Introduction to GPU Computing
Add GPUs: Accelerate Science Applications
GPUs Accelerate Science

- Medical Imaging
  U of Utah

- Molecular Dynamics
  U of Illinois, Urbana

- Video Transcoding
  Elemental Tech

- Matlab Computing
  AccelerEyes

- Astrophysics
  RIKEN

- Financial Simulation
  Oxford

- Linear Algebra
  Universidad Jaime

- 3D Ultrasound
  Techniscan

- Quantum Chemistry
  U of Illinois, Urbana

- Gene Sequencing
  U of Maryland
Small Changes, Big Speed-up

Application Code

GPU
Compute-Intensive Functions
Use GPU to Parallelize

Rest of Sequential CPU Code

CPU
3 Ways to Accelerate Applications

Applications

Libraries
“Drop-in” Acceleration

OpenACC Directives
Easily Accelerate Applications

Programming Languages
Maximum Performance
Apps on Hokiespeed (module avail)

- OpenFOAM - (solver packs available)
- fftw -> cuFFT
- gromacs
- hmmer
- Python -> PyCUDA, copperhead, NumbaPro, others
- Beagle
- Scalapack -> MAGMA
- tau
- ipp -> npp
- lammps

More: http://www.nvidia.com/teslaapps
3 Ways to Accelerate 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
- GPU VSIPL
- CULA tools
- MAGMA
- NVIDIA cuFFT
- Rogue Wave Software
- ArrayFire
- CUSP
- Thrust

Vector Signal Image Processing
GPU Accelerated Linear Algebra
Matrix Algebra on GPU and Multicore
Sparse Linear Algebra
C++ STL Features for CUDA
3 Steps to CUDA-accelerated application

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

```c
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

```bash
nvcc myobj.o -l cublas
```
int N = 1 << 20;

// Perform SAXPY on 1M elements: y[] = a*x[] + y[]
saxpy(N, 2.0, d_x, 1, d_y, 1);
int N = 1 << 20;

// Perform SAXPY on 1M elements: \(d_y[] = a \times d_x[] + d_y[]\)
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);

Add “cublas” prefix and use device variables
int N = 1 << 20;
cublasInit();

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

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

// 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();
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();
Explore the CUDA (Libraries) Ecosystem

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

developer.nvidia.com/cuda-tools-ecosystem
3 Ways to Accelerate Applications

- **Applications**
  - Libraries
    - “Drop-in” Acceleration
  - OpenACC Directives
    - Easily Accelerate Applications
  - Programming Languages
    - Maximum Flexibility
OpenACC Directives

Program myscience
... serial code ...
!$acc kernels
do k = 1,n1
do i = 1,n2
... parallel code ...
enddo
enddo
!$acc end kernels
... End Program myscience

Simple Compiler hints

Compiler Parallelizes code

Works on many-core GPUs & multicore CPUs

Your original Fortran or C code
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:

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

**Step 2:** Compile & run:

```
pgf90 -ta=nvidia -Minfo=accel file.f
```
OpenACC Directives Example

```plaintext
!$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

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
2x in 4 Hours
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.”

-- Developer at the Global Manufacturer of Navigation Systems
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
3 Ways to Accelerate Applications

- **Libraries**: “Drop-in” Acceleration
- **OpenACC Directives**: Easily Accelerate Applications
- **Programming Languages**: Maximum Flexibility
## GPU Programming Languages

<table>
<thead>
<tr>
<th>Category</th>
<th>Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numerical analytics</td>
<td>MATLAB, Mathematica, LabVIEW</td>
</tr>
<tr>
<td>Fortran</td>
<td>OpenACC, CUDA Fortran</td>
</tr>
<tr>
<td>C</td>
<td>OpenACC, CUDA C</td>
</tr>
<tr>
<td>C++</td>
<td>OpenACC, Thrust, CUDA C++</td>
</tr>
<tr>
<td>Python</td>
<td>PyCUDA, Copperhead</td>
</tr>
<tr>
<td>C#</td>
<td>Cudafy</td>
</tr>
<tr>
<td><strong>Java, R, DSL’s, etc.</strong></td>
<td></td>
</tr>
</tbody>
</table>
CUDA C

Standard C Code

```c
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

```c
__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

```cpp
// 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 (=)

```fortran
module mymodule
  contains
    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
  - CUDAFY

- Numerical Analytics
  - MATLAB
  - Wolfram Mathematica®
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++

- Thrust C++ Template Library

- CUDA Fortran

- cudafy
  [http://cudafy.codeplex.com](http://cudafy.codeplex.com)

- MATLAB

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

- Mathematica
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

---

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);
...

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)
...
You can also call cuBLAS from Fortran, C++, Python, and other languages

http://developer.nvidia.com/cublas
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);

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);
```

www.boost.org/libs/lambda  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

```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

```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
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.
Where to get help?

- Sign up as a registered developer: [https://developer.nvidia.com/](https://developer.nvidia.com/)
- Access the NVIDIA community forums: [https://devtalk.nvidia.com/](https://devtalk.nvidia.com/)
- StackOverflow:
  - CUDA: [http://stackoverflow.com/questions/tagged/cuda](http://stackoverflow.com/questions/tagged/cuda)
Thank you

developer.nvidia.com