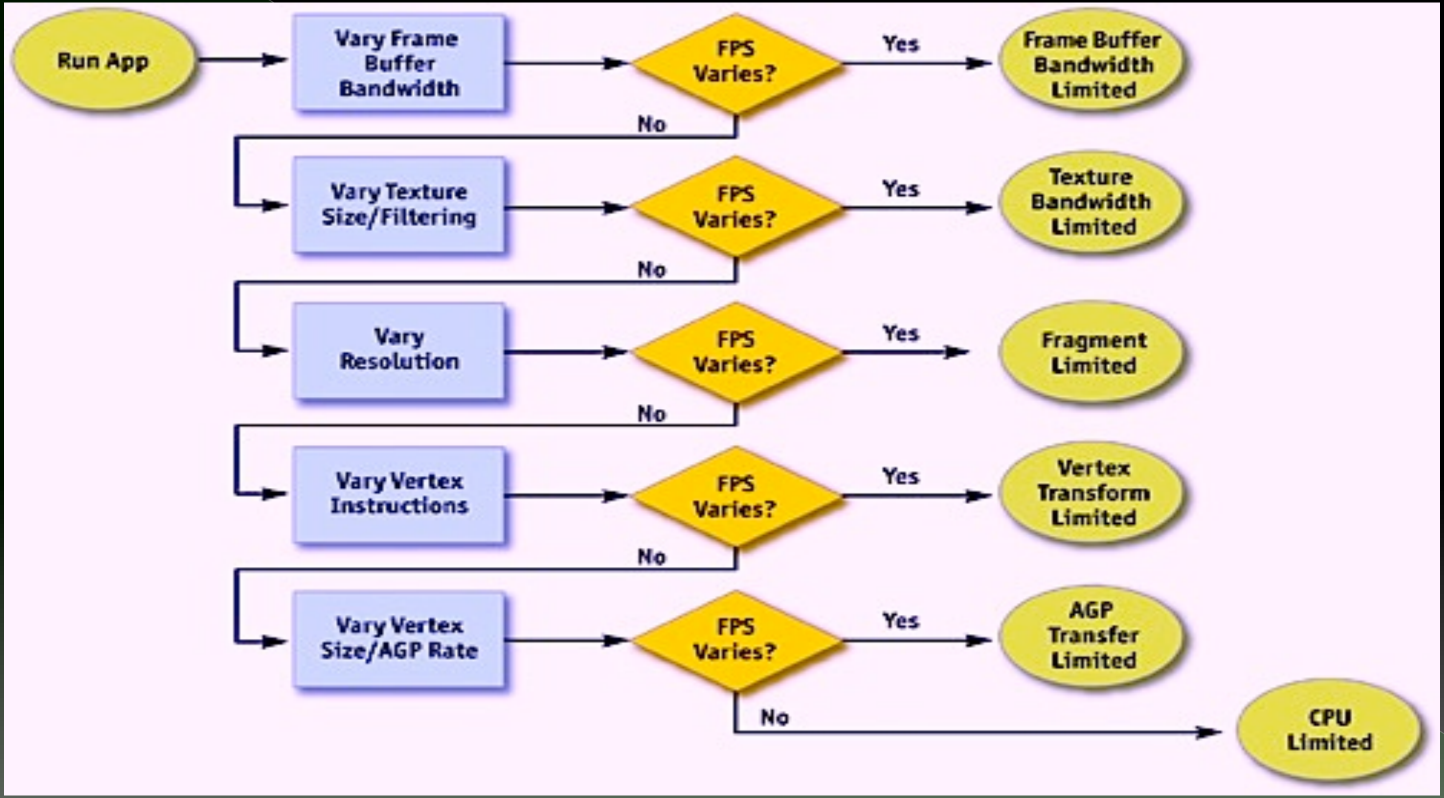
GPU Graphics Pipeline





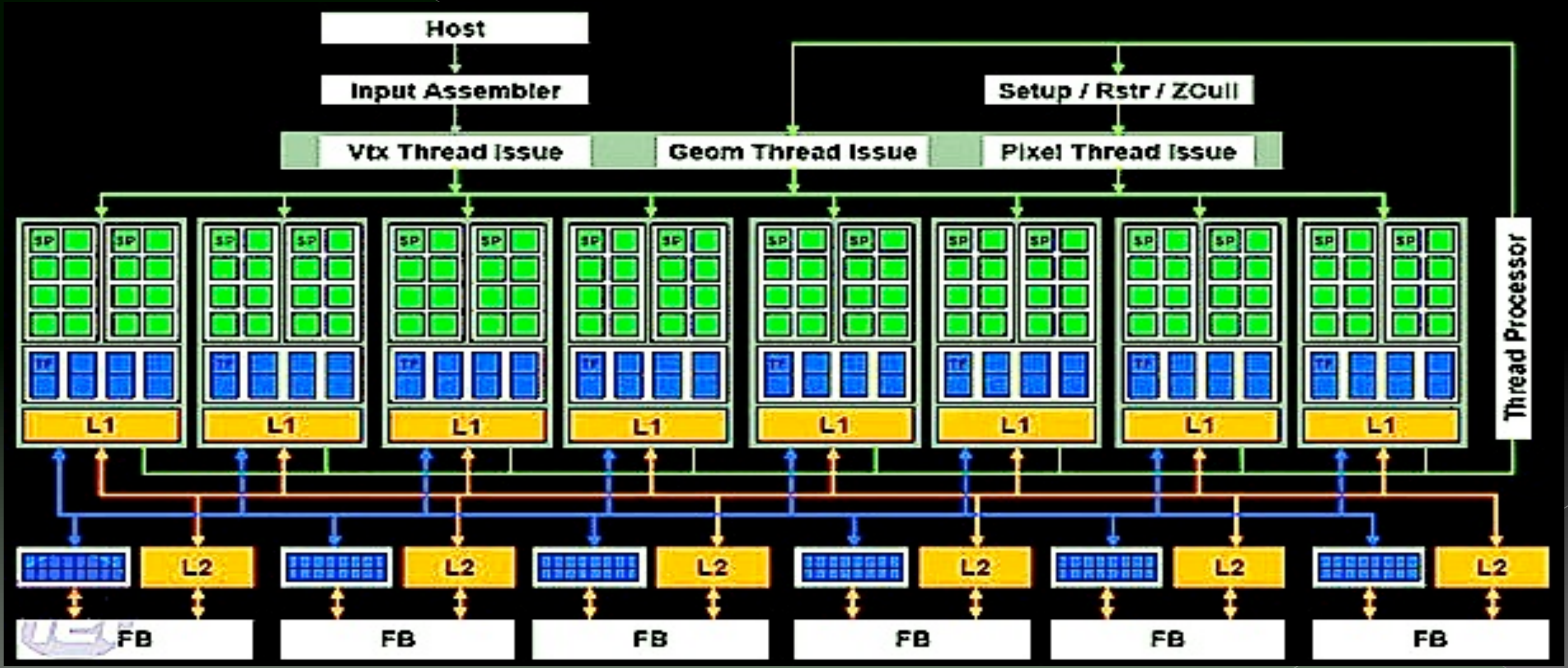
From http://developer.nvidia.com/page/home.html

Bottleneck Flow



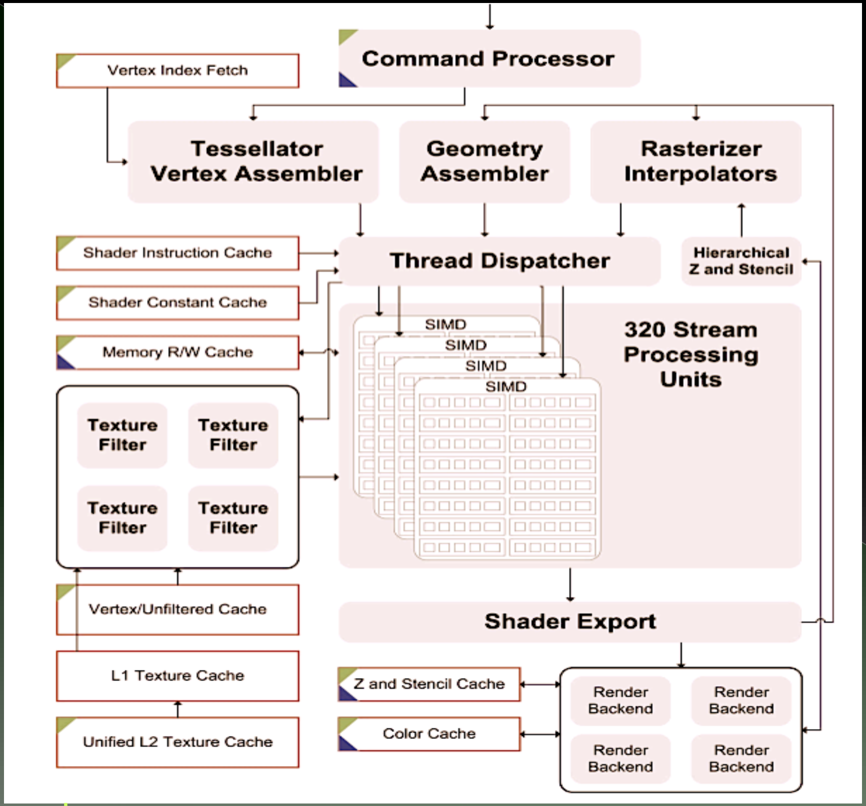
From http://developer.nvidia.com/page/home.html

Nvidia™ GeForce™ 8080 Pipeline



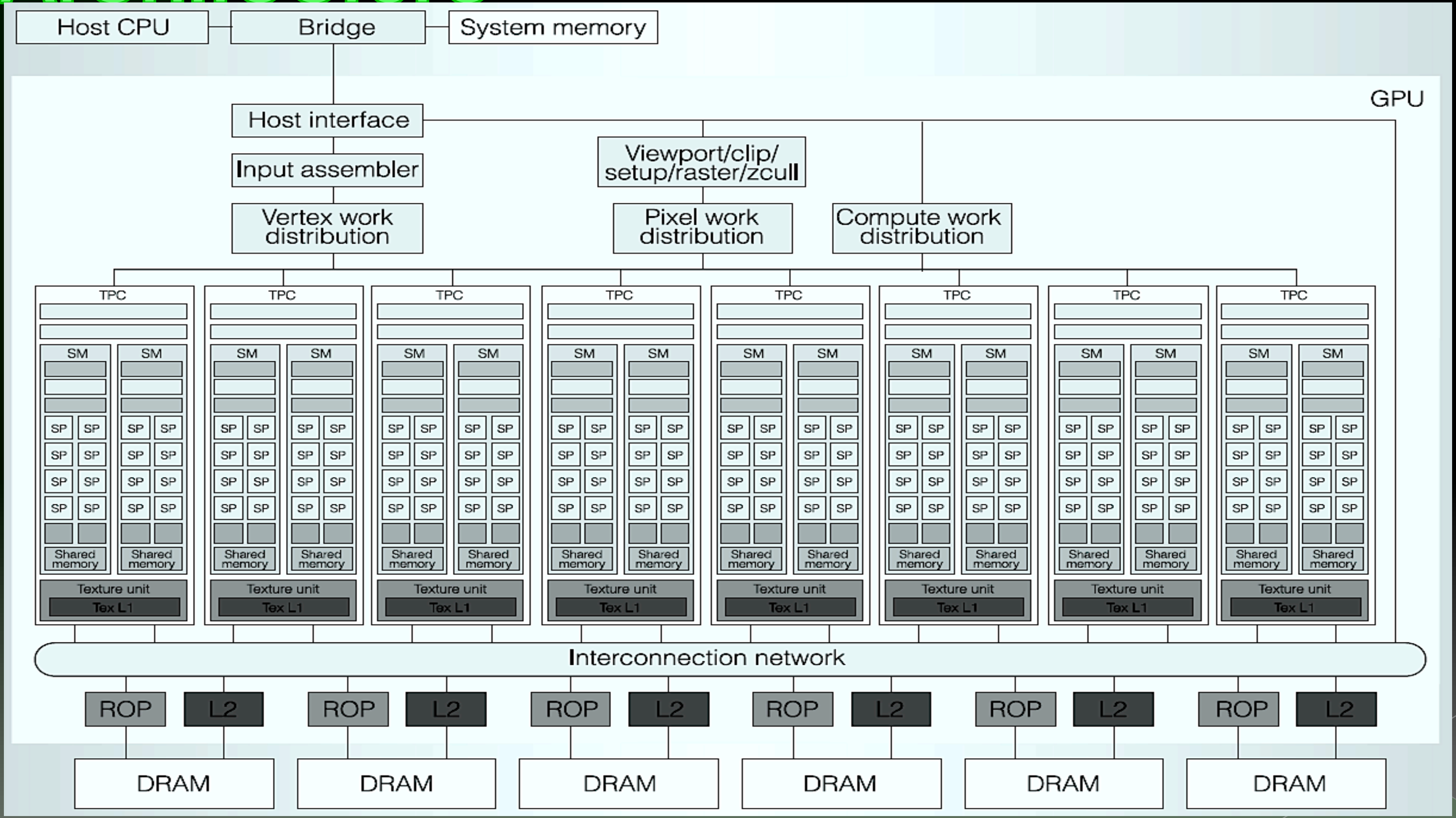
From http://www.nvidia.com/object/cuda_home_new.html

AMD™ 's RADEON™ HD2900XT Pipeline



From www.amd.com

TESLA™ Graphics and Computing Architecture



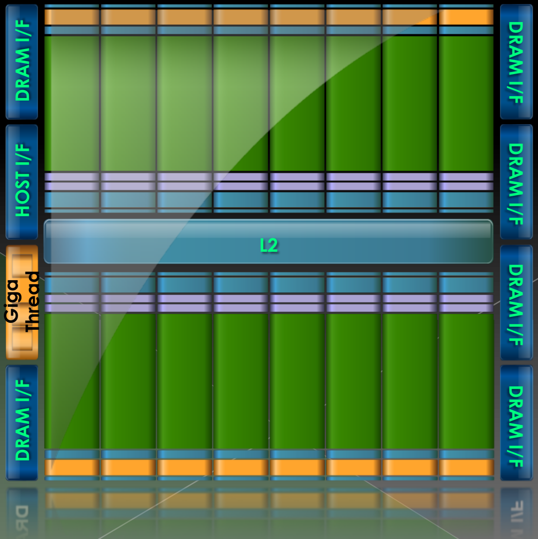
From http://www.nvidia.com/object/cuda_home_new.html

TESLA™ Graphics and Computing Architecture Features

- TESLA™ shader processors are fully programmable
  - Large instructions memory
  - Cache Instructions
  - Logic Sequence Instructions
- TESLA™ to non-graphics programs:
  - Hierarchical Parallel Threads
  - Barrier Synchronization
  - Atomic Operators (Manage Highly Parallel Computing Work)

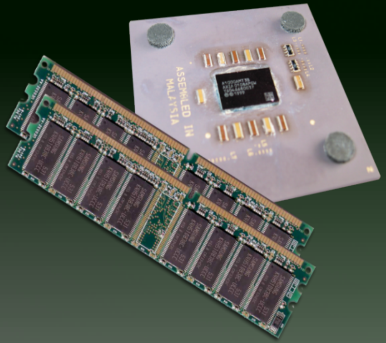
GPU Architecture: Two Main Components

- Global memory
  - Analogous to RAM in a CPU server
  - Accessible by both GPU and CPU
  - Currently up to 6 GB
  - Bandwidth currently up to 150 GB/s for Quadro and Tesla products
  - ECC on/off option for Quadro and Tesla products
- Streaming Multiprocessors (SMs)
  - Perform the actual computations
  - Each SM has its own:
    - Control units, registers, execution pipelines, caches



Heterogeneous Computing

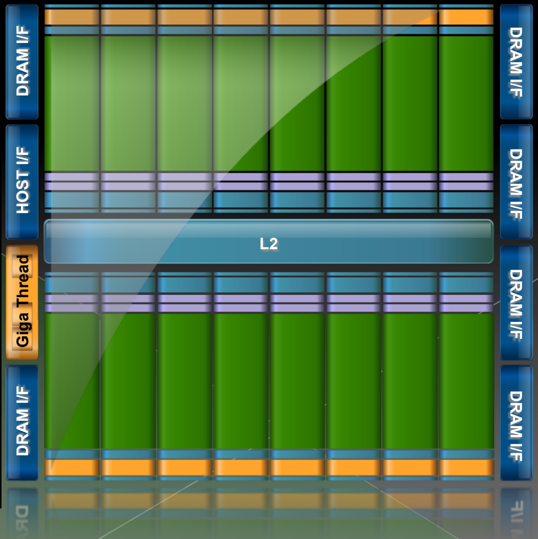
- Terminology:
  - *Host* The CPU and its memory (host memory)
  - *Device* The GPU and its memory (device memory)


Host

Device

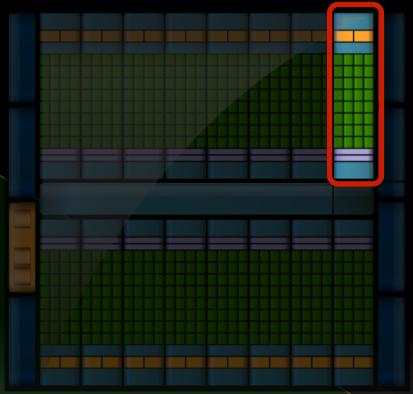
GPU Architecture: Two Main Components

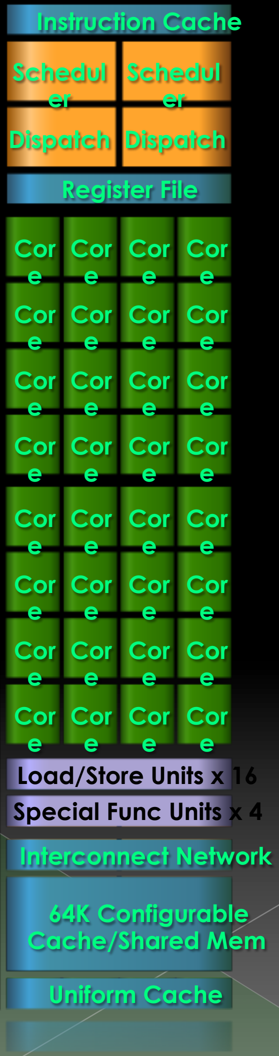
- Global memory
  - Analogous to RAM in a CPU server
  - Accessible by both GPU and CPU
  - Currently up to 6 GB
  - Bandwidth currently up to 150 GB/s for Quadro and Tesla products
  - ECC on/off option for Quadro and Tesla products
- Streaming Multiprocessors (SMs)
  - Perform the actual computations
  - Each SM has its own:
    - Control units, registers, execution pipelines, caches



GPU Architecture – Fermi: Streaming Multiprocessor (SM)

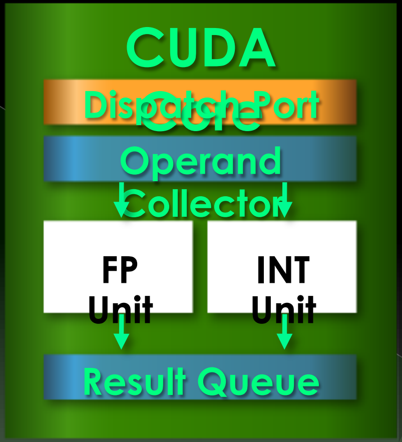
- 32 CUDA Cores per SM
  - 32 fp32 ops/clock
  - 16 fp64 ops/clock
  - 32 int32 ops/clock
- 2 warp schedulers
  - Up to 1536 threads concurrently
- 4 special-function units
- 64KB shared mem + L1 cache
- 32K 32-bit registers

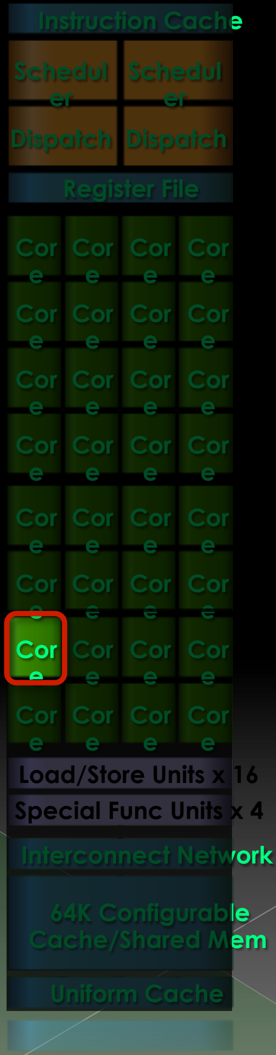




GPU Architecture – Fermi: CUDA Core

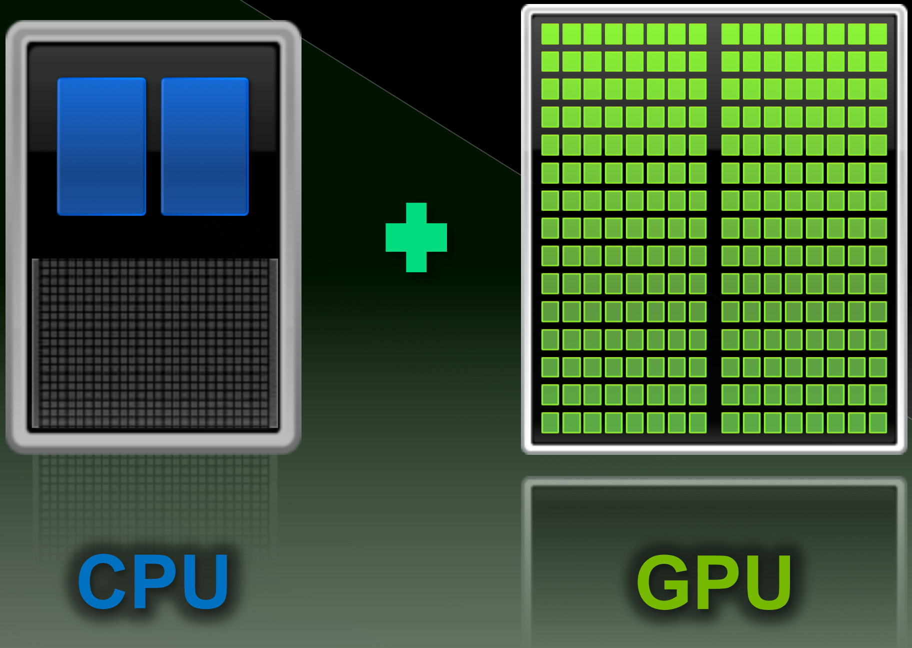
- Floating point & Integer unit
  - IEEE 754-2008 floating-point standard
  - Fused multiply-add (FMA) instruction for both single and double precision
- Logic unit
- Move, compare unit
- Branch unit



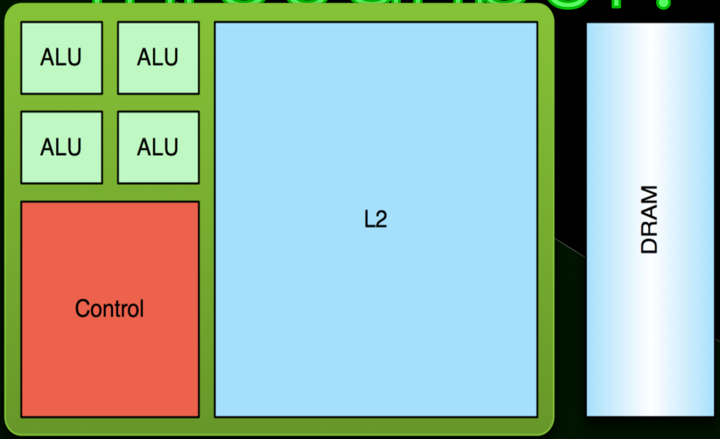


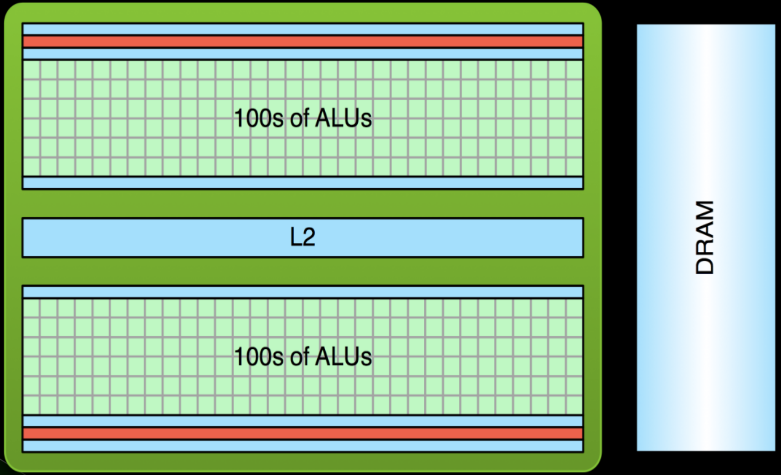
GPGPU Accelerate Computing

Latency Processor + Throughput processor



Low Latency or High Throughput?





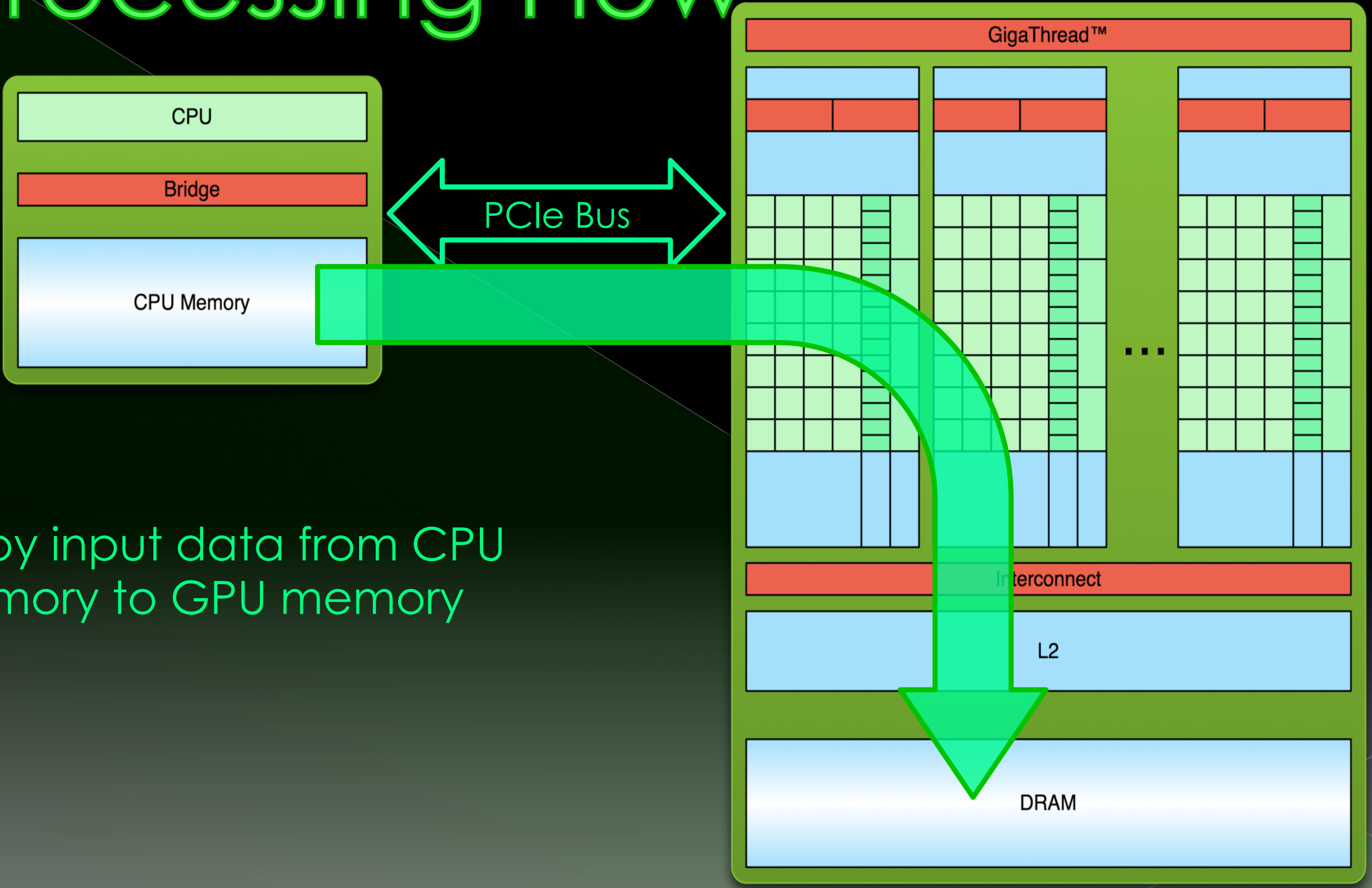
CPU

- Optimized for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution

GPU

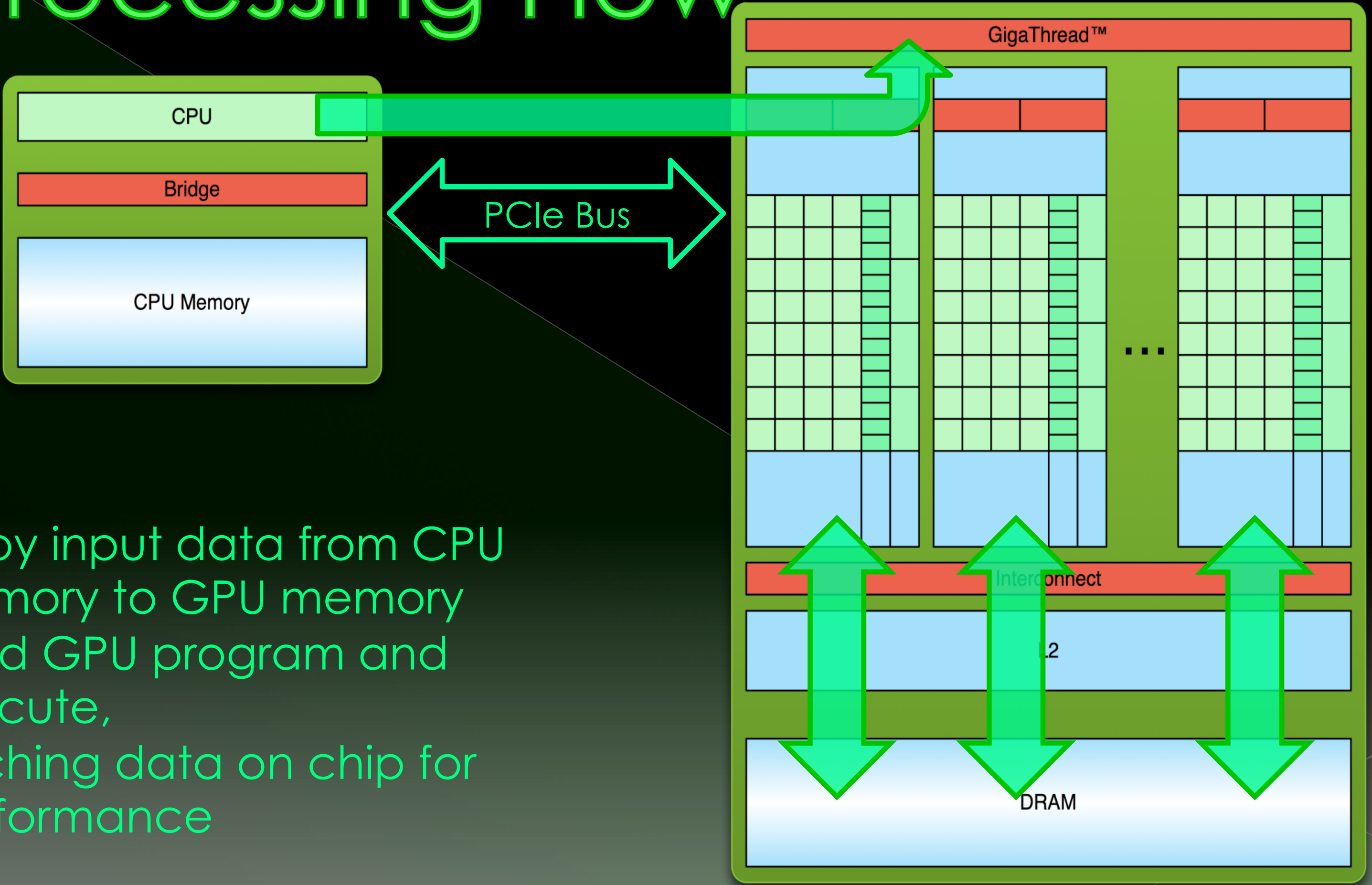
- Optimized for data-parallel, throughput computation
- Architecture tolerant of memory latency
- More transistors dedicated to computation

Processing Flow



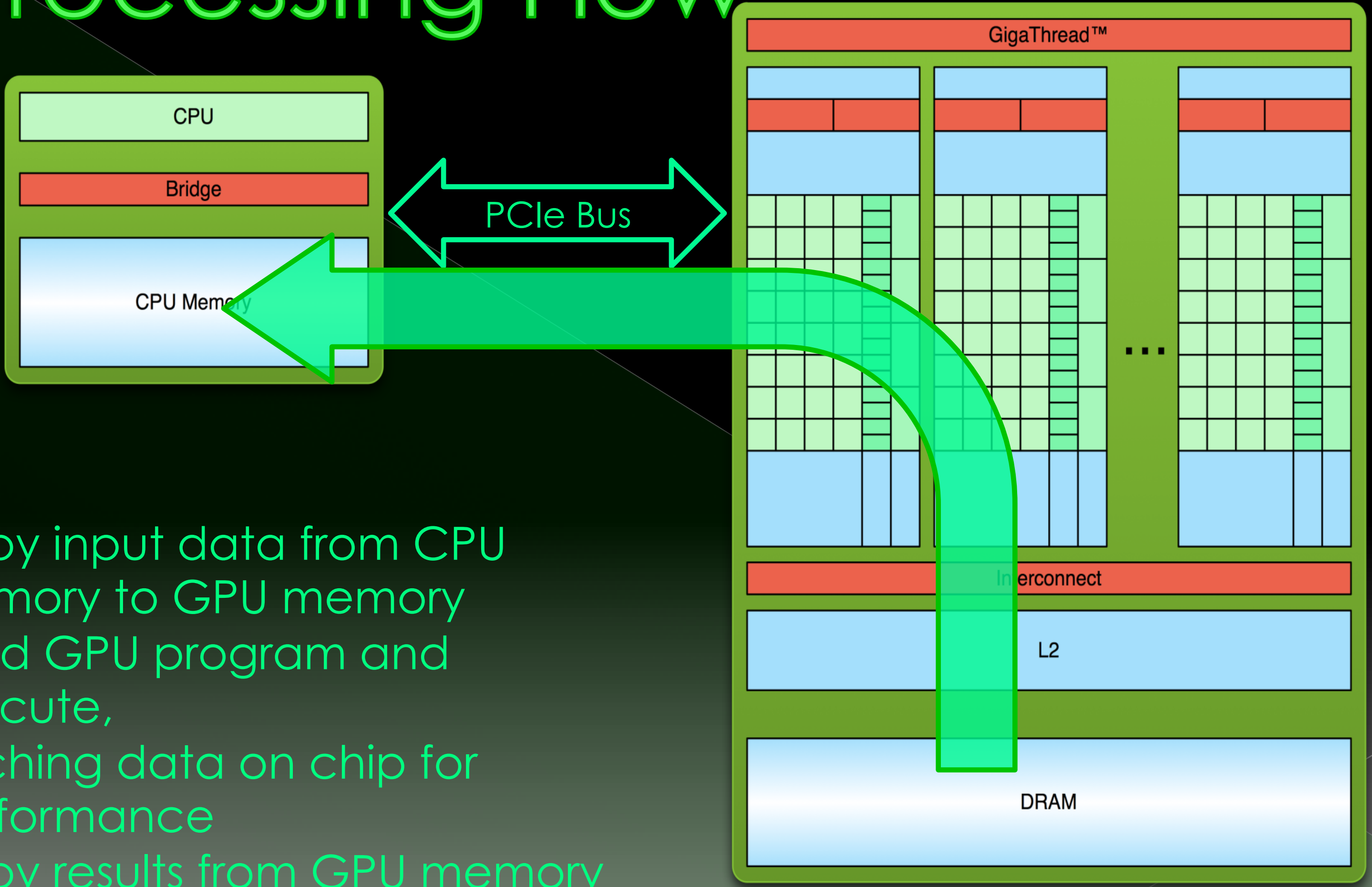
1. Copy input data from CPU memory to GPU memory

Processing Flow



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

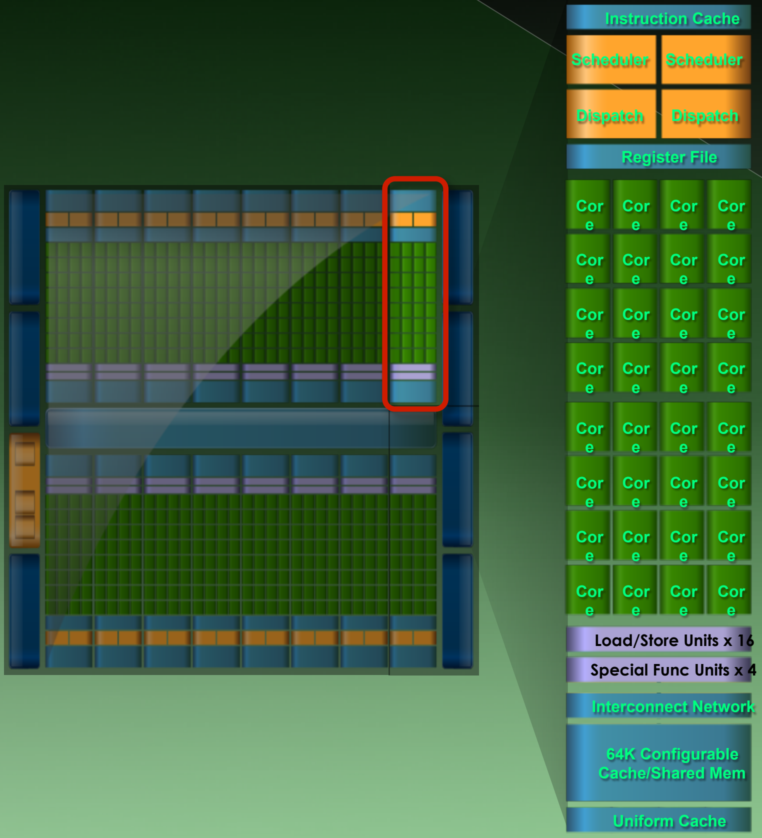
Processing Flow



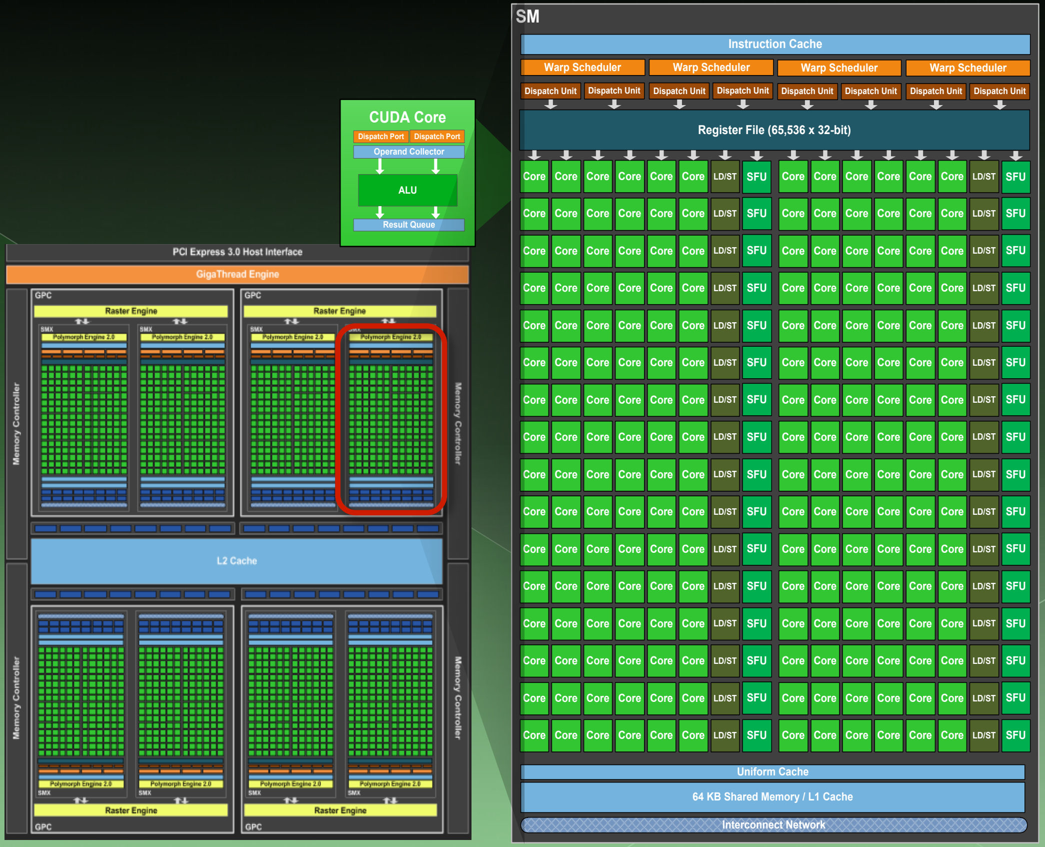
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute,
   caching data on chip for performance
3. Copy results from GPU memory to CPU memory

TESLA : SC3 Choice in GPUs

Fermi (GUANE-1)

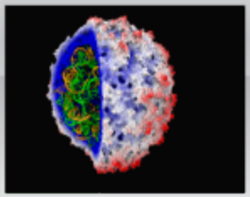


Kepler (GUANE-2)



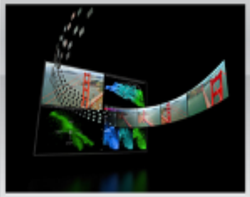
146X

Medical Imaging U of Utah



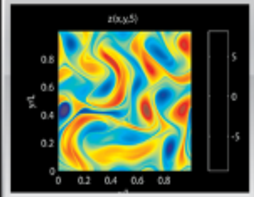
36X

Molecular Dynamics U of Illinois, Urbana



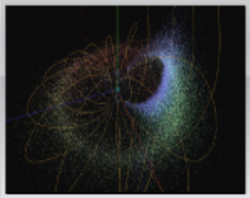
18X

Video Transcoding Elemental Tech



50X

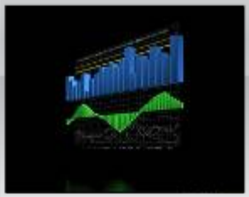
Matlab Computing AccelerEyes



100X

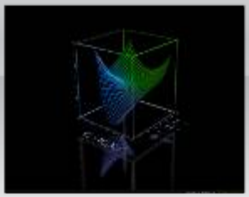
Astrophysics RIKEN

GPUs Accelerate Science



149X

Financial Simulation Oxford



47X

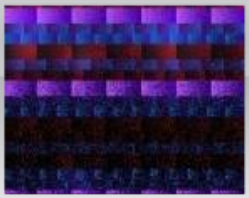
Linear Algebra Universidad Jaime

20X

3D Ultrasound Techniscan

130X

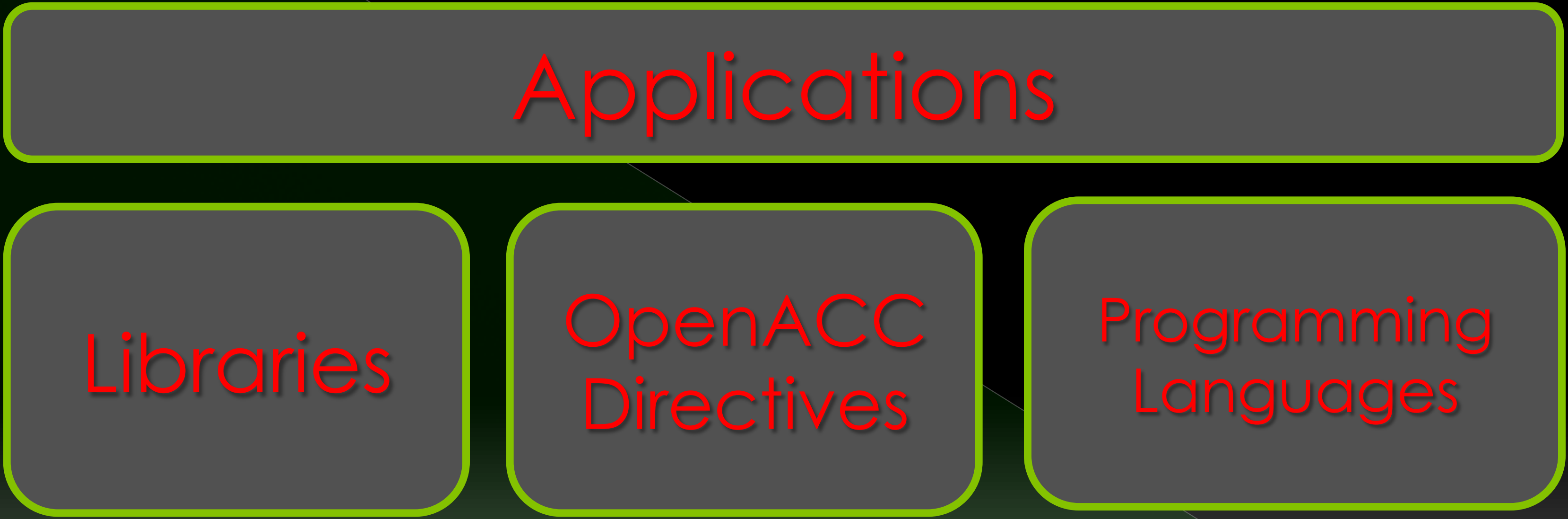
Quantum Chemistry U of Illinois, Urbana



30X

Gene Sequencing U of Maryland

3 Ways to Accelerate Applications



"Drop-in" Acceleration

Easily Accelerate Applications

Maximum Performance

3 Steps to CUDA-accelerated application

- **Step 1:** Substitute library calls with equivalent CUDA library calls

  `saxpy ( … )` ➤ `cublasSaxpy ( … )`

- **Step 2:** Manage data locality

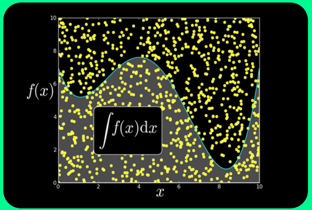
  - with CUDA: `cudaMalloc(), cudaMemcpy(), etc.`
  - with CUBLAS: `cublasAlloc(), cublasSetVector(), etc.`

- **Step 3:** Rebuild and link the CUDA-accelerated library

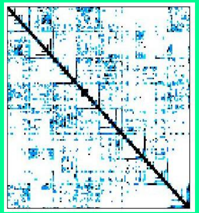
```
nvcc myobj.o –l cublas
```

Some GPU-accelerated Libraries

NVIDIA cuBLAS



NVIDIA cuRAND



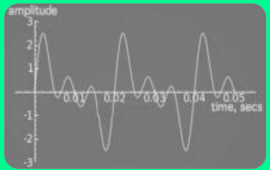
NVIDIA cuSPARSE

NVIDIA NPP

Vector Signal Image Processing

GPU Accelerated Linear Algebra

Matrix Algebra on GPU and Multicore



NVIDIA cuFFT

IMSL Library

ArrayFire Matrix Computations

CUDA

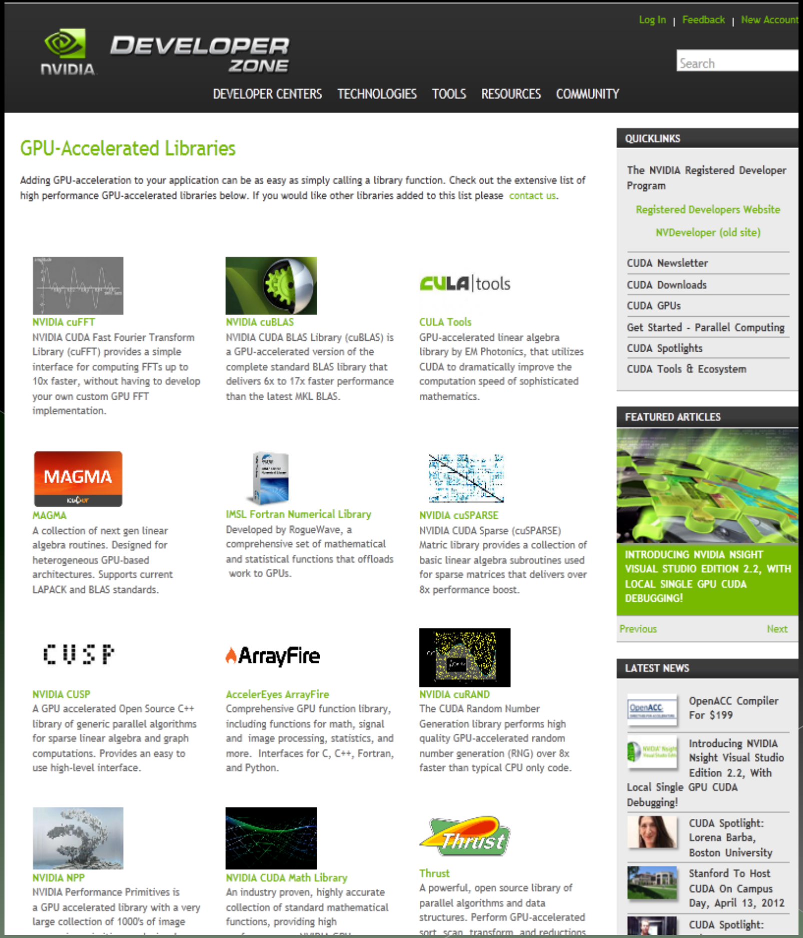
Sparse Linear Algebra

C++ STL Features for CUDA

Explore the CUDA (Libraries) Ecosystem

- CUDA Tools and Ecosystem described in detail on NVIDIA Developer Zone:

  developer.nvidia.com/cuda-tools-ecosystem


Log In | Feedback | New Account
NVIDIA
DEVELOPER ZONE
Search
DEVELOPER CENTERS TECHNOLOGIES TOOLS RESOURCES COMMUNITY

GPU-Accelerated Libraries

Adding GPU-acceleration to your application can be as easy as simply calling a library function. Check out the extensive list of high performance GPU-accelerated libraries below. If you would like other libraries added to this list please contact us.

NVIDIA cuFFT
NVIDIA CUDA Fast Fourier Transform Library (cuFFT) provides a simple interface for computing FFTs up to 10x faster, without having to develop your own custom GPU FFT implementation.

NVIDIA cuBLAS
NVIDIA CUDA BLAS Library (cuBLAS) is a GPU-accelerated version of the complete standard BLAS library that delivers 6x to 17x faster performance than the latest MKL BLAS.

CULA Tools
GPU-accelerated linear algebra library by EM Photonics, that utilizes CUDA to dramatically improve the computation speed of sophisticated mathematics.

MAGMA
A collection of next gen linear algebra routines. Designed for heterogeneous GPU-based architectures. Supports current LAPACK and BLAS standards.

IMSL Fortran Numerical Library
Developed by RogueWave, a comprehensive set of mathematical and statistical functions that offloads work to GPUs.

NVIDIA cuSPARSE
NVIDIA CUDA Sparse (cuSPARSE) Matrix library provides a collection of basic linear algebra subroutines used for sparse matrices that delivers over 8x performance boost.

NVIDIA CUSP
A GPU accelerated Open Source C++ library of generic parallel algorithms for sparse linear algebra and graph computations. Provides an easy to use high-level interface.

AccelerEyes ArrayFire
Comprehensive GPU function library, including functions for math, signal and image processing, statistics, and more. Interfaces for C, C++, Fortran, and Python.

NVIDIA cuRAND
The CUDA Random Number Generation library performs high quality GPU-accelerated random number generation (RNG) over 8x faster than typical CPU only code.

NVIDIA NPP
NVIDIA Performance Primitives is a GPU accelerated library with a very large collection of 1000's of image

NVIDIA CUDA Math Library
An industry proven, highly accurate collection of standard mathematical functions, providing high

Thrust
A powerful, open source library of parallel algorithms and data structures. Perform GPU-accelerated

QUICKLINKS

- The NVIDIA Registered Developer Program
- Registered Developers Website
- NVDeveloper (old site)
- CUDA Newsletter
- CUDA Downloads
- CUDA GPUs
- Get Started - Parallel Computing
- CUDA Spotlights
- CUDA Tools & Ecosystem

FEATURED ARTICLES

INTRODUCING NVIDIA NSIGHT VISUAL STUDIO EDITION 2.2, WITH LOCAL SINGLE GPU CUDA DEBUGGING!

Previous Next

LATEST NEWS

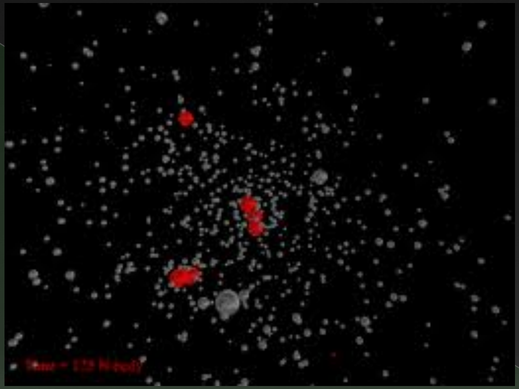
- OpenACC Compiler For $199
- Introducing NVIDIA Nsight Visual Studio Edition 2.2, With Local Single GPU CUDA Debugging!
- CUDA Spotlight: Lorena Barba, Boston University
- Stanford To Host CUDA On Campus Day, April 13, 2012
- CUDA Spotlight:

3 in House Examples

Microscopy Using EDF

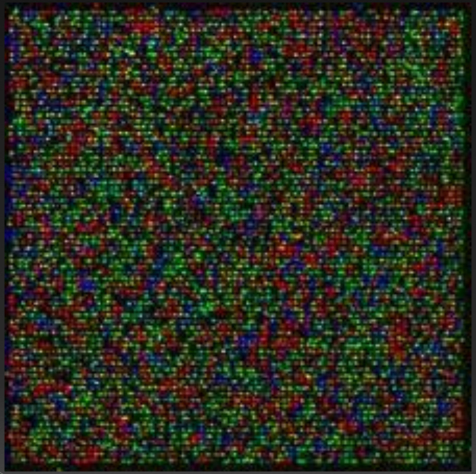
5x in 5 Hours

N-Bodies Based in Montecarlo



12x in 1 Hour

Iinfluenza MetaGenomics



125x in 2 Hours

3 Technical Developments

- CUDA 6.0 Performance Evaluation
  - Memory Allocation
- NVIDIA TESLA K40 Performance Evaluation
  - Application Assembly and Portability
- NVIDIA TESLAK20x + OMMPs
  - Application Assembly and Dynamic Job Scheduling

3 Application Examples

- GROMACS +FlowVR
  - Visualisation Assembly addressed to Multi-GPU Execution (In Consortium with INRIA Rhône Alpes, France)
- Seismic (Inverse and Elastic) Methods for Oil and Gas Prospective
  - GPU and Multi-GPU Codes (In Consortium with ICP-Ecopetrol and Barcelona Supercomputing Center(BSC_CNS))
- VisioPlatform for Astronomy and Astrophysics
  - Based in Stallion (In Consortium with Texas Advanced Computing Center, USA)

New Challenges

- Gas Pipelines Simulation
- Water Simulations
- Falls of Cosmic Ray
- Seismic Solvers to Oil and Gas Needs (in Consortium with BSC)

More Programming Languages

Python — PyCUDA

C# .NET — GPU.NET tidepowerd

Numerical Analytics — MATLAB *The Language of Technical Computing* Wolfram *Mathematica* 8

Get Started Today

These languages are supported on all CUDA-capable GPUs.

You might already have a CUDA-capable GPU in your laptop or desktop PC!

CUDA C/C++
http://developer.nvidia.com/cuda-toolkit

Thrust C++ Template Library
http://developer.nvidia.com/thrust

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

PyCUDA (Python)
http://mathema.tician.de/software/pycuda

GPU.NET
http://tidepowerd.com

MATLAB
http://www.mathworks.com/discovery/matlab-gpu.html

Mathematica
http://www.wolfram.com/mathematica/new-in-8/cuda-and-opencl-support/

CUDA C

Standard 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);
```

Parallel C

```
__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

4

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

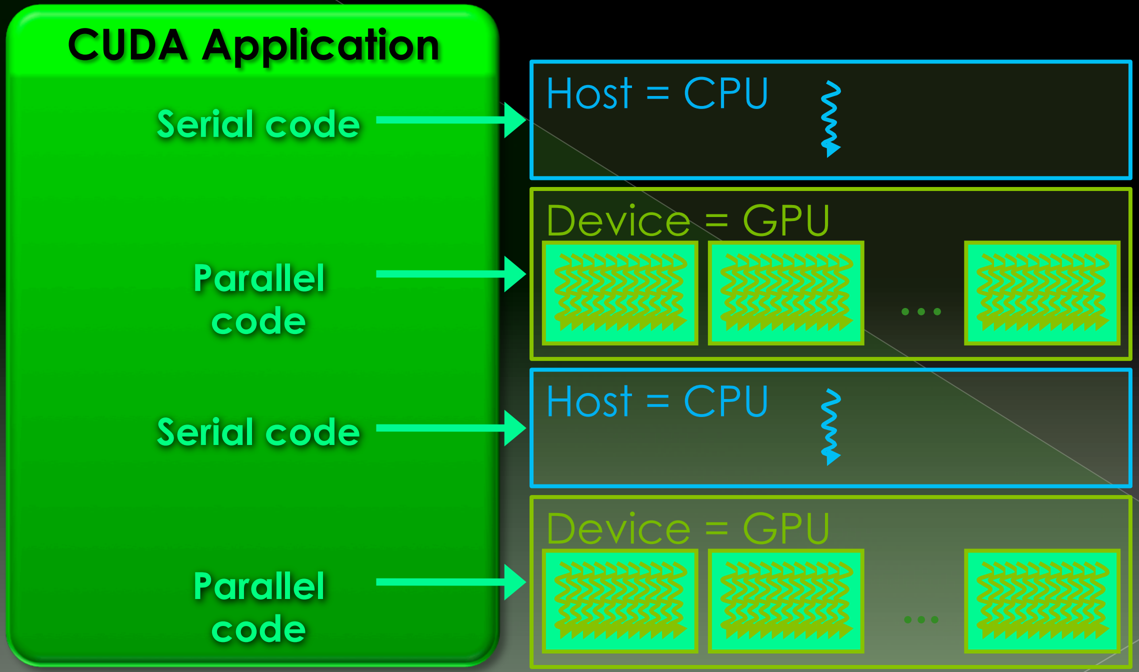
Copperhead: Parallel Python

```
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



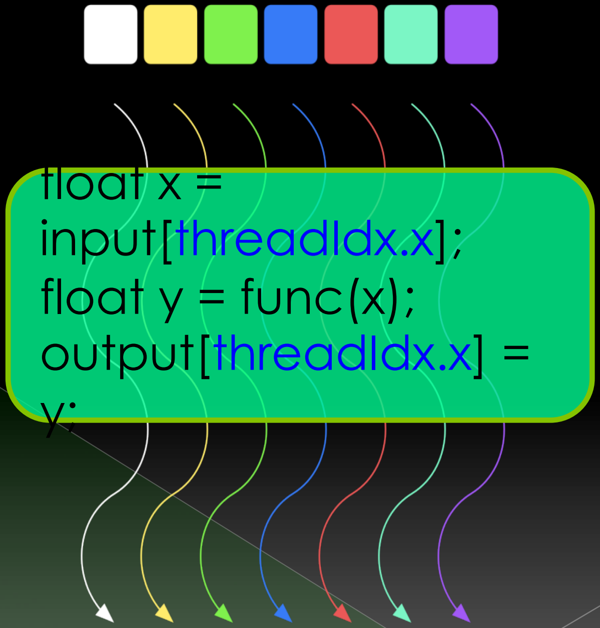
CUDA Kernels

- 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

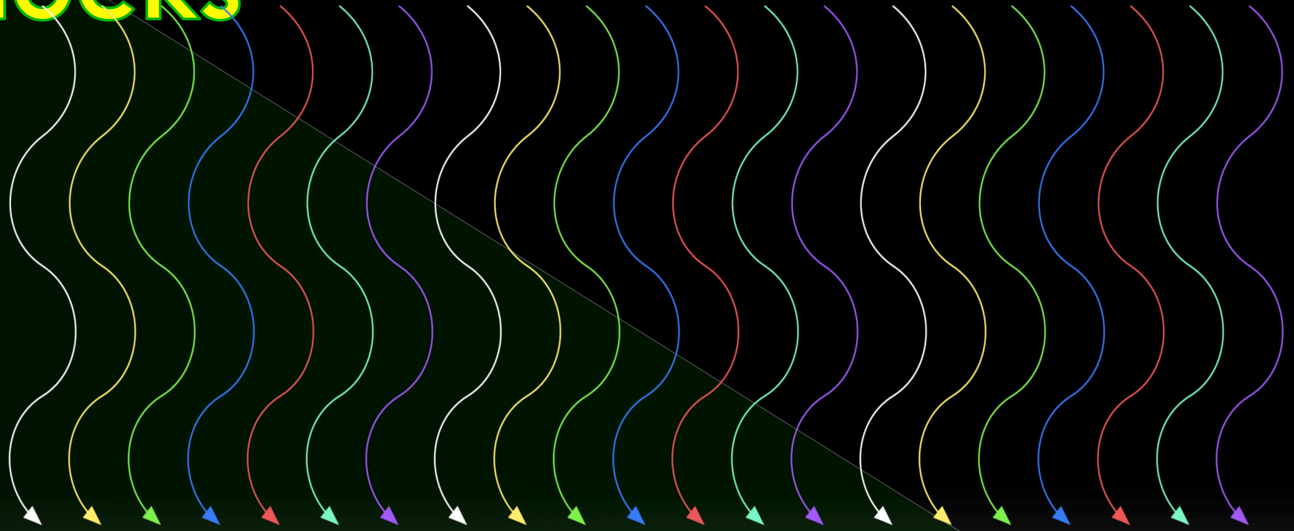
CUDA Kernels: Parallel Threads

- 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

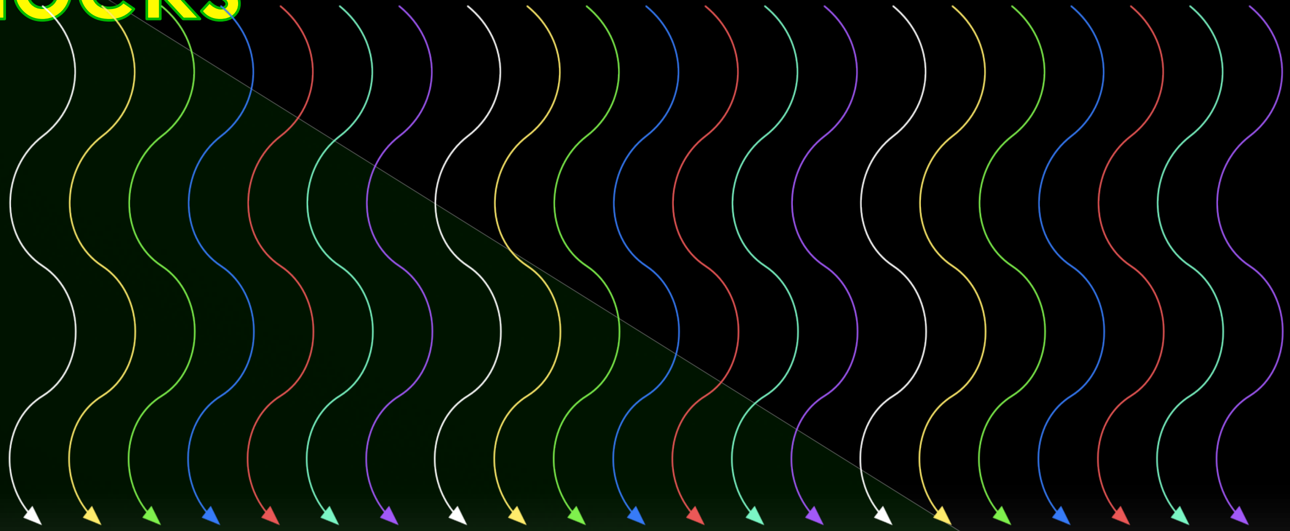


```
float x = input[threadIdx.x];
float y = func(x);
output[threadIdx.x] = y;
```

CUDA Kernels: Subdivide into Blocks

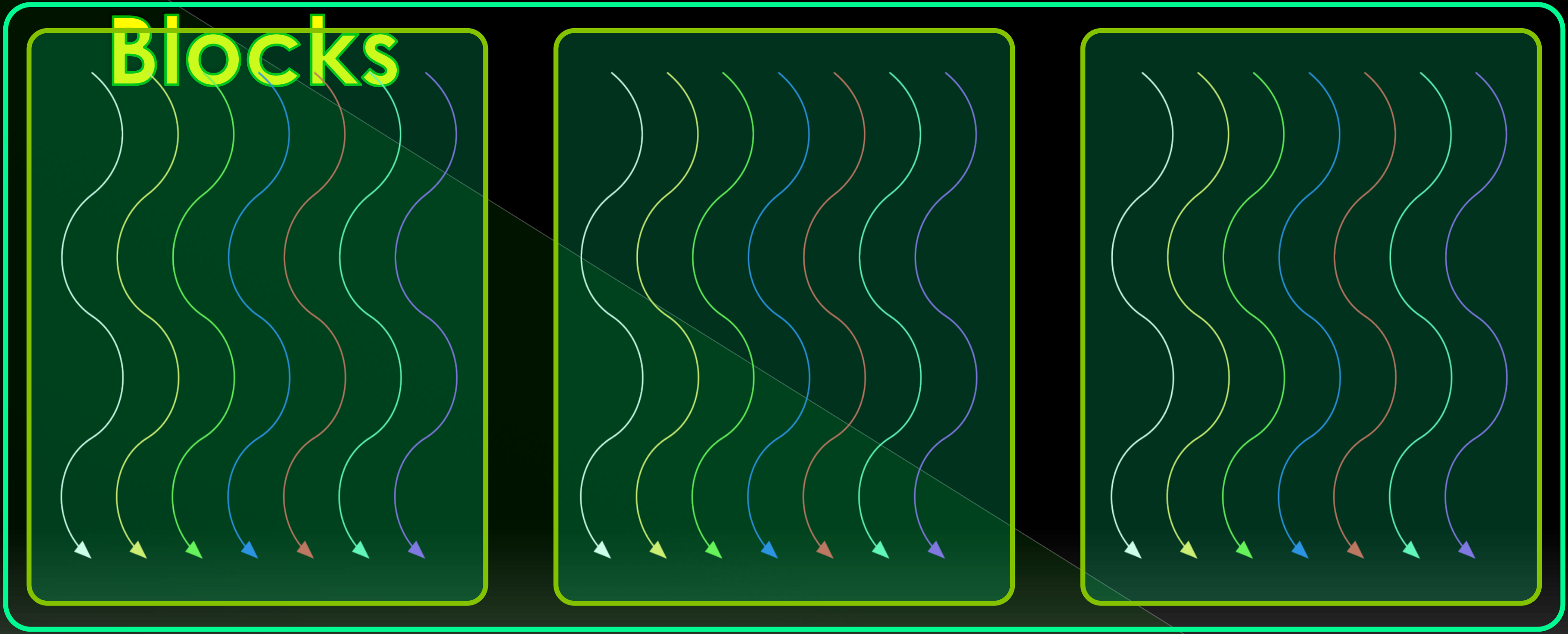


CUDA Kernels: Subdivide into Blocks



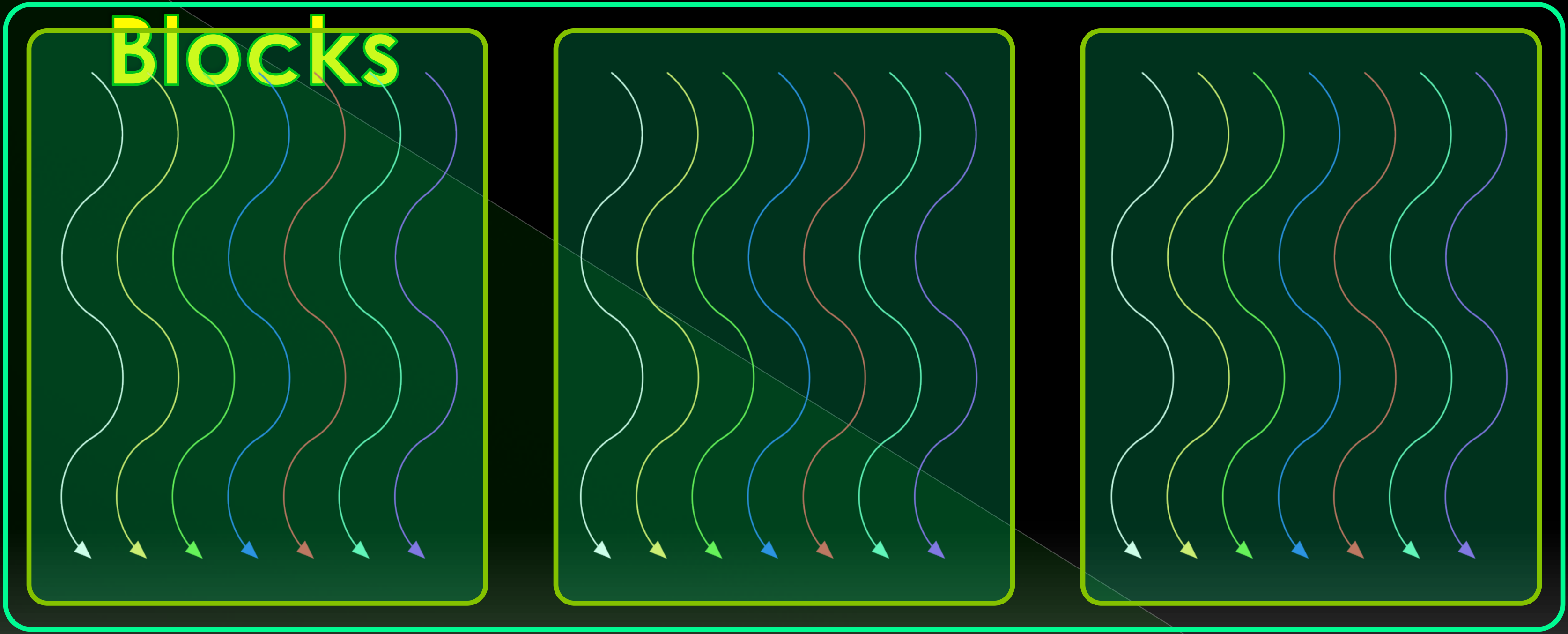
- Threads are grouped into blocks

CUDA Kernels: Subdivide into Blocks



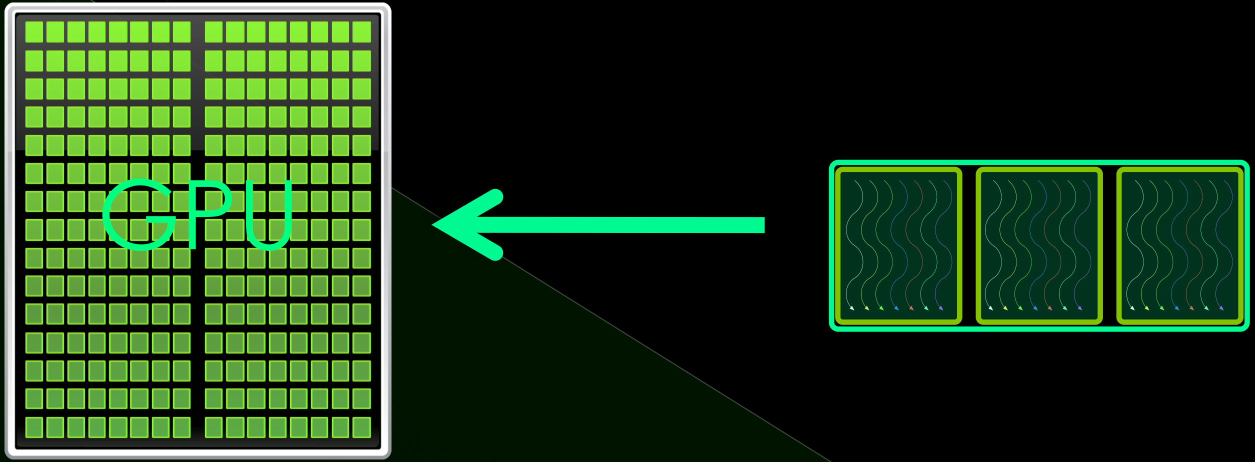
- Threads are grouped into blocks
- Blocks are grouped into a grid

CUDA Kernels: Subdivide into Blocks



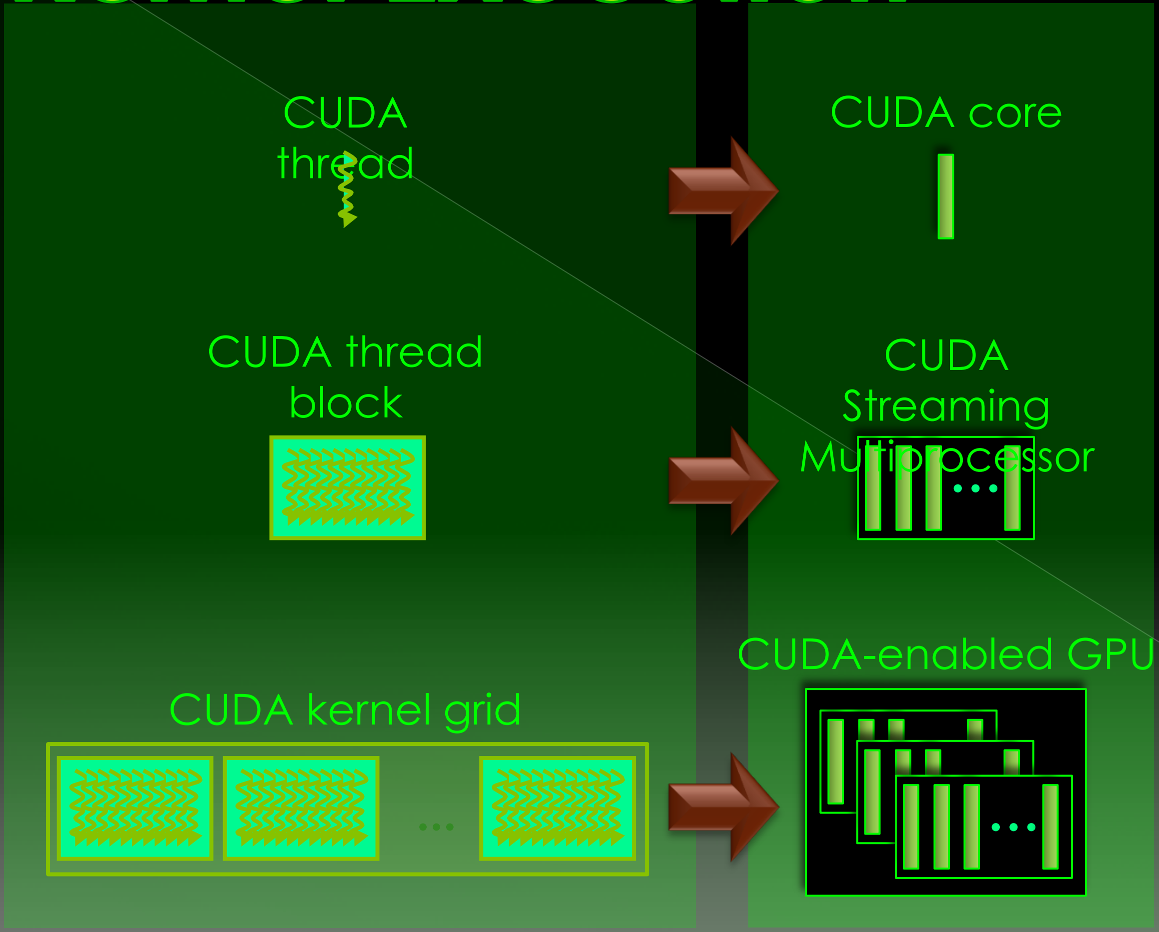
- Threads are grouped into blocks
- Blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads

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



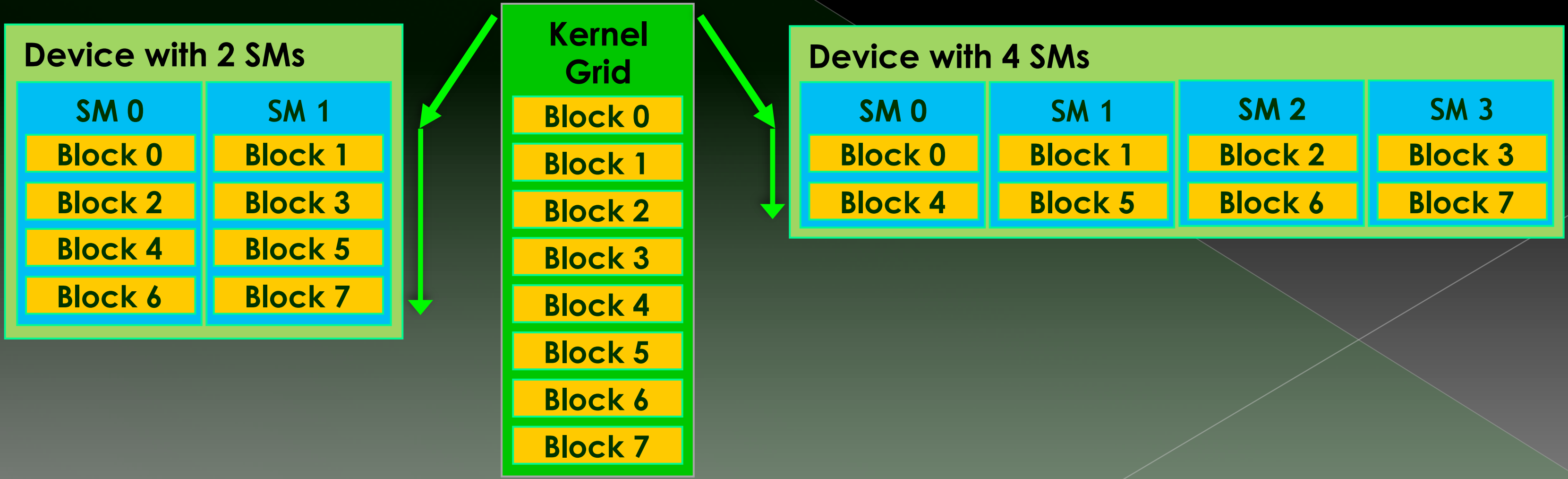
- Each thread is executed by a core
- Each block is executed by one SM and does not migrate
- Several concurrent blocks can reside on one SM depending on the blocks' memory requirements and the SM's memory resources
- Each kernel is executed on one device
- Multiple kernels can execute on a device at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

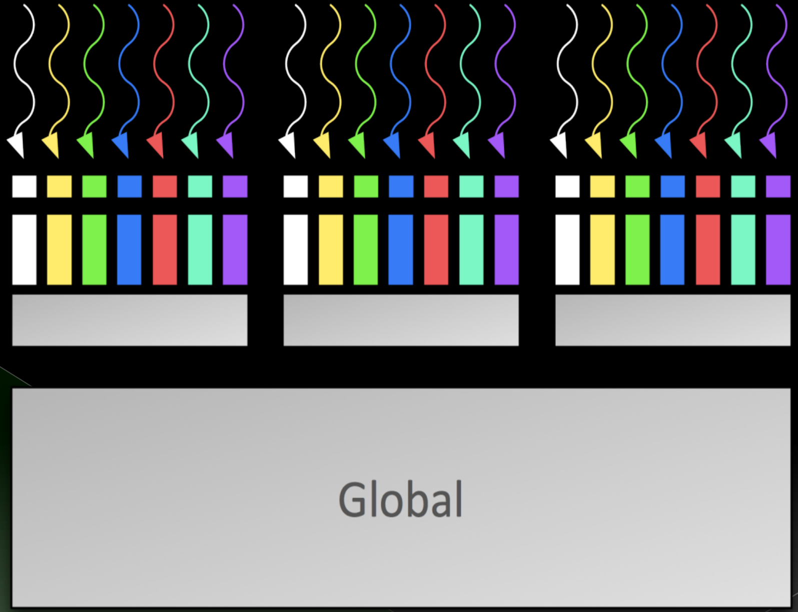
Thread blocks allow scalability

- Blocks can execute in any order, concurrently or sequentially
- This independence between blocks gives scalability:
  - A kernel scales across any number of SMs

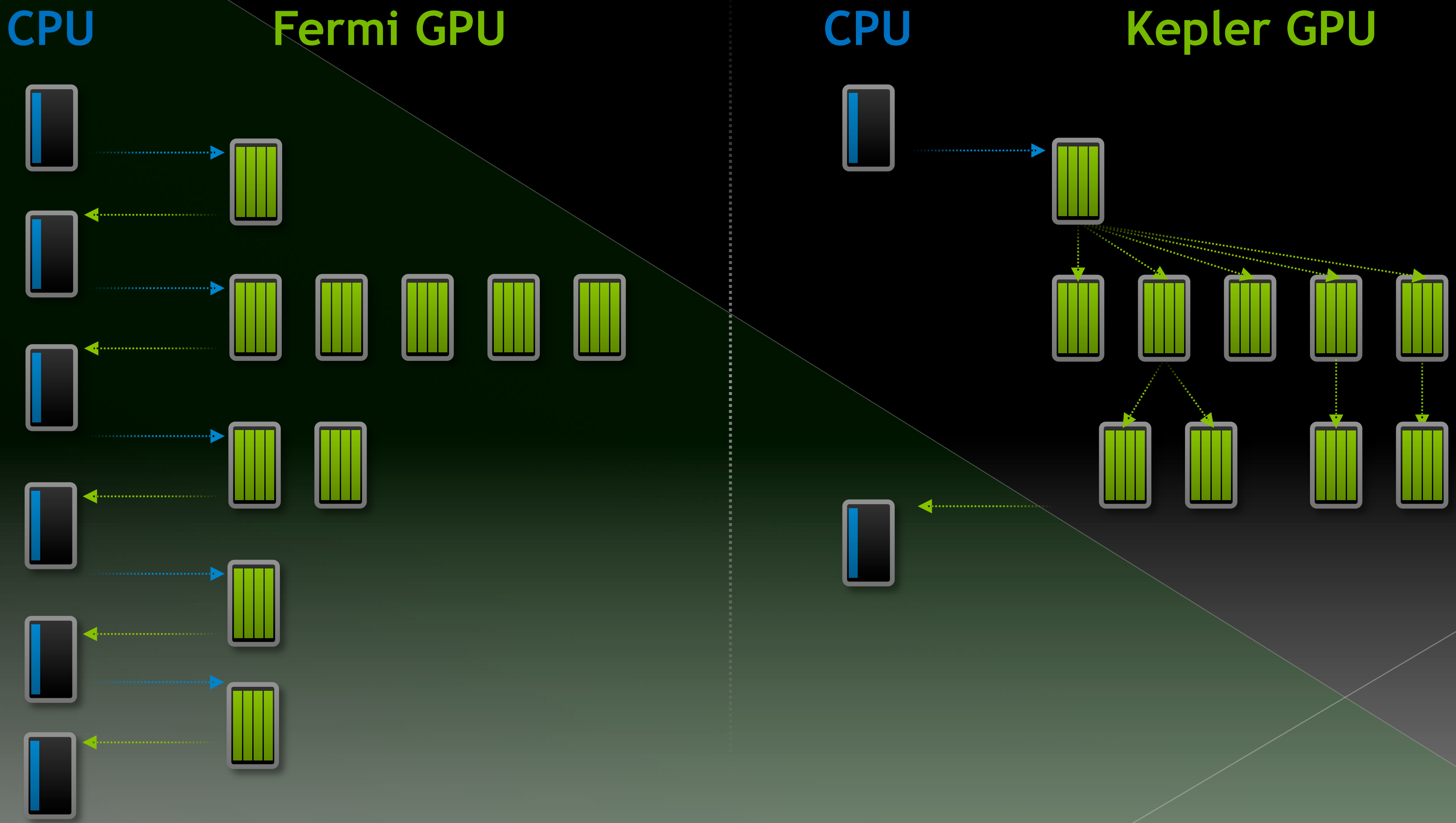


Memory hierarchy

- Thread:
  - Registers
  - Local memory
- Block of threads:
  - Shared memory
- All blocks:
  - Global memory

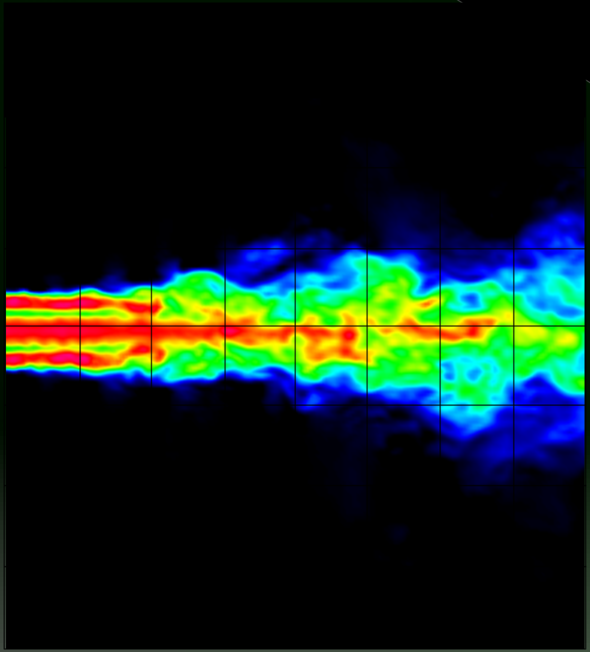


Dynamic Parallelism



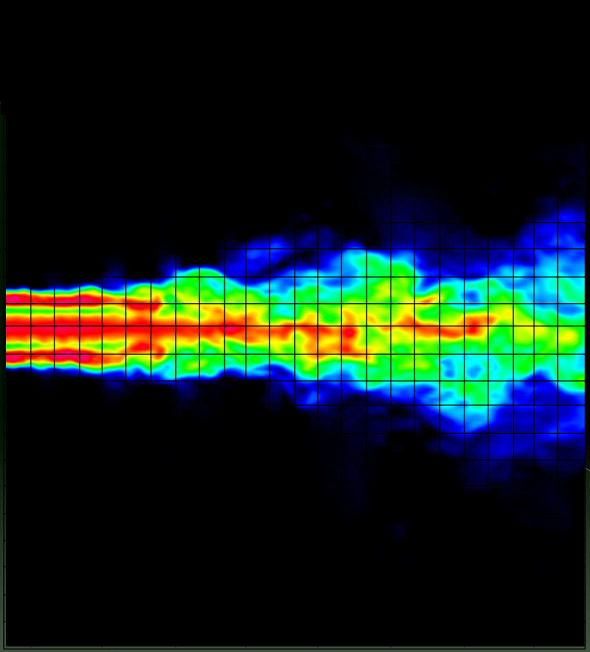
Dynamic Work Generation

Coarse grid



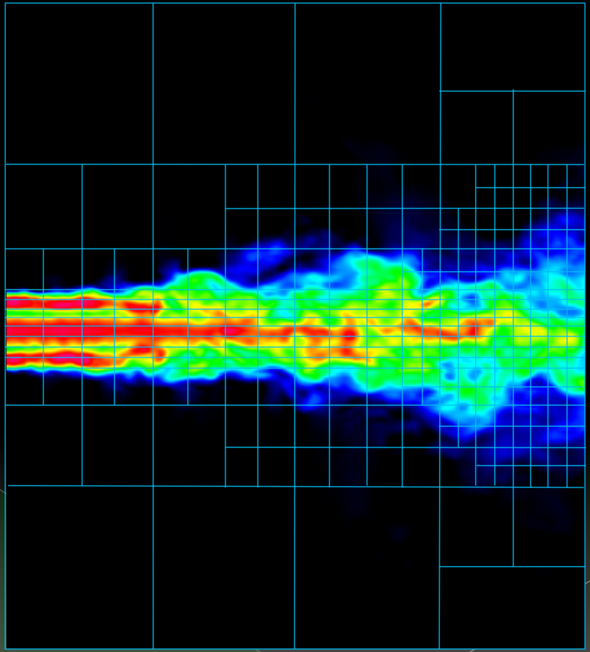
Higher Performance
Lower Accuracy

Fine grid



Lower Performance
Higher Accuracy

Dynamic grid



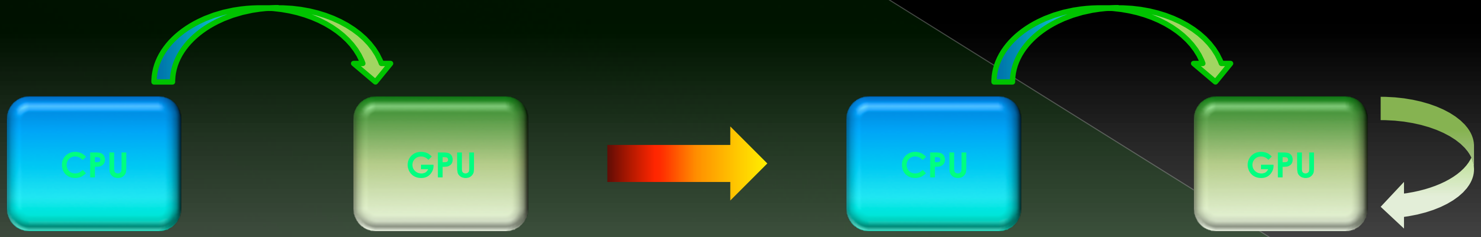
Target performance where accuracy is required

Supported on GK110 GPUs

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



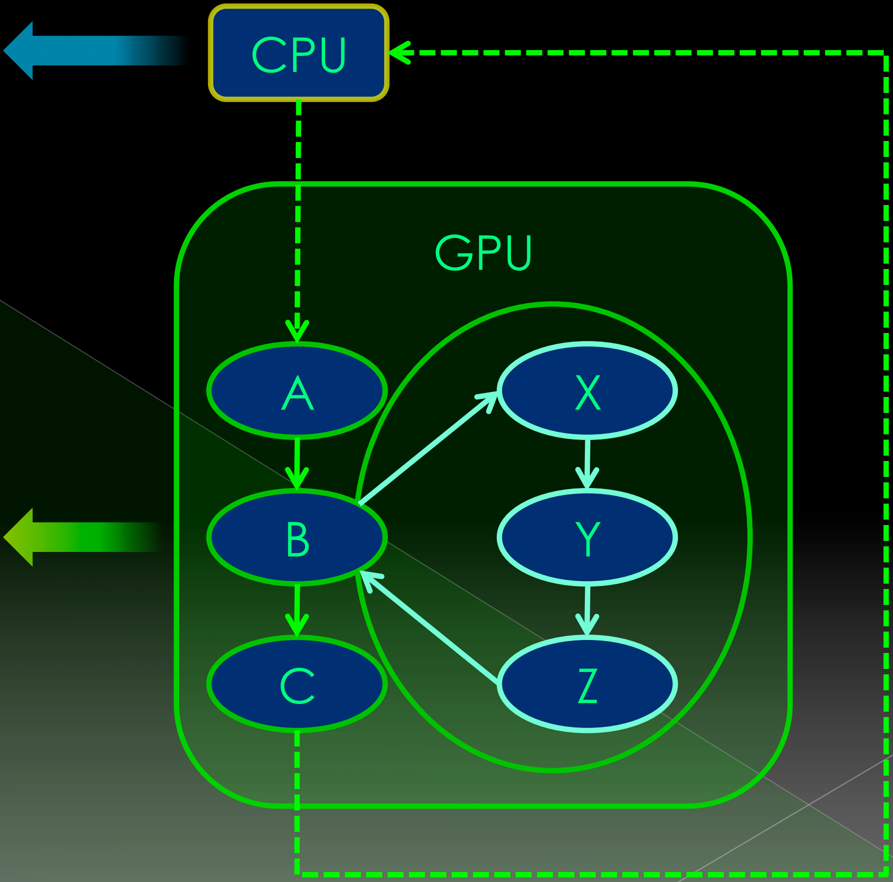
Fermi: Only CPU can generate GPU work

Kepler: GPU can generate work for itself

Familiar Programming Model

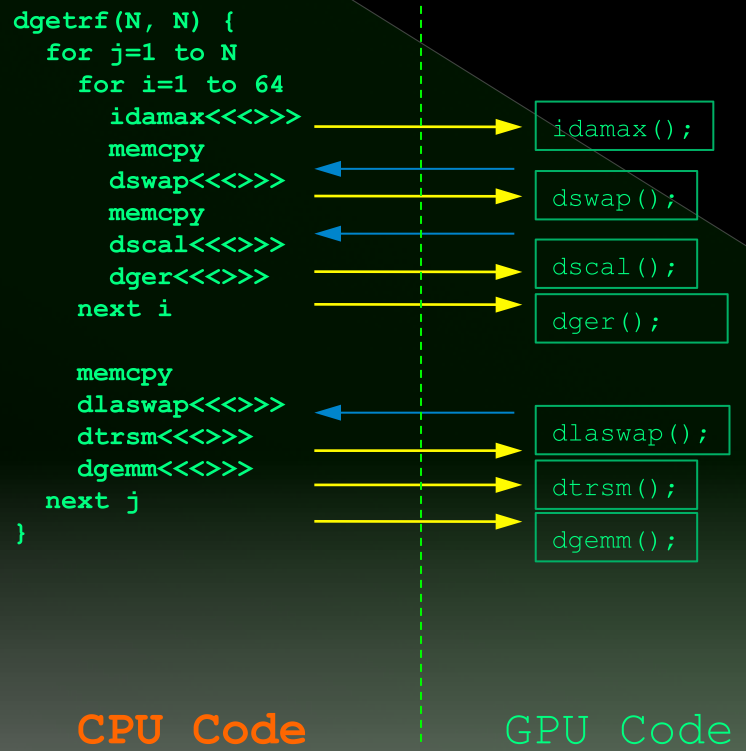
```
int main() {
    float *data;
    setup(data);

    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);
}
```



Simpler Code: LU Example

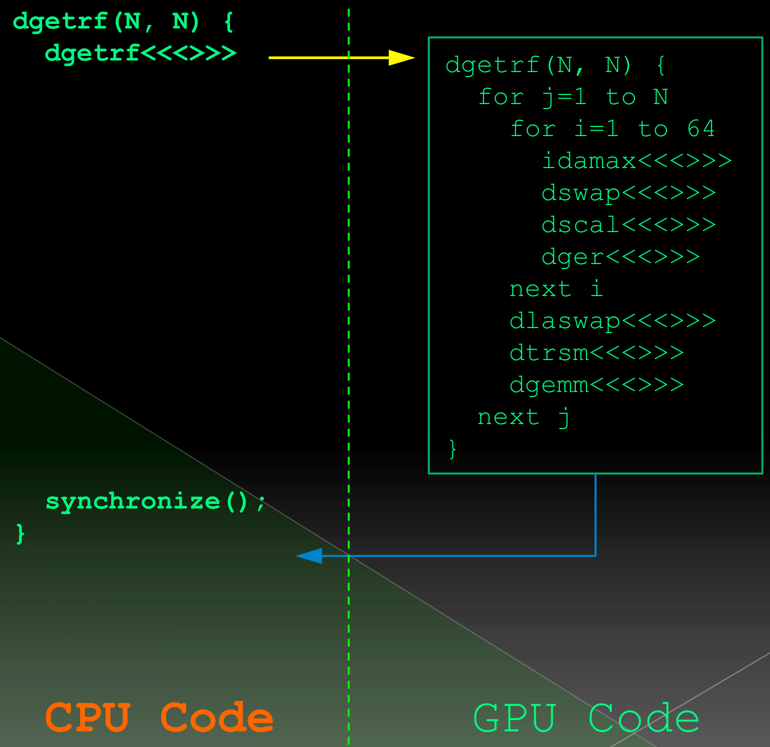
LU decomposition (Fermi)



```
dgetrf(N, N) {
  for j=1 to N
    for i=1 to 64
      idamax<<<>>>
      memcpy
      dswap<<<>>>
      memcpy
      dscal<<<>>>
      dger<<<>>>
    next i

    memcpy
    dlaswap<<<>>>
    dtrsm<<<>>>
    dgemm<<<>>>
  next j
}
```

```
idamax();
dswap();
dscal();
dger();
dlaswap();
dtrsm();
dgemm();
```

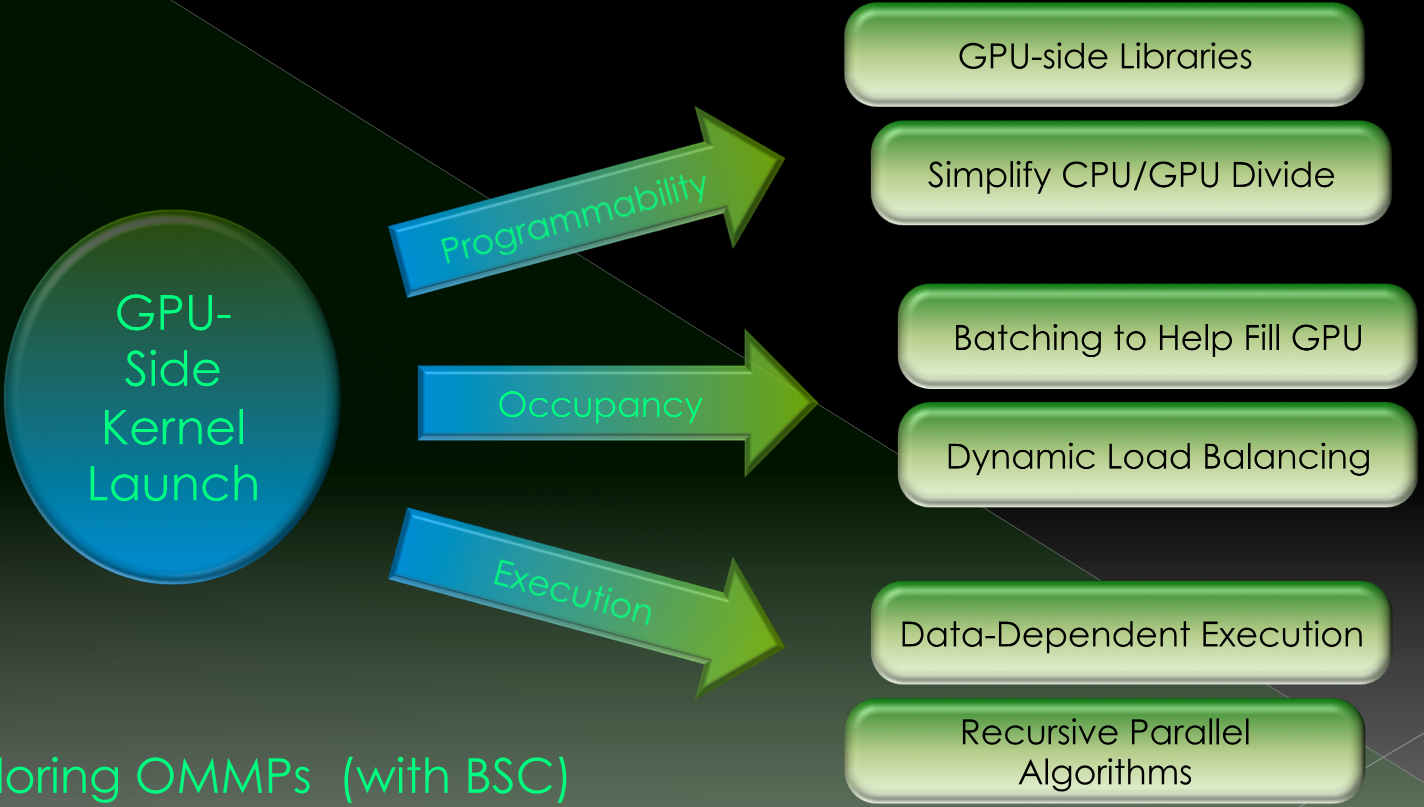
LU decomposition (Kepler)



```
dgetrf(N, N) {
  dgetrf<<<>>>



  synchronize();
}
```

```
dgetrf(N, N) {
  for j=1 to N
    for i=1 to 64
      idamax<<<>>>
      dswap<<<>>>
      dscal<<<>>>
      dger<<<>>>
    next i
    dlaswap<<<>>>
    dtrsm<<<>>>
    dgemm<<<>>>
  next j
}
```

Program Model Interests



- Exploring OMMPs (with BSC)
- OpenACC (With Cray, NVIDIA)
- FlowVR (With INRIA Rhône Alpes)

Interesting Web References

- NVIDIA DEVELOPMENT SITE
  - http://developer.nvidia.com/page/home.html
- NVIDIA CUDA ZONE
  - http://www.nvidia.com/object/cuda_home_new.html

Compiling a CUDA™ code

- Using nvcc™ compilator
  - Visit this site and run the examples:
    - http://developer.nvidia.com/object/cuda_3_1_downloads.html

- Typical compiling

nvcc mycudacode.cu

- Specific compilation

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

Final Notes

- CUDA is good
  - Parallel Massive Programs
  - Low Bandwidth and Fine granularity Programs
  - Scale Programs
- Efficient Load Balancing
  - Multi-GPU Processing
  - Exploit Massive Concurrent Features

Thanks - Questions?

http://sc3.uis.edu.co