



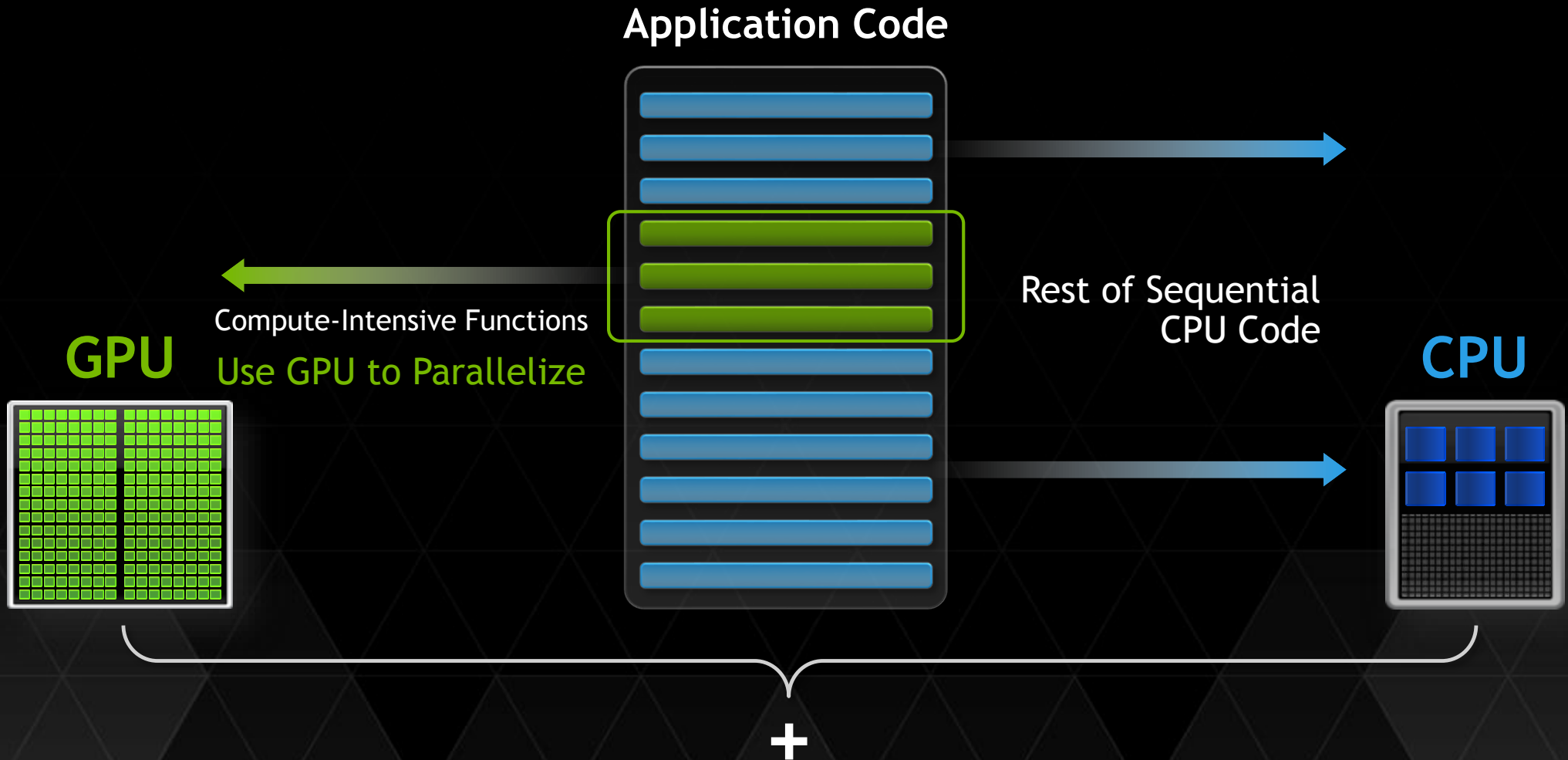
INTRODUCTION TO GPU COMPUTING PROGRAMMING

Carlo Nardone, Sr. Solution Architect, PSG

AGENDA

- 1 Intro
- 2 Libraries
- 3 OpenACC
- 4 Languages
- 5 An example: 6 ways to SAXPY
- 6 Software Roadmap

SMALL CHANGES, BIG SPEED-UP



3 WAYS TO ACCELERATE APPLICATIONS

Applications

Libraries

“Drop-in”
Acceleration

OpenACC
Directives

Easily Accelerate
Applications

Programming
Languages

Maximum
Performance

3 WAYS TO ACCELERATE APPLICATIONS

Applications

Libraries

“Drop-in”
Acceleration

OpenACC
Directives

Easily Accelerate
Applications

Programming
Languages

Maximum
Flexibility

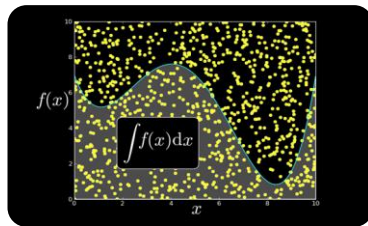
LIBRARIES: EASY, HIGH-QUALITY ACCELERATION

- **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

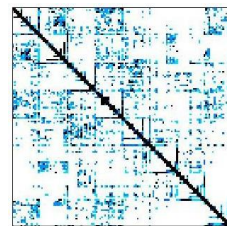
SOME GPU-ACCELERATED LIBRARIES



NVIDIA cuBLAS



NVIDIA cuRAND



NVIDIA cuSPARSE



NVIDIA NPP



Vector Signal
Image Processing



GPU Accelerated
Linear Algebra



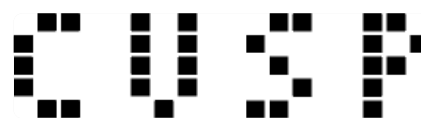
Matrix Algebra on
GPU and Multicore



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);
```



Transfer data to GPU

```
// 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);
```



Read data back GPU

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

EXPLORE THE CUDA (LIBRARIES) ECOSYSTEM

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

developer.nvidia.com/cuda-tools-ecosystem

The screenshot displays the NVIDIA Developer Zone website. At the top, there's a navigation bar with the NVIDIA logo, 'DEVELOPER ZONE' text, and links for 'Log In', 'Feedback', and 'New Account'. Below this is a search bar and a menu with 'DEVELOPER CENTERS', 'TECHNOLOGIES', 'TOOLS', 'RESOURCES', and 'COMMUNITY'. The main content area is titled 'GPU-Accelerated Libraries' and includes a brief introduction: '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](#).' The page features a grid of library cards:

- 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.
- 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 1000s of image processing functions.
- NVIDIA CUDA Math Library**: An industry proven, highly accurate collection of standard mathematical functions, providing high performance for a wide range of applications.
- Thrust**: A powerful, open source library of parallel algorithms and data structures. Perform GPU-accelerated sort, scan, transform and reductions.

 On the right side, there's a 'QUICKLINKS' section with links to 'The NVIDIA Registered Developer Program', 'Registered Developers Website', 'NVIDIA Developer (old site)', 'CUDA Newsletter', 'CUDA Downloads', 'CUDA GPUs', 'Get Started - Parallel Computing', 'CUDA Spotlights', and 'CUDA Tools & Ecosystem'. Below that is a 'FEATURED ARTICLES' section with a featured article titled 'INTRODUCING NVIDIA NSIGHT VISUAL STUDIO EDITION 2.2, WITH LOCAL SINGLE GPU CUDA DEBUGGING!'. At the bottom right is a 'LATEST NEWS' section with articles like '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', and 'CUDA Spotlight: ...'.

3 WAYS TO ACCELERATE APPLICATIONS

Applications

Libraries

“Drop-in”
Acceleration

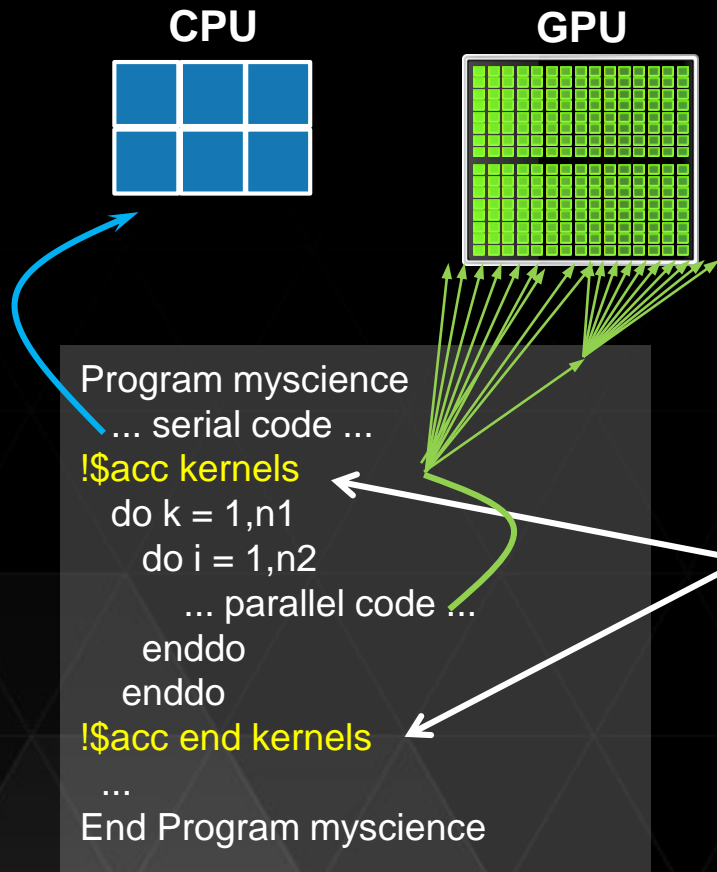
OpenACC
Directives

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

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
```

```
!$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
```

```
!$acc end kernels
```

```
IF(mod(iter,100)==0 .or. iter == 1)    print *, iter, err
```

```
A= Anew
```

```
end do
```

```
!$acc end data
```

Copy arrays into GPU memory
within data region

Parallelize code inside region

Close off parallel region

Close off data region,
copy data back

Directives: Easy & Powerful

Real-Time Object Detection

Global Manufacturer of Navigation Systems



5x in 40 Hours

Valuation of Stock Portfolios using Monte Carlo

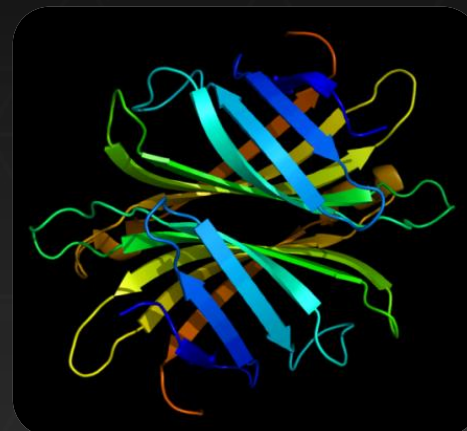
Global Technology Consulting Company



2x in 4 Hours

Interaction of Solvents and Biomolecules

University of Texas at San Antonio



5x in 8 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.”

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



GPU COMPUTING SOLUTIONS

- 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

SOFTWARE AND HARDWARE INFO

- Tesla Product Literature
- Tesla Software Features
- Software Development Tools
- CUDA Training and Consulting Services
- GPU Cloud Computing Service Providers
- OpenACC GPU Directives

Accelerate Your Scientific Code with OpenACC

The Open Standard for GPU Accelerator Directives

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 dimensional magnetic directives approach en port my existing code perform my computat which resulted in a speedup (more than 20 computation." [Learn more](#)

Professor M. Amin Kay
University of Houston

"The PGI compiler is not just how powerful it is software we are writing times faster on the NV are very pleased and future uses. It's like on supercomputer." [Learn more](#)

Dr. Kerry Black
University of Melbourne

22 NVIDIA

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

CUDA C

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**



Wolfram Mathematica⁸

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>

GPU.NET

<http://tidepowerd.com>

Thrust C++ Template Library

<http://developer.nvidia.com/thrust>

MATLAB

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

CUDA Fortran

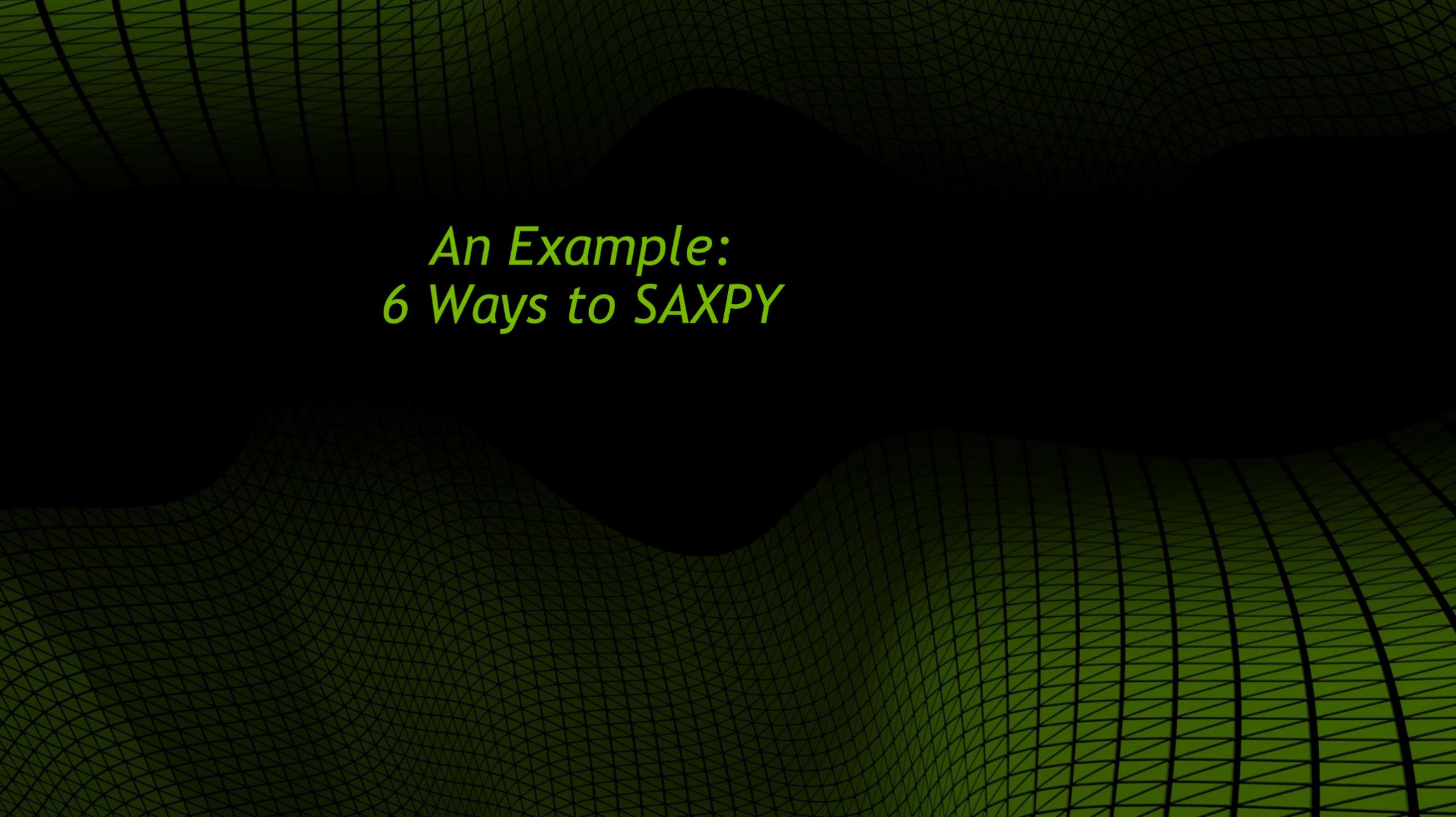
<http://developer.nvidia.com/cuda-toolkit>

Mathematica

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

PyCUDA (Python)

<http://mathematician.de/software/pycuda>



*An Example:
6 Ways to SAXPY*

SINGLE PRECISION ALPHA X PLUS Y (SAXPY)

Part of Basic Linear Algebra Subroutines (BLAS) Library

$$z = \alpha x + y$$

x, y, z : vector

α : scalar

GPU SAXPY in multiple languages and libraries

A menagerie* of possibilities, not a tutorial

1

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)  
...
```

2

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);
```

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);
```

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);
```

5

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

```

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>

ENABLING ENDLESS WAYS TO SAXPY

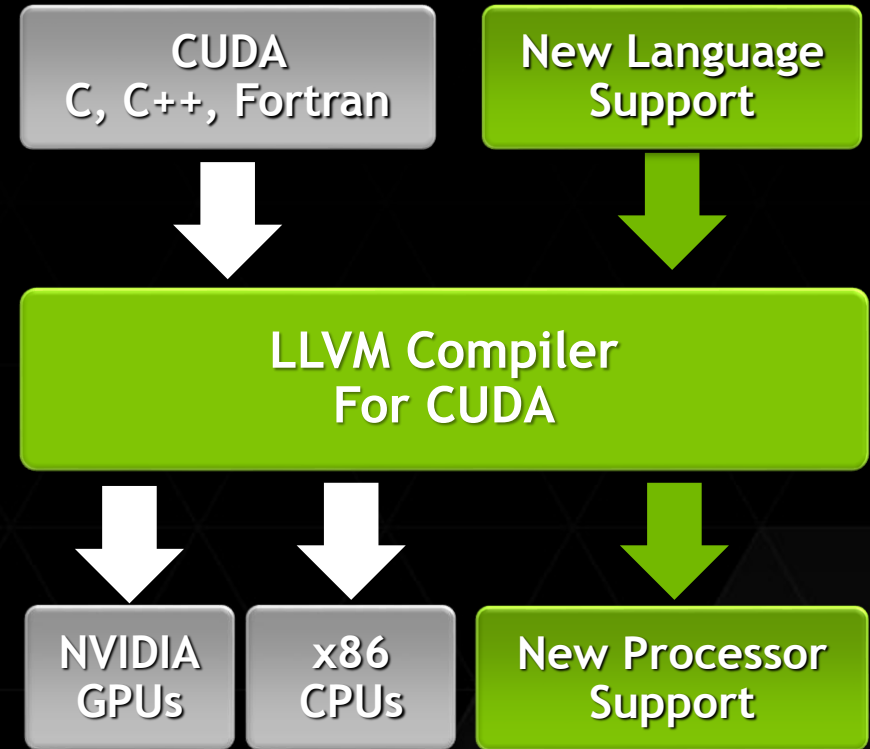
Developers want to build
front-ends for

Java, Python, R, DSLs

Target other processors like

ARM, FPGA, GPUs, x86

**CUDA Compiler Contributed to
Open Source LLVM**



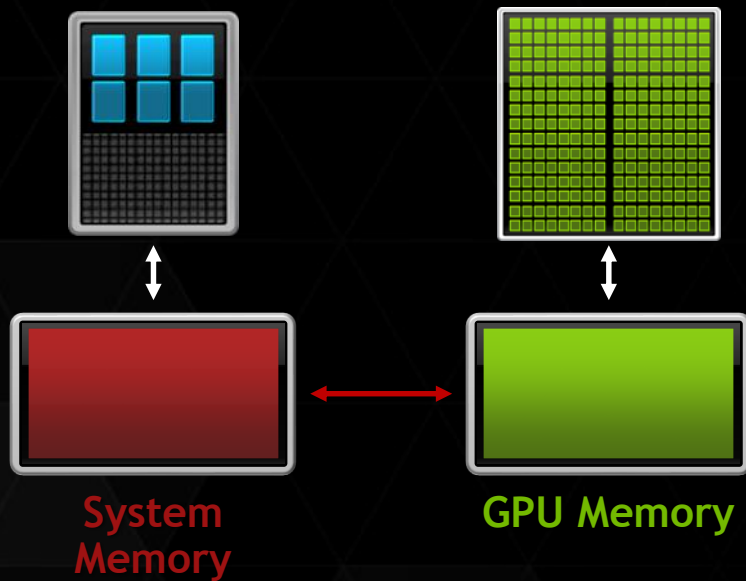
The background features a green grid pattern that is warped and curved, creating a sense of depth and movement. Overlaid on this grid are several large, solid black, wavy shapes that resemble liquid or smoke, adding a dynamic and modern feel to the design.

Software Roadmap

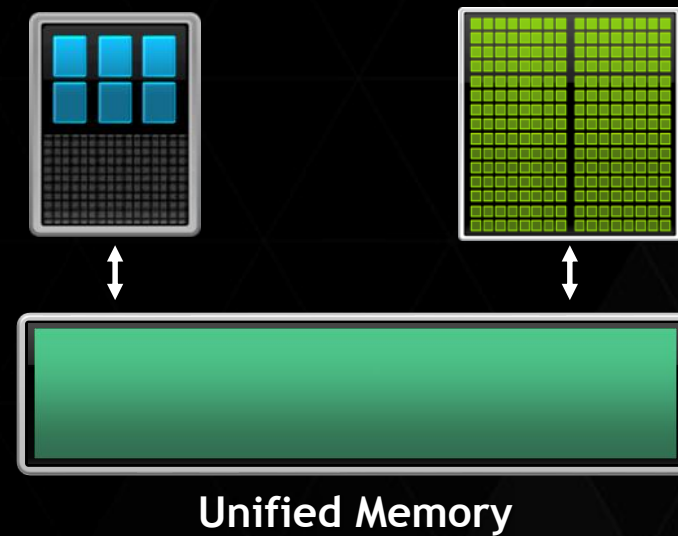
UNIFIED MEMORY

Dramatically Lower Developer Effort

Developer View Today



Developer View With Unified Memory



UNIFIED MEMORY DELIVERS

1. Simpler Programming & Memory Model

- Single pointer to data, accessible anywhere
- Tight language integration
- Greatly simplifies code porting

2. Performance Through Data Locality

- Migrate data to accessing processor
- Guarantee global coherency
- Still allows *cudaMemcpyAsync()* hand tuning

SIMPLIFIED MEMORY MANAGEMENT

CPU Code

```
void sortfile(FILE *fp, int N) {  
    char *data;  
    data = (char *)malloc(N);  
  
    fread(data, 1, N, fp);  
  
    qsort(data, N, 1, compare);  
  
    use_data(data);  
  
    free(data);  
}
```

CUDA 6 Code with Unified Memory

```
void sortfile(FILE *fp, int N) {  
    char *data;  
    cudaMallocManaged(&data, N);  
  
    fread(data, 1, N, fp);  
  
    qsort<<<...>>>(data, N, 1, compare);  
    cudaDeviceSynchronize();  
  
    use_data(data);  
  
    cudaFree(data);  
}
```

GRAFICAL & CLI PROFILING TOOLS

- **NVIDIA® Visual Profiler**

- Standalone (**nvvp**)   
- Integrated into NVIDIA® Nsight™ Eclipse Edition (**nsight**)  

- **nvprof**     *

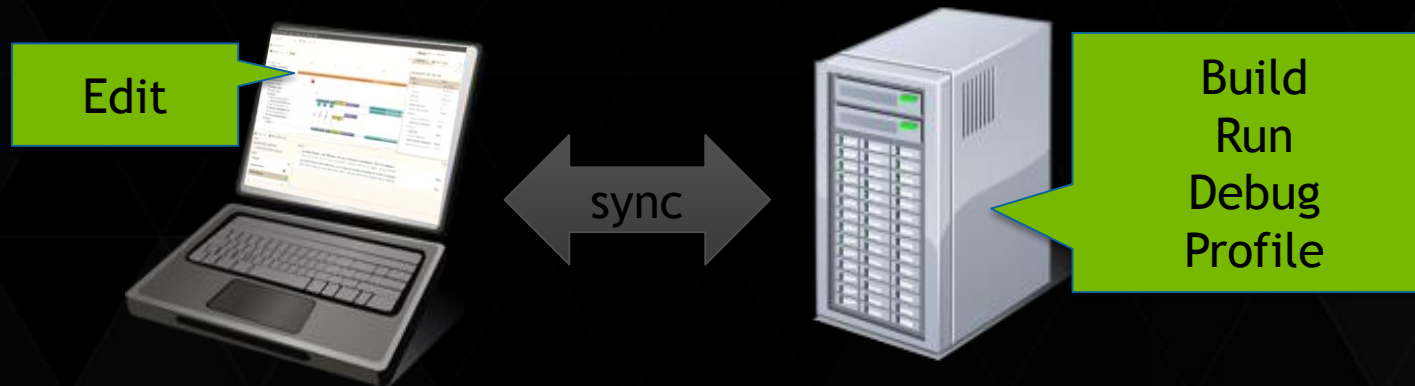
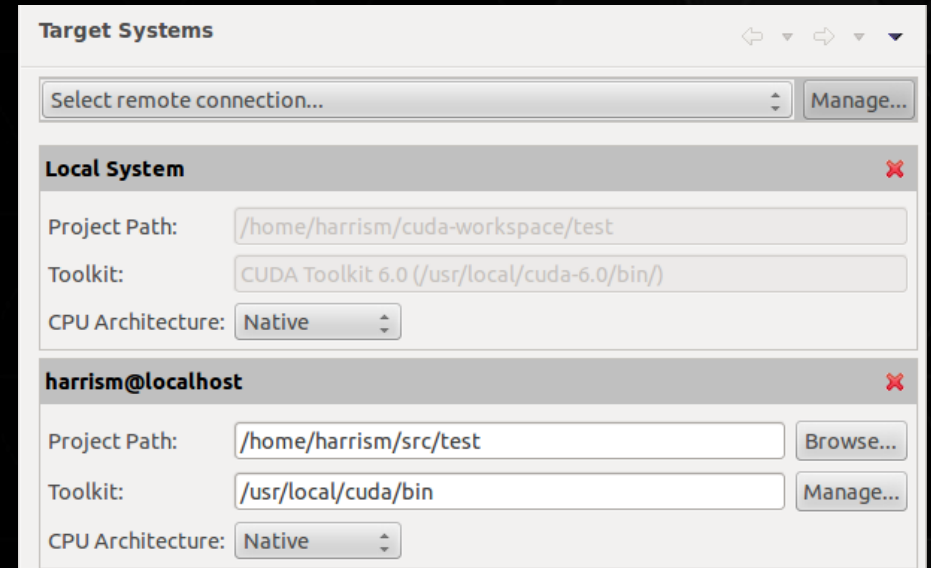
- **NVIDIA® Nsight™ Visual Studio Edition** 

- Old environment variable based command-line profiler still available     *

* Android CUDA APK profiling not supported (yet)

REMOTE DEVELOPMENT TOOLS

- ▶ Local IDE, remote application
 - ▶ Edit locally, build & run remotely
 - ▶ Automatic sync via ssh
 - ▶ Cross-compilation to ARM
- ▶ Full debugging & profiling via remote connection



EXTENDED (XT) LIBRARY INTERFACES

Automatic Scaling to multiple GPUs per node

cuFFT 2D/3D & cuBLAS level 3

Operate directly on large datasets that reside in CPU memory

developer.nvidia.com/cublasxt

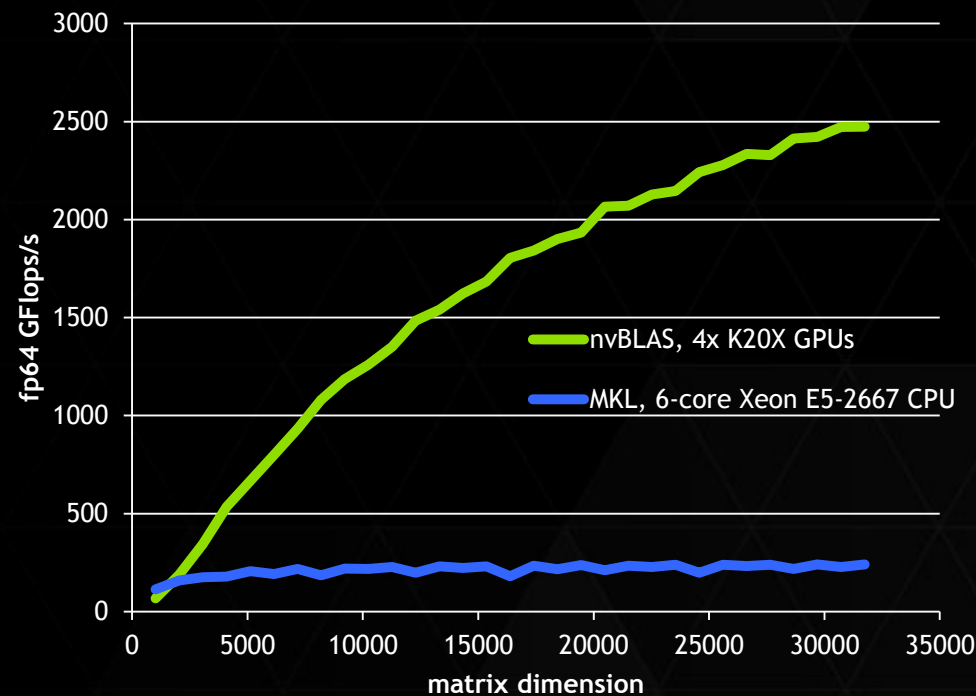


NEW DROP-IN NVBLAS LIBRARY

- ▶ Drop-in replacement for CPU-only BLAS
 - ▶ Automatically route BLAS3 calls to cuBLAS
- ▶ Example: Drop-in Speedup for R

```
> LD_PRELOAD=/usr/local/cuda/lib64/libnvblas.so R
> A <- matrix(rnorm(4096*4096), nrow=4096, ncol=4096)
> B <- matrix(rnorm(4096*4096), nrow=4096, ncol=4096)
> system.time(C <- A %*% B)
  user  system elapsed 
0.348   0.142   0.289
```
- ▶ Use in any app that uses standard BLAS3
 - ▶ Octave, Scilab, etc.

Matrix-Matrix Multiplication in R



GOALS FOR THE CUDA PLATFORM

Simplicity

- Learn, adopt, & use parallelism with ease

Productivity

- Quickly achieve feature & performance goals

Portability

- Write code that can execute on all targets

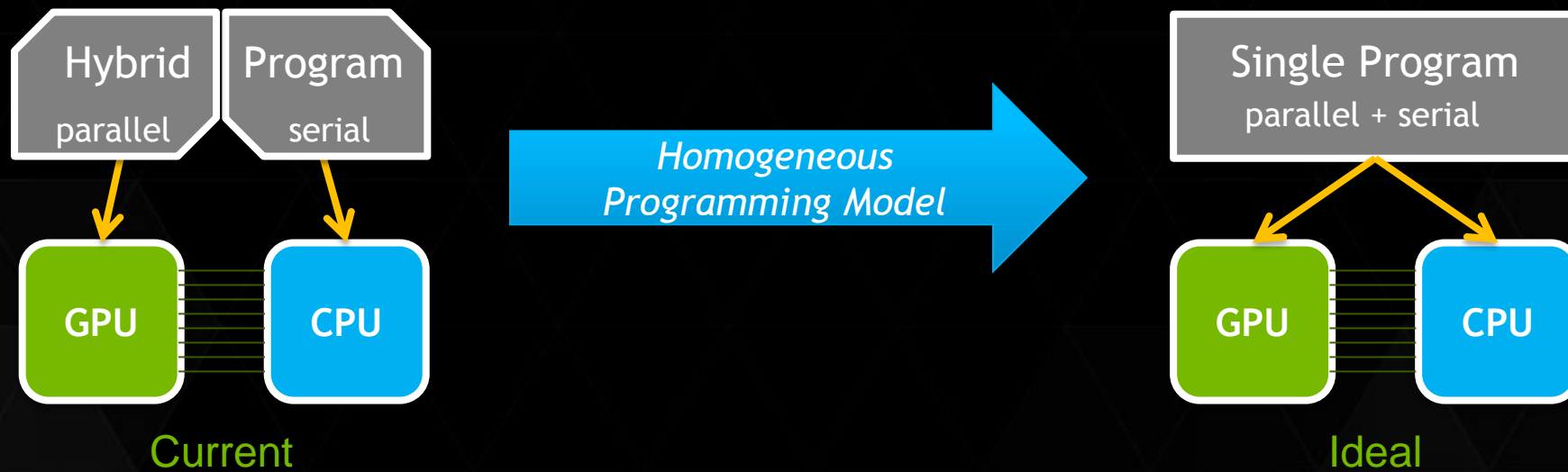
Performance

- High absolute performance and scalability

SIMPLER HETEROGENEOUS APPLICATIONS

We want: *homogeneous* programs, *heterogeneous* execution

- ▶ Unified programming model includes parallelism in language
- ▶ Abstract heterogeneous execution via Runtime or Virtual Machine



PARALLELISM IN MAINSTREAM LANGUAGES

- ▶ Enable more programmers to write parallel software
- ▶ Give programmers the choice of language to use
- ▶ GPU support in key languages



C++ PARALLEL ALGORITHMS LIBRARY

```
std::vector<int> vec = ...  
  
// previous standard sequential loop  
std::for_each(vec.begin(), vec.end(), f);  
  
// explicitly sequential loop  
std::for_each(std::seq, vec.begin(), vec.end(), f);  
  
// permitting parallel execution  
std::for_each(std::par, vec.begin(), vec.end(), f);
```

- Complete set of parallel primitives: `for_each`, `sort`, `reduce`, `scan`, etc.
- ISO C++ committee voted unanimously to accept as official tech. specification working draft

A Parallel Algorithms Library | N3724

Jared Hoberock Jaydeep Marathe Michael Garland Olivier Giroux
Vinod Grover {jhoberock, jmarathe, mgarland, ogiroux, vgrover}@nvidia.com
Artur Laksberg Herb Sutter {arturl, hsutter}@microsoft.com Arch Robison

Document Number: N3960
Date: 2014-02-28
Reply to: Jared Hoberock
NVIDIA Corporation
jhoberock@nvidia.com

Working Draft, Technical
Specification for C++ Extensions for
Parallelism, Revision 1

N3960 Technical Specification Working Draft:
<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3960.pdf>
Prototype:
<https://github.com/n3554/n3554>

GNU LINUX GCC TO SUPPORT OPENACC

- **Open Source**
 - GCC Efforts by Samsung & Mentor Graphics
- **Pervasive Impact**
 - Free to all Linux users
- **Mainstream**
 - Most Widely Used HPC Compiler



Incorporating OpenACC into GCC is an excellent example of open source and open standards working together to make accelerated computing broadly accessible to all Linux developers. ”

Oscar Hernandez
Oak Ridge National Laboratories



NUMBA PYTHON COMPILER

- ▶ Free and open source compiler for array-oriented Python
- ▶ NEW numba.cuda module integrates CUDA directly into Python

```
@cuda.jit("void(float32[:], float32, float32[:], float32[:])")
def saxpy(out, a, x, y):
    i = cuda.grid(1)
    out[i] = a * x[i] + y[i]

# Launch saxpy kernel
saxpy[griddim, blockdim](out, a, x, y)
```

- ▶ <http://numba.pydata.org/>



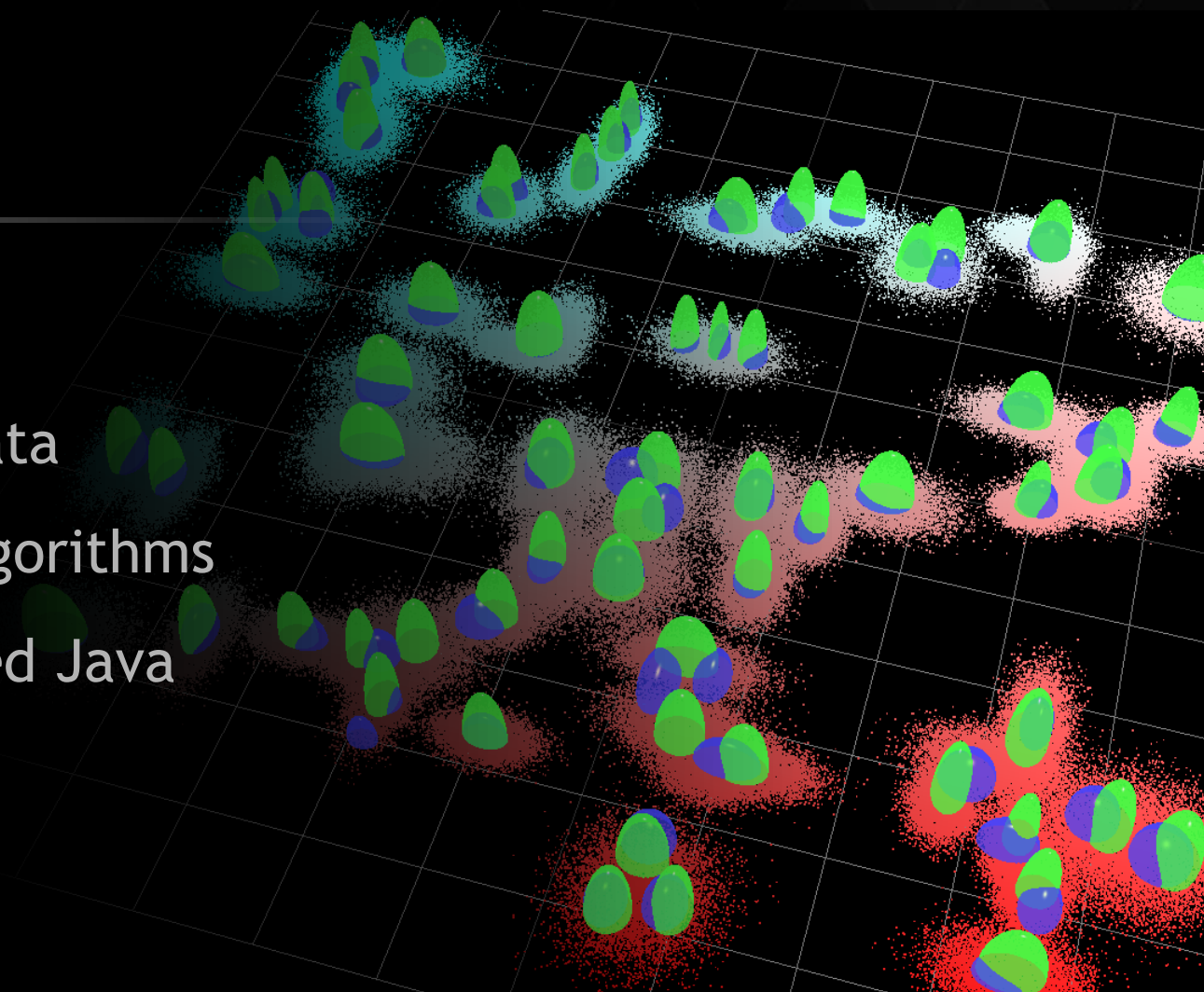
GPU-ACCELERATED HADOOP



Extract insights from customer data

Data Analytics using clustering algorithms

Developed using CUDA-accelerated Java



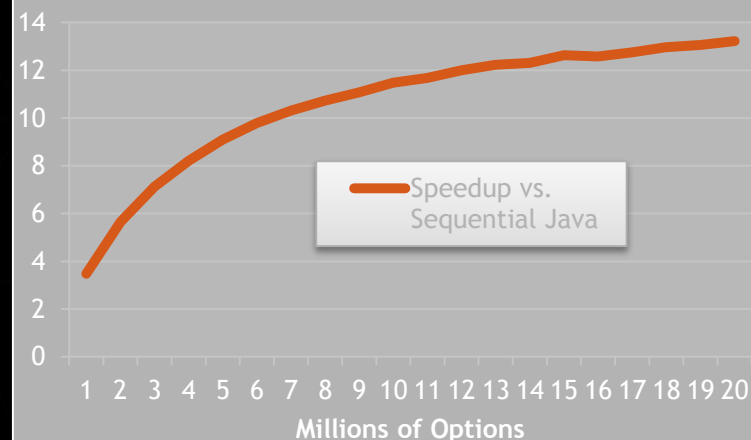
COMPILE JAVA FOR GPUS



- Approach: apply a closure to a set of arrays

```
// vector addition
float[] X = {1.0, 2.0, 3.0, 4.0, ... };
float[] Y = {9.0, 8.1, 7.2, 6.3, ... };
float[] Z = {0.0, 0.0, 0.0, 0.0, ... };
jog.foreach(X, Y, Z, new jogContext(),
    new jogClosureRet<jogContext>() {
        public float execute(float x, float y) {
            return x + y;
        }
    }
);
```

Java Black-Scholes Options
Pricing Speedup



- foreach iterations parallelized over GPU threads
 - Threads run closure execute() method



THANKS!

Carlo Nardone

+39 335 5828197

cnardone@nvidia.com

