



# Introduction to GPU Computing

Mike Clark, NVIDIA  
Developer Technology Group

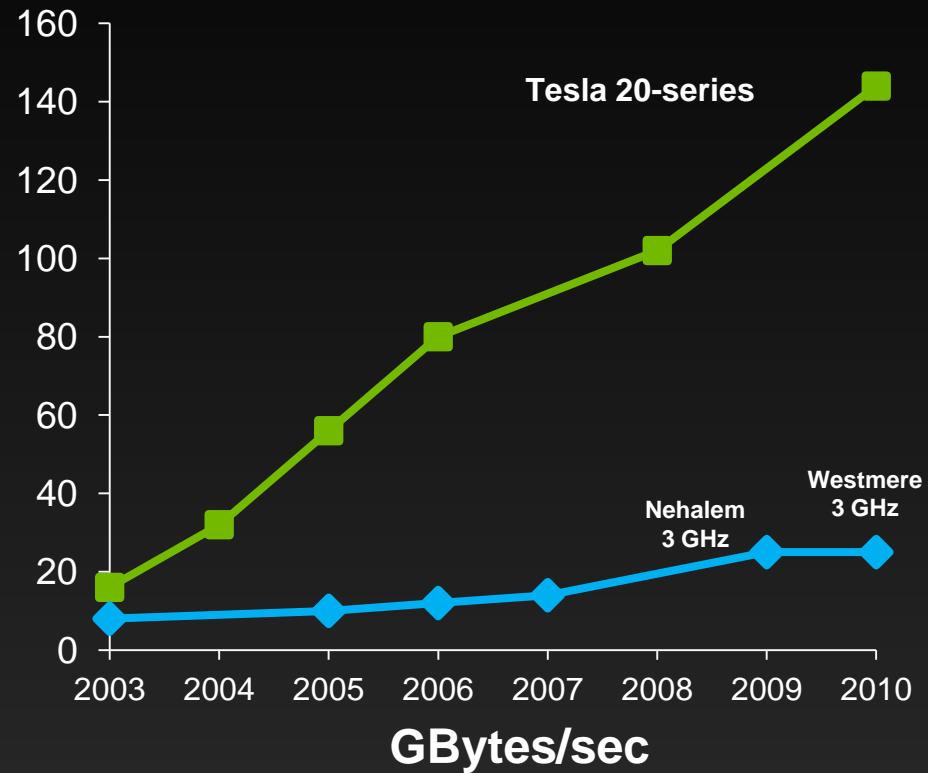
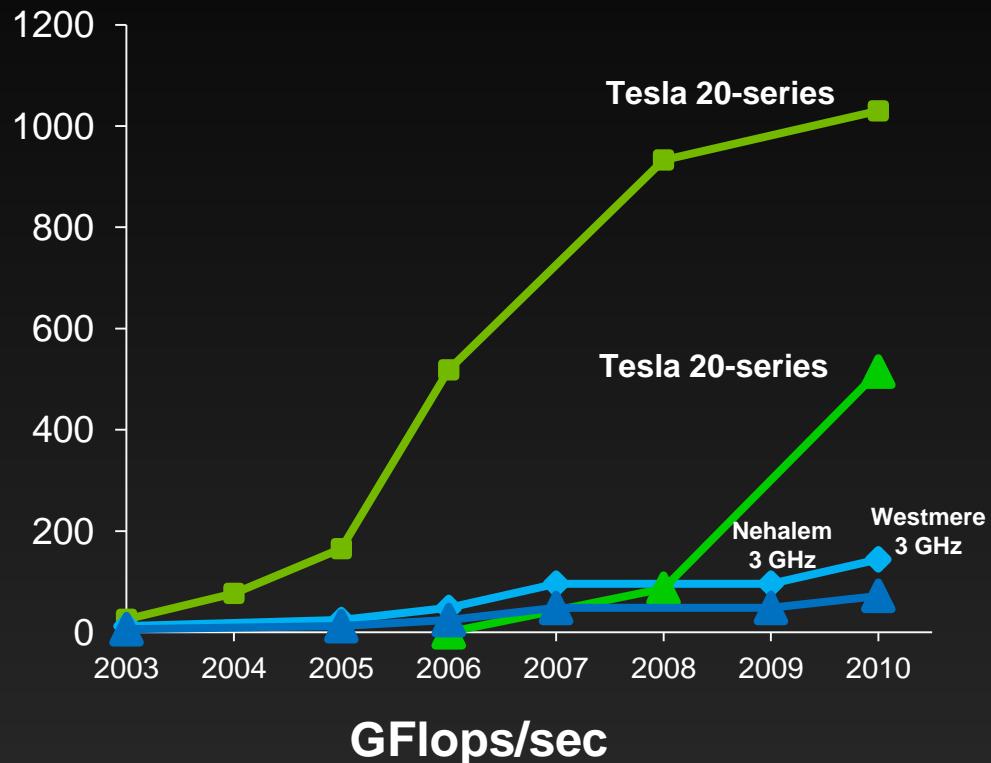


# Outline



- Today
  - Motivation
  - GPU Architecture
  - Three ways to accelerate applications
- Tomorrow
  - QUDA: QCD on GPUs

# Why GPU Computing?



■ Single Precision: NVIDIA GPU  
▲ Double Precision: NVIDIA GPU

◆ Single Precision: x86 CPU  
◆ Double Precision: x86 CPU

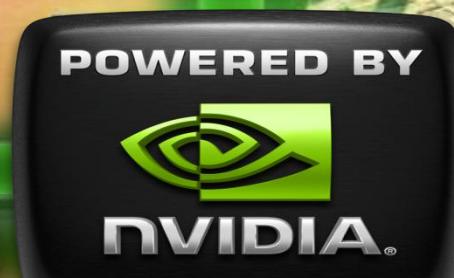
■ NVIDIA GPU    ◆ X86 CPU  
                  ECC off

Stunning Graphics Realism

Lush, Rich Worlds



Crysis © 2006 Crytek / Electronic Arts



Id software ©

Incredible Physics Effects

Core of the Definitive Gaming Platform

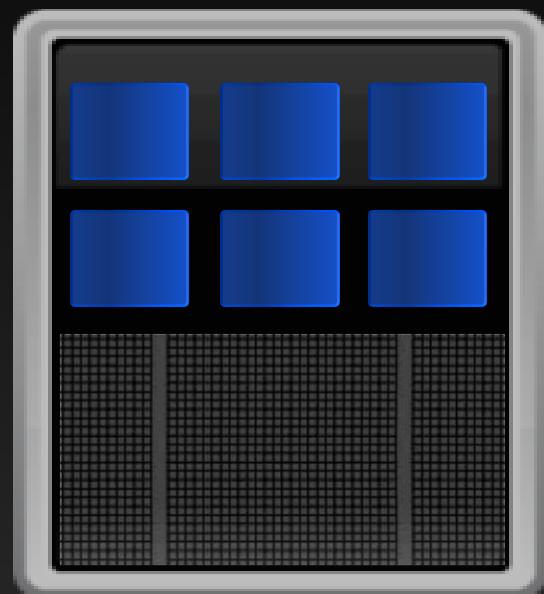


**MORE THAN JUST INNOVATIVE.  
GAME-CHANGING.**

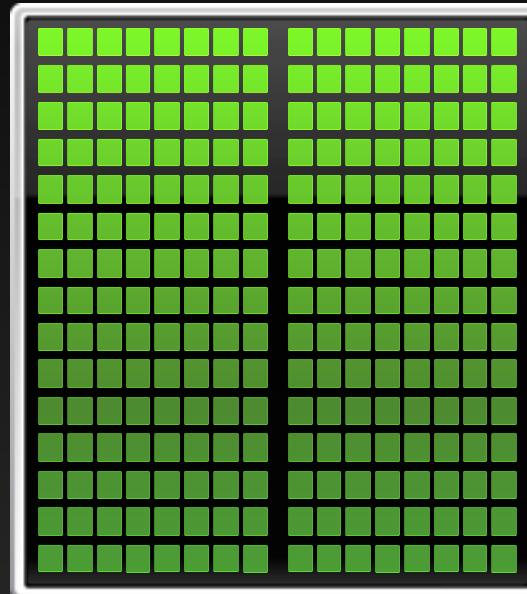
EXPERIENCE THE GEFORCE® GTX 690.

# Add GPUs: Accelerate Science Applications

CPU



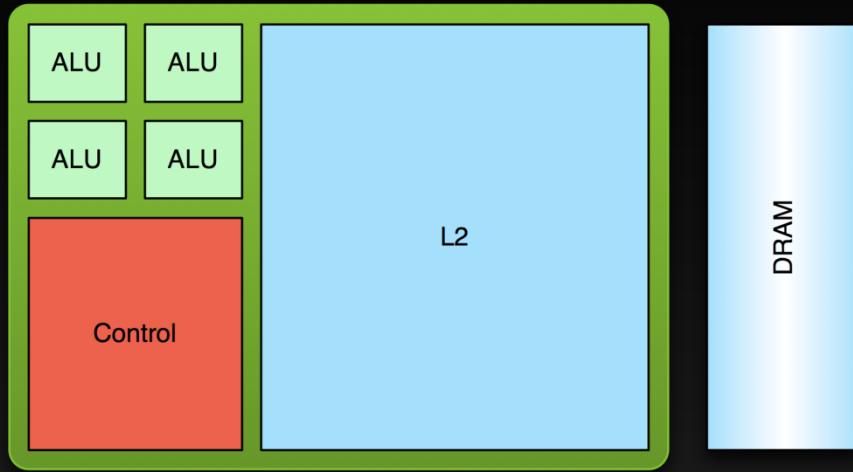
GPU



# Nbody GPU versus CPU

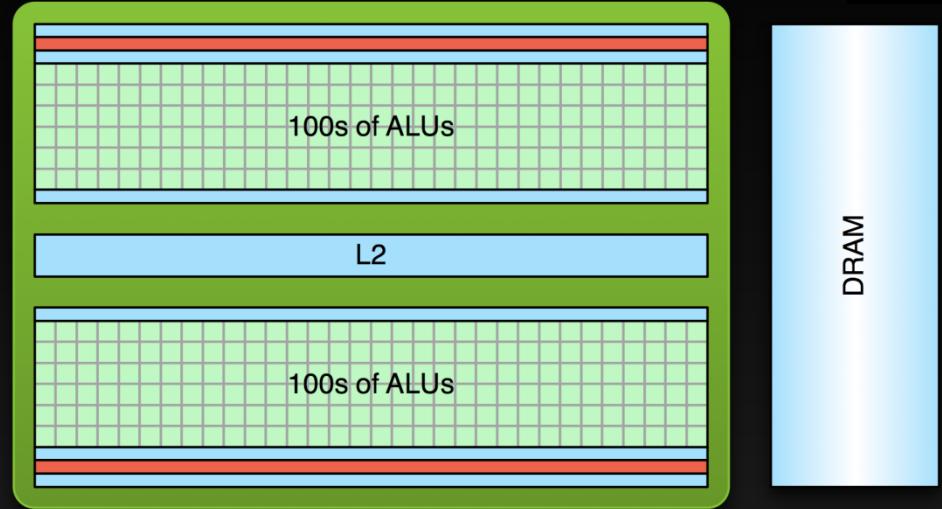


# Low Latency or High Throughput?



## CPU

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

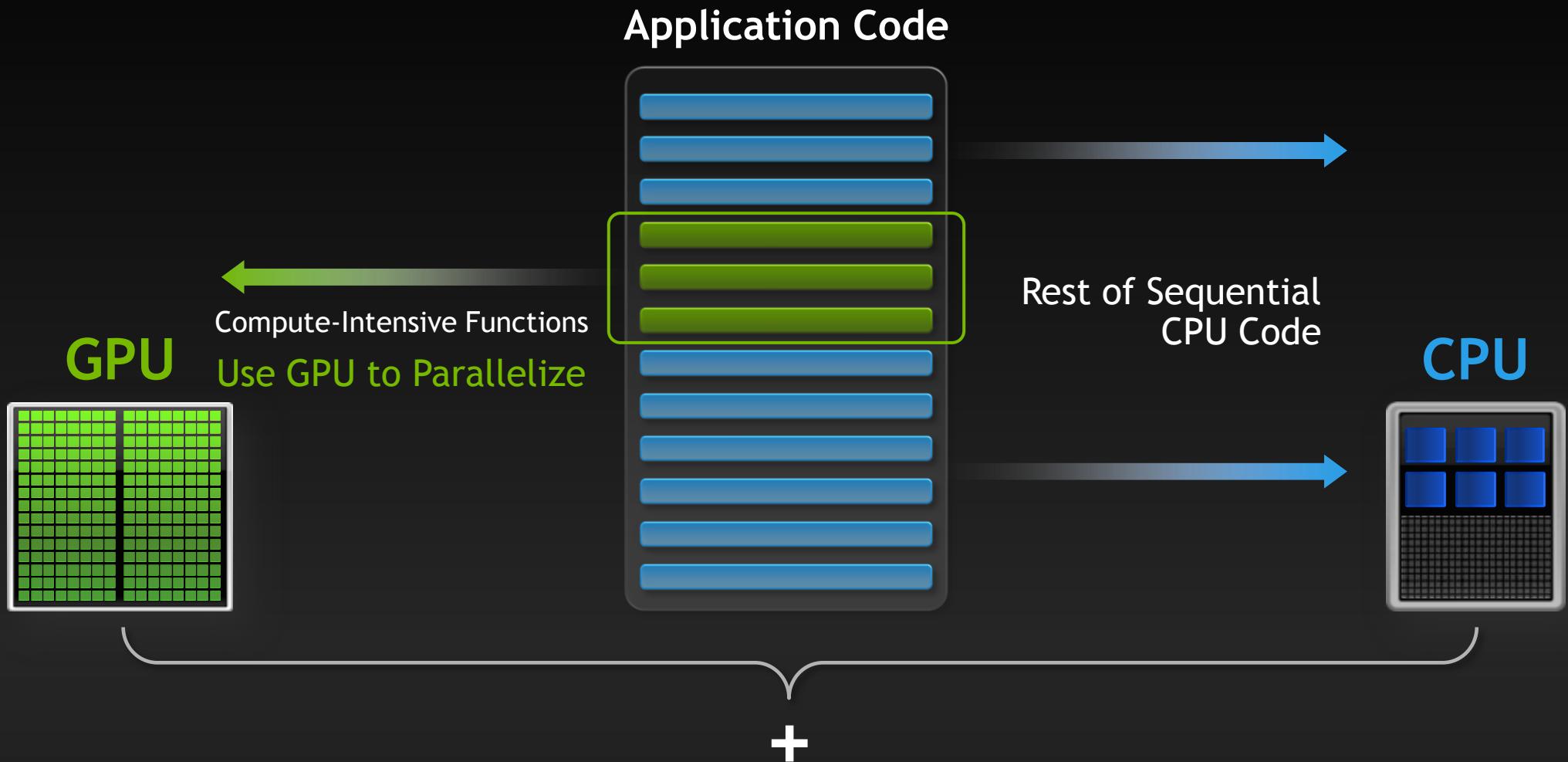


## GPU

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

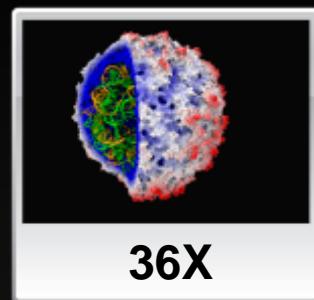


# Small Changes, Big Speed-up

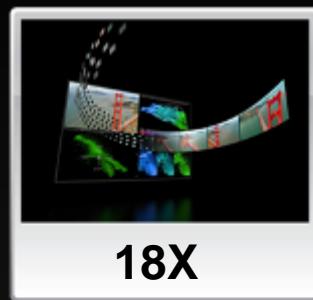




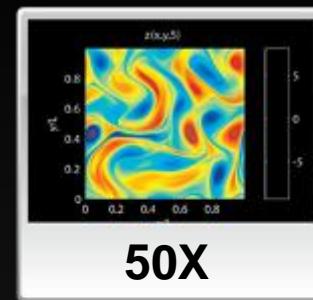
Medical Imaging  
U of Utah



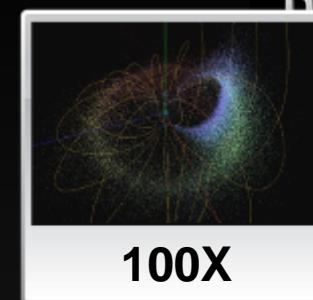
Molecular Dynamics  
U of Illinois, Urbana



Video Transcoding  
Elemental Tech

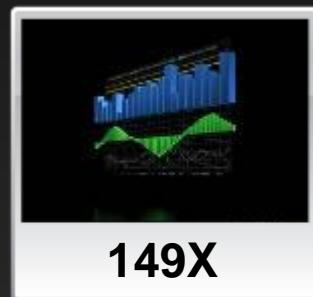


Matlab Computing  
AccelerEyes



Astrophysics  
RIKEN

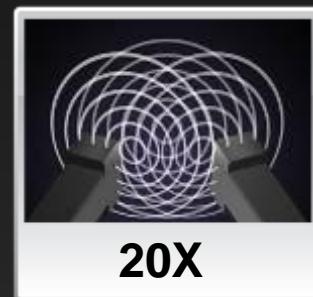
## GPUs Accelerate Science



Financial Simulation  
Oxford



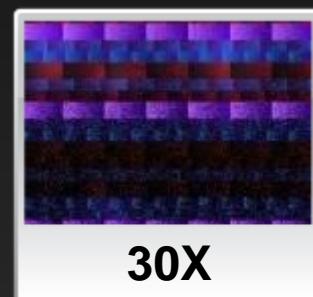
Linear Algebra  
Universidad Jaime



3D Ultrasound  
Technisan



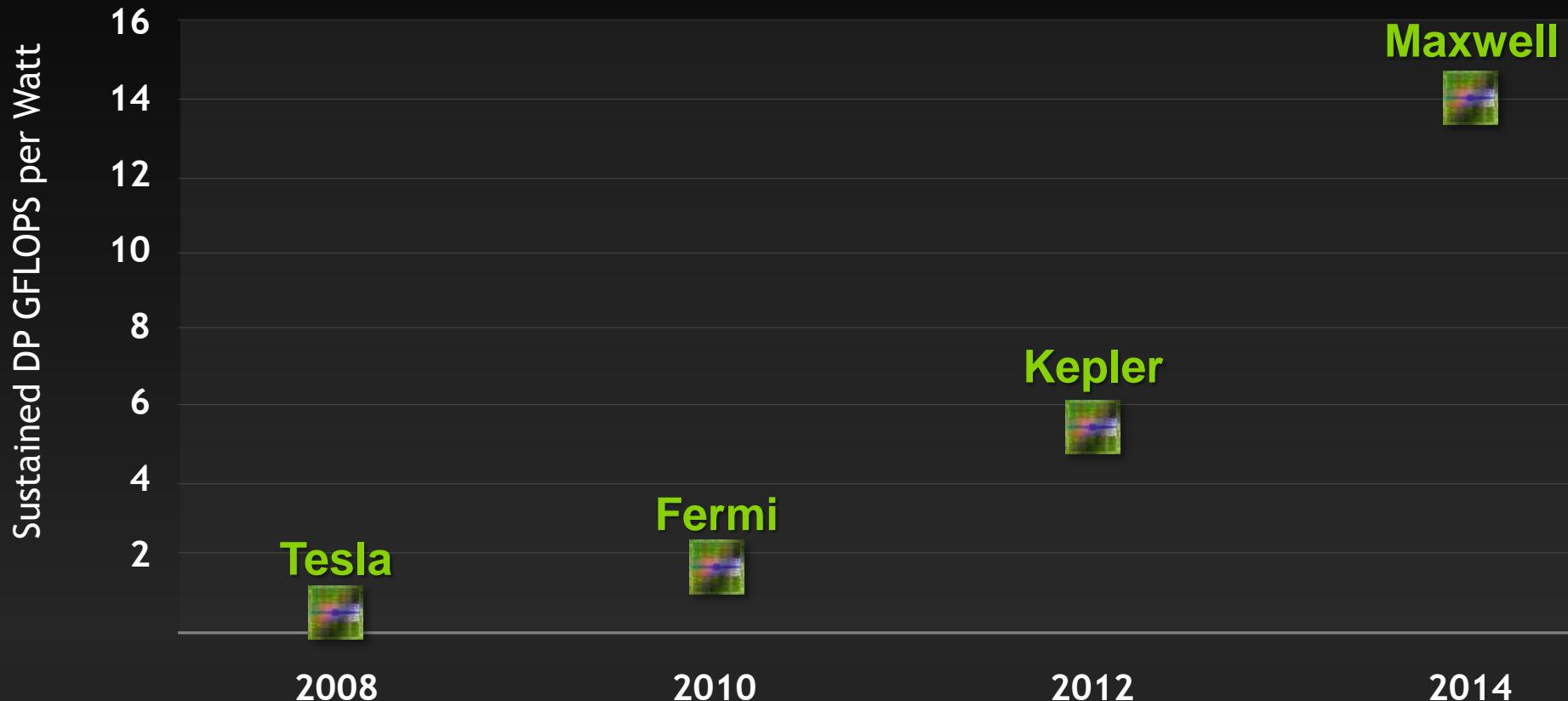
Quantum Chemistry  
U of Illinois, Urbana



Gene Sequencing  
U of Maryland



# NVIDIA GPU Roadmap: Increasing Performance/Watt





# GPU Architecture

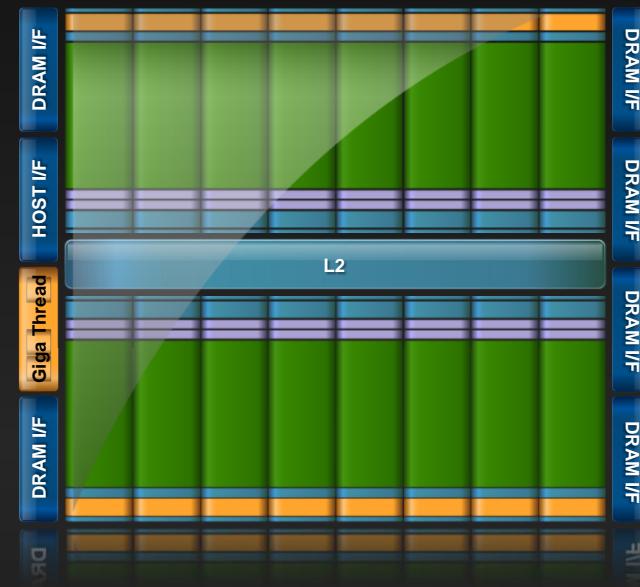
# GPU Architecture: Two Main Components

- **Global memory**

- Analogous to RAM in a CPU server
- Accessible by both GPU and CPU
- Currently up to **6 GB**
- Bandwidth currently up to **177 GB/s** for Quadro and Tesla products
- **ECC on/off** option for Quadro and Tesla products

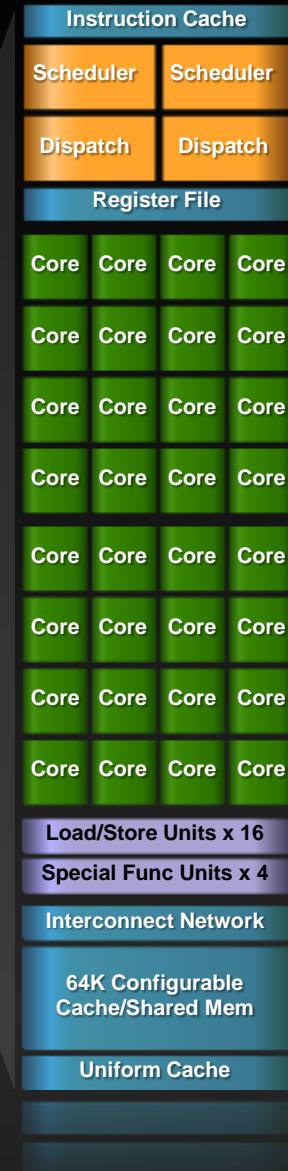
- **Streaming Multiprocessors (SMs)**

- Perform the actual computations
- Each SM has its own:
  - Control units, registers, execution pipelines, caches



# GPU Architecture - Fermi: Streaming Multiprocessor (SM)

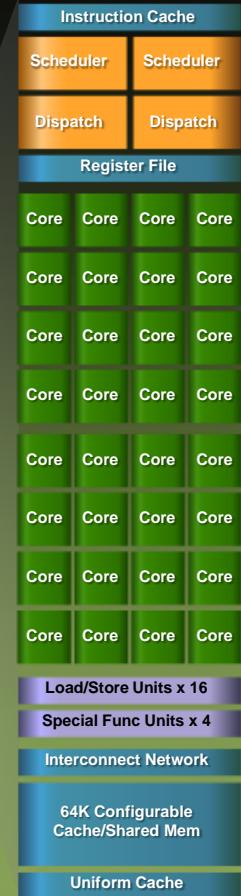
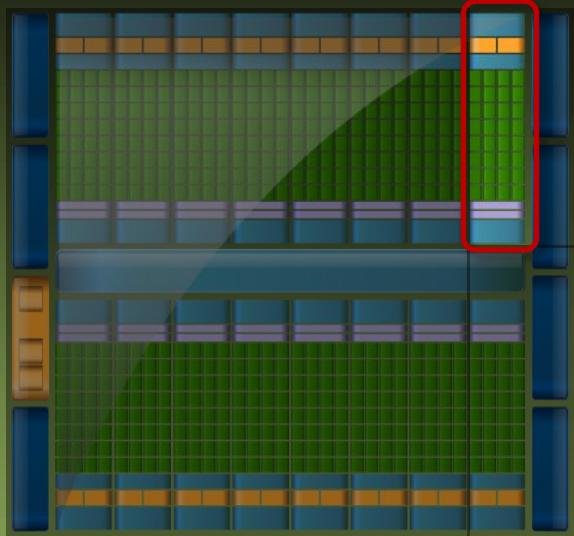
- 32 CUDA Cores per SM
  - 32 fp32 ops/clock
  - 16 fp64 ops/clock
  - 32 int32 ops/clock
- 2 warp schedulers
  - Up to 1536 threads concurrently
- 4 special-function units
- 64KB shared mem + L1 cache
- 32K 32-bit registers



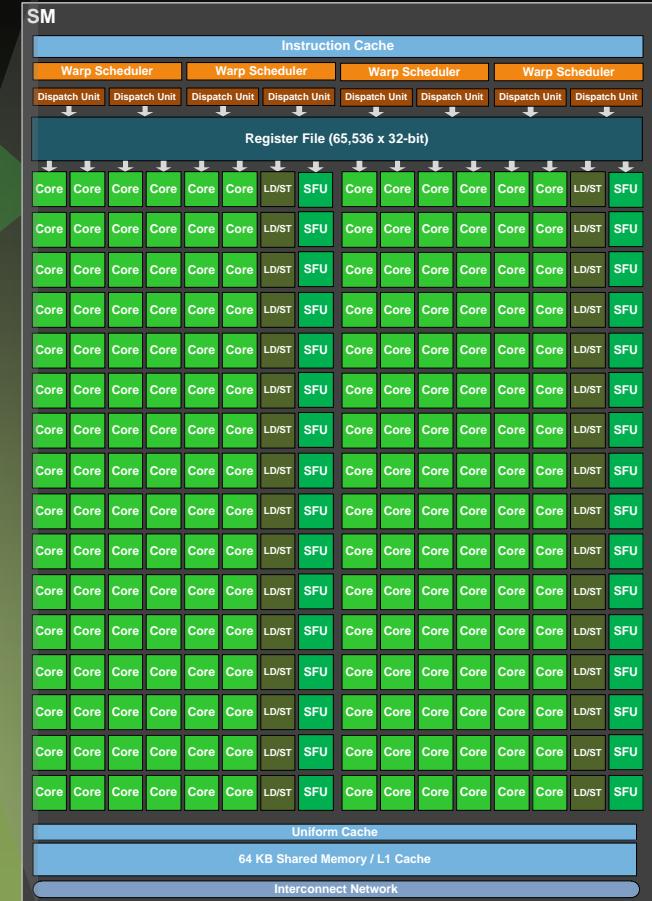
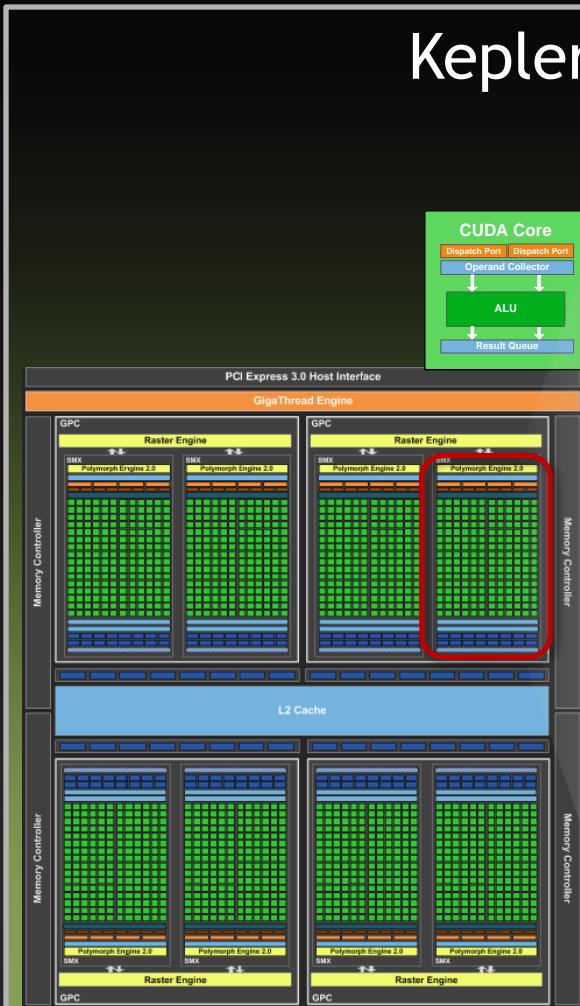
# Kepler



Fermi



# Kepler



# 3 Ways to Accelerate Applications



## Applications

Libraries

“Drop-in”  
Acceleration

OpenACC  
Directives

Easily Accelerate  
Applications

Programming  
Languages

Maximum  
Flexibility

# Libraries: Easy, High-Quality Acceleration

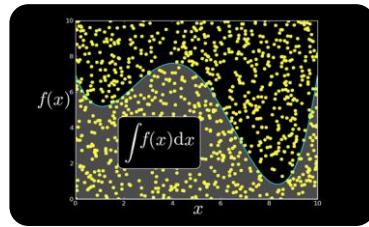


- **Ease of use:** Using libraries enables GPU acceleration without in-depth knowledge of GPU programming
- **“Drop-in”:** Many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes
- **Quality:** Libraries offer high-quality implementations of functions encountered in a broad range of applications
- **Performance:** NVIDIA libraries are tuned by experts

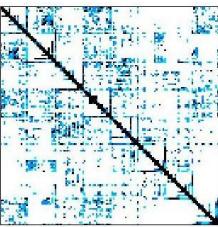
# Some GPU-accelerated Libraries



NVIDIA cuBLAS



NVIDIA cuRAND



NVIDIA cuSPARSE



NVIDIA NPP



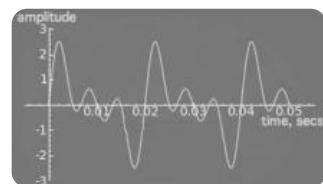
Vector Signal  
Image Processing



GPU Accelerated  
Linear Algebra



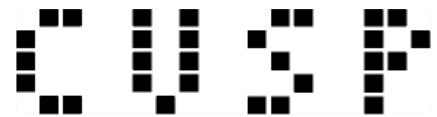
Matrix Algebra on  
GPU and Multicore



NVIDIA cuFFT



ArrayFire Matrix  
Computations



Sparse Linear  
Algebra



C++ STL Features  
for CUDA



# 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

# Drop-In Acceleration (Step 1)



```
int N = 1 << 20;
```

```
// Perform SAXPY on 1M elements: y[] = a*x[] + y[]
saxpy(N, 2.0, d_x, 1, d_y, 1);
```

# Drop-In Acceleration (Step 1)



```
int N = 1 << 20;
```

```
// Perform SAXPY on 1M elements: d_y[] = a * d_x[] + d_y[]
cublassaxpy(N, 2.0, d_x, 1, d_y, 1);
```



Add “cublas” prefix and  
use device variables

# Drop-In Acceleration (Step 2)



```
int N = 1 << 20;  
cUBLASInit();
```



Initialize CUBLAS

```
// Perform SAXPY on 1M elements: d_y[] = a * d_x[] + d_y[]  
cUBLASSaxpy(N, 2.0, d_x, 1, d_y, 1);
```

```
cUBLASShutdown();
```



Shut down CUBLAS



# Drop-In Acceleration (Step 2)

```
int N = 1 << 20;  
cublasInit();  
cublasAlloc(N, sizeof(float), (void**)&d_x);  
cublasAlloc(N, sizeof(float), (void*)&d_y);
```



Allocate device vectors

```
// Perform SAXPY on 1M elements: d_y[] = a * d_x[] + d_y[]  
cublassaxpy(N, 2.0, d_x, 1, d_y, 1);
```

```
cublasFree(d_x);  
cublasFree(d_y);  
cublasShutdown();
```



Deallocate device vectors



# Drop-In Acceleration (Step 2)

```
int N = 1 << 20;  
cUBLASInit();  
cUBLASAlloc(N, sizeof(float), (void**)&d_x);  
cUBLASAlloc(N, sizeof(float), (void*)&d_y);  
  
cUBLASSetVector(N, sizeof(x[0]), x, 1, d_x, 1);  
cUBLASSetVector(N, sizeof(y[0]), y, 1, d_y, 1);  
  
// Perform SAXPY on 1M elements: d_y[] = a * d_x[] + d_y[]  
cUBLASSaxpy(N, 2.0, d_x, 1, d_y, 1);  
  
cUBLASGetVector(N, sizeof(y[0]), d_y, 1, y, 1);  
  
cUBLASFree(d_x);  
cUBLASFree(d_y);  
cUBLASShutdown();
```



Transfer data to GPU



Read data back GPU

# Explore the CUDA (Libraries) Ecosystem



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

[developer.nvidia.com/cuda-tools-ecosystem](http://developer.nvidia.com/cuda-tools-ecosystem)

Log In | Feedback | New Account

Search

NVIDIA DEVELOPER ZONE

DEVELOPER CENTERS TECHNOLOGIES TOOLS RESOURCES COMMUNITY

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 INSIGHT VISUAL STUDIO EDITION 2.2, WITH LOCAL SINGLE GPU CUDA DEBUGGING!

Previous Next

**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**

**CULA Tools**

GPU-accelerated linear algebra library in 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.

**CUSP**

**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 processing primitives.

**NVIDIA CUDA Math Library**

An industry proven, highly accurate collection of standard mathematical functions, providing high performance on GPU.

**Thrust**

A powerful, open source library of parallel algorithms and data structures. Perform GPU-accelerated sort, scan, transform, and reductions.

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 Ways to Accelerate Applications



Applications

Libraries

“Drop-in”  
Acceleration

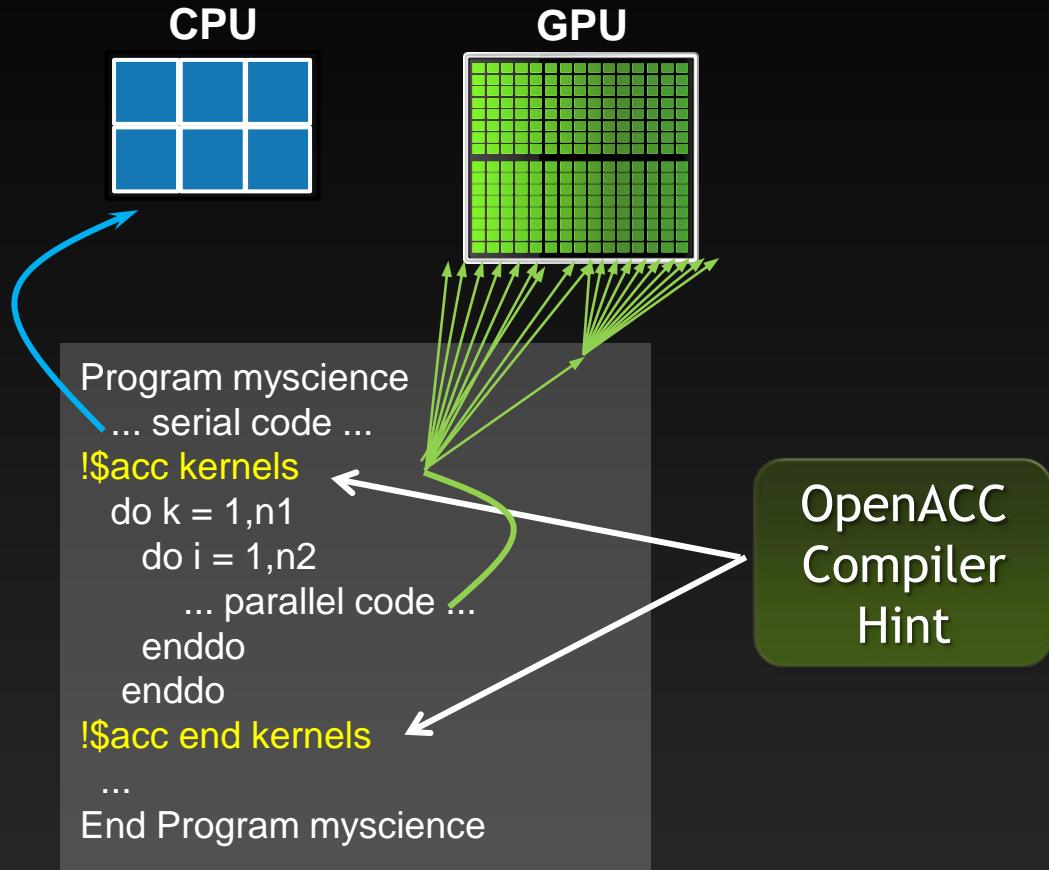
OpenACC  
Directives

Easily Accelerate  
Applications

Programming  
Languages

Maximum  
Flexibility

# OpenACC Directives



Your original  
Fortran or C code

Simple Compiler hints

Compiler Parallelizes code

Works on many-core GPUs &  
multicore CPUs

# OpenACC

## Open Programming Standard for Parallel Computing



“OpenACC will enable programmers to easily develop portable applications that maximize the performance and power efficiency benefits of the hybrid CPU/GPU architecture of Titan.”



*--Buddy Bland, Titan Project Director, Oak Ridge National Lab*

“OpenACC is a technically impressive initiative brought together by members of the OpenMP Working Group on Accelerators, as well as many others. We look forward to releasing a version of this proposal in the next release of OpenMP.”



*--Michael Wong, CEO OpenMP Directives Board*

## OpenACC Standard





# OpenACC

## The Standard for GPU Directives

- **Easy:** Directives are the easy path to accelerate compute intensive applications
- **Open:** OpenACC is an open GPU directives standard, making GPU programming straightforward and portable across parallel and multi-core processors
- **Powerful:** GPU Directives allow complete access to the massive parallel power of a GPU

**OpenACC®**

DIRECTIVES FOR ACCELERATORS

# 2 Basic Steps to Get Started



- **Step 1: Annotate source code with directives:**

```
!$acc data copy(util1,util2,util3) copyin(ip,scp2,scp2i)  
 !$acc parallel loop  
 ...  
 !$acc end parallel  
 !$acc end data
```

- **Step 2: Compile & run:**

```
pgf90 -ta=nvidia -Minfo=accel file.f
```



# OpenACC Directives Example

```
!$acc data copy(A,Anew)
iter=0
do while ( err > tol .and. iter < iter_max )

iter = iter +1
err=0._fp_kind
```

Copy arrays into GPU memory  
within data region

```
!$acc kernels
do j=1,m
do i=1,n
Anew(i,j) = .25_fp_kind *( A(i+1,j) + A(i-1,j) &
                           +A(i,j-1) + A(i,j+1))
err = max( err, Anew(i,j)-A(i,j))
end do
end do
```

Parallelize code inside region

```
!$acc end kernels
IF(mod(iter,100)==0 .or. iter == 1)      print *, iter, err
A= Anew
```

Close off parallel region

```
end do
!$acc end data
```

Close off data region,  
copy data back

# Directives: Easy & Powerful

## Real-Time Object Detection

Global Manufacturer of Navigation Systems



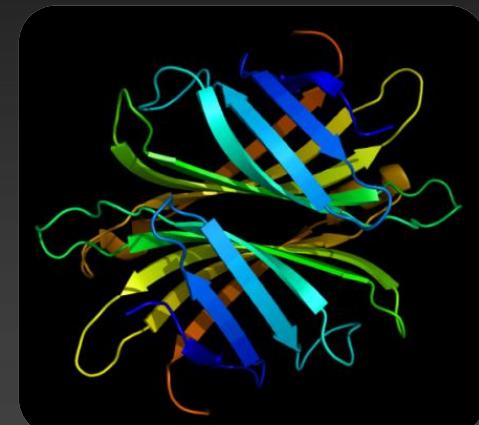
## Valuation of Stock Portfolios using Monte Carlo

Global Technology Consulting Company



## Interaction of Solvents and Biomolecules

University of Texas at San Antonio



## 5x in 40 Hours

“Optimizing code with directives is quite easy, especially compared to CPU threads or writing CUDA kernels. The most important thing is avoiding restructuring of existing code for production applications.”

## 2x in 4 Hours

## 5x in 8 Hours

# Start Now with OpenACC Directives



Sign up for a **free trial** of the  
directives compiler now!

Free trial license to PGI Accelerator

Tools for quick ramp

[www.nvidia.com/gpudirectives](http://www.nvidia.com/gpudirectives)



The screenshot shows the NVIDIA website's main navigation bar with links for DOWNLOAD DRIVERS, COOL STUFF, SHOP, PRODUCTS, TECHNOLOGIES, COMMUNITIES, and SUPPORT. Below this is a green header bar with the word "TESLA". The main content area has a sub-header "GPU COMPUTING SOLUTIONS" with links to Main, What is GPU Computing?, Why Choose Tesla, Industry Software Solutions, Tesla Workstation Solutions, Tesla Data Center Solutions, Tesla Bio Workbench, Where to Buy, Contact US, Sign up for Tesla Alerts, Fermi GPU Computing Architecture, and SOFTWARE AND HARDWARE INFO. A large orange box contains sample C code for calculating pi using OpenACC directives. To the right, there is a testimonial from Professor M. Amin Kay, a quote from Dr. Kerry Black, and a sidebar with a quote from Dr. Michael Minion.

Thousands of cores working for you.  
Based on the [OpenACC](#) standard, GPU directives are the easy, proven way to accelerate your scientific or industrial code. With GPU directives, you can accelerate your code by simply inserting compiler hints into your code and the compiler will automatically map compute-intensive portions of your code to the GPU. Here's an example of how easy a single directive hint can accelerate the calculation of pi. With GPU directives, you can get started and see results in the same afternoon.

```
#include <stdio.h>
#define N 10000
int main(void) {
    double pi = 0.0f; long i;
    #pragma acc region for
    for (i=0; i<N; i++)
    {
        double t=(double)((i+0.5)/N);
        pi += 4.0/(1.0+t*t);
    }
    printf("pi=%f\n",pi/N);
    return 0;
}
```

By starting with a free, 30-day trial of PGI directives today, you are working on the technology that is the foundation of the OpenACC directives standard. OpenACC is:

"I have written micron (written in Fortran 90) properties of two and three dimensional magnetic directives approach embed my existing code perform my computation which resulted in a significant speedup (more than 20 times faster)."  
Learn more

Professor M. Amin Kay  
University of Houston

"The PGI compiler is not just how powerful it is, the software we are writing runs times faster on the NVidia GPUs. We are very pleased and excited about future uses. It's like owning a supercomputer."  
Learn more

Dr. Kerry Black  
University of Melbourne

# 3 Ways to Accelerate Applications



## Applications

Libraries

“Drop-in”  
Acceleration

OpenACC  
Directives

Easily Accelerate  
Applications

Programming  
Languages

Maximum  
Flexibility

# GPU Programming Languages



Numerical analytics ►

MATLAB, Mathematica, LabVIEW

Fortran ►

OpenACC, CUDA Fortran

C ►

OpenACC, CUDA C

C++ ►

Thrust, CUDA C++

Python ►

PyCUDA, Copperhead

C# ►

GPU.NET

## Standard C Code

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

// Perform SAXPY on 1M elements
saxpy_serial(4096*256, 2.0, x, y);
```

## Parallel C Code

```
__global__
void saxpy_parallel(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];
}

// Perform SAXPY on 1M elements
saxpy_parallel<<<4096,256>>>(n,2.0,x,y);
```

# CUDA C++: Develop Generic Parallel Code



CUDA C++ features enable sophisticated and flexible applications and middleware

Class hierarchies

`__device__` methods

Templates

Operator overloading

Functors (function objects)

Device-side new/delete

More...

```
template <typename T>
struct Functor {
    __device__ Functor(_a) : a(_a) {}
    __device__ T operator(T x) { return a*x; }
    T a;
}

template <typename T, typename Oper>
__global__ void kernel(T *output, int n) {
    Oper op(3.7);
    output = new T[n]; // dynamic allocation
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n)
        output[i] = op(i); // apply functor
}
```

# Rapid Parallel C++ Development



- Resembles C++ STL
- High-level interface
  - Enhances developer productivity
  - Enables performance portability between GPUs and multicore CPUs
- Flexible
  - CUDA, OpenMP, and TBB backends
  - Extensible and customizable
  - Integrates with existing software
- Open source



```
// generate 32M random numbers on host
thrust::host_vector<int> h_vec(32 << 20);
thrust::generate(h_vec.begin(),
                 h_vec.end(),
                 rand);

// transfer data to device (GPU)
thrust::device_vector<int> d_vec = h_vec;

// sort data on device
thrust::sort(d_vec.begin(), d_vec.end());

// transfer data back to host
thrust::copy(d_vec.begin(),
             d_vec.end(),
             h_vec.begin());
```

# CUDA Fortran



- Program GPU using Fortran
  - Key language for HPC
- Simple language extensions
  - Kernel functions
  - Thread / block IDs
  - Device & data management
  - Parallel loop directives
- Familiar syntax
  - Use allocate, deallocate
  - Copy CPU-to-GPU with assignment (=)

```
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
    call saxpy<<<4096,256>>>(2**20,3.0,x_d,y_d,)
    y = y_d
    write(*,*) 'max error=', maxval(abs(y-5.0))
end program main
```



# More Programming Languages

Python



PyCUDA



C# .NET



GPU.NET



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/>

A large, metallic, three-dimensional NVIDIA logo watermark is positioned diagonally across the background. It features a stylized 'N' shape with a textured surface, appearing to be cut through a dark, reflective material.

# Six Ways to SAXPY

Programming Languages  
for GPU Computing



# Single precision Alpha X Plus Y (**SAXPY**)



Part of Basic Linear Algebra Subroutines (BLAS) Library

$$z = \alpha x + y$$

$x, y, z$  : vector  
 $\alpha$  : scalar

GPU SAXPY in multiple languages and libraries

A menagerie\* of possibilities, not a tutorial

# OpenACC Compiler Directives



## Parallel C Code

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

...
// Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
...
```

## Parallel Fortran Code

```
subroutine saxpy(n, a, x, y)
    real :: x(:), y(:), a
    integer :: n, i
 !$acc kernels
    do i=1,n
        y(i) = a*x(i)+y(i)
    enddo
 !$acc end kernels
end subroutine saxpy

...
! Perform SAXPY on 1M elements
call saxpy(2**20, 2.0, x_d, y_d)
...
```

# CUBLAS Library



## *Serial BLAS Code*

```
int N = 1<<20;  
  
...  
  
// use your choice of blas library  
  
// Perform SAXPY on 1M elements  
blas_saxpy(N, 2.0, x, 1, y, 1);
```

## *Parallel cuBLAS Code*

```
int N = 1<<20;  
  
cublasInit();  
cublasSetVector(N, sizeof(x[0]), x, 1, d_x, 1);  
cublasSetVector(N, sizeof(y[0]), y, 1, d_y, 1);  
  
// Perform SAXPY on 1M elements  
cublassaxpy(N, 2.0, d_x, 1, d_y, 1);  
  
cublasGetVector(N, sizeof(y[0]), d_y, 1, y, 1);  
  
cublasshutdown();
```

You can also call cuBLAS from Fortran,  
C++, Python, and other languages  
<http://developer.nvidia.com/cublas>

## 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);
```

# 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);
```

## *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);
```

# 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
```

# 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)
```

## *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)
```

CU

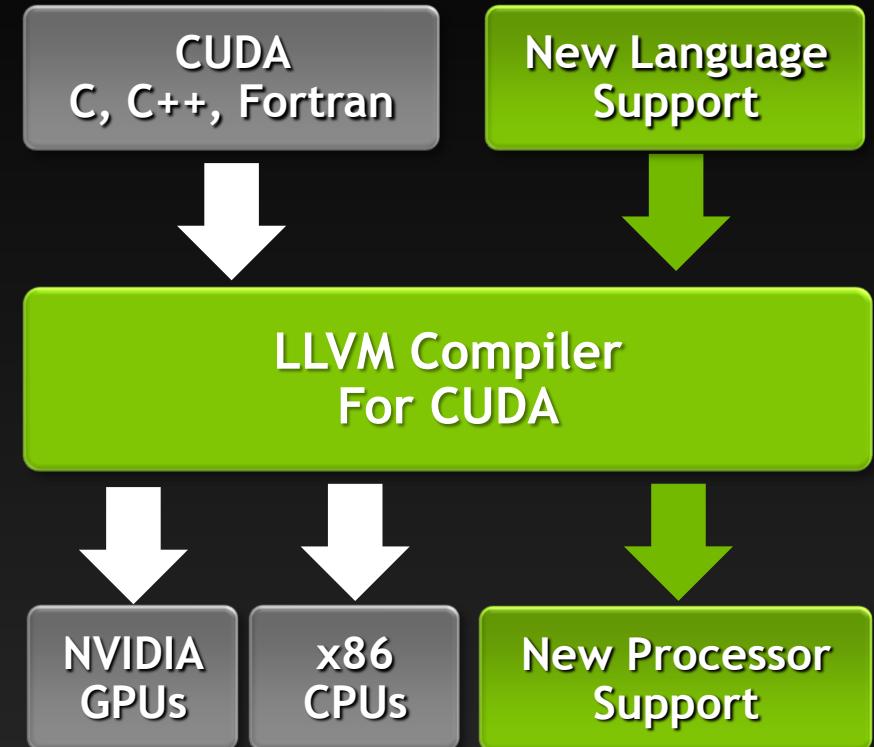


# Enabling Endless Ways to SAXPY

Developers want to build front-ends for Java, Python, R, DSLs

Target other processors like ARM, FPGA, GPUs, x86

CUDA Compiler Contributed to Open Source LLVM





**Thank you**  
**[developer.nvidia.com](https://developer.nvidia.com)**

