AGENDA

1 Intro
2 Libraries
3 Directives
4 Languages
5 An example: 6 ways to SAXPY
6 Software Roadmap
<table>
<thead>
<tr>
<th>ABCPL</th>
<th>CORRELATE</th>
<th>GLU</th>
<th>PORT</th>
<th>Paralation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACE</td>
<td>CPS</td>
<td>GUARD</td>
<td>Legion</td>
<td>Parallel2</td>
</tr>
<tr>
<td>ACT++</td>
<td>CRL</td>
<td>HAaL</td>
<td>Meta Chaos</td>
<td>Paralax</td>
</tr>
<tr>
<td>Active Messages</td>
<td>CSP</td>
<td>Haskell</td>
<td>Millipede</td>
<td>Paralax++</td>
</tr>
<tr>
<td>Adi</td>
<td>Cthreads</td>
<td>HPC++</td>
<td>Mirage</td>
<td>ParC</td>
</tr>
<tr>
<td>Admuth</td>
<td>CUMULVS</td>
<td>JAVA</td>
<td>Mirge</td>
<td>ParLib+</td>
</tr>
<tr>
<td>ADDAP</td>
<td>DAGGER</td>
<td>HORUS</td>
<td>MpiC</td>
<td>ParLib++</td>
</tr>
<tr>
<td>AFAPI</td>
<td>DAPPLE</td>
<td>IMPACT</td>
<td>MOSIX</td>
<td>ParLin</td>
</tr>
<tr>
<td>ALWAN</td>
<td>AM</td>
<td>ISIS</td>
<td>Modula-P</td>
<td>Parti</td>
</tr>
<tr>
<td>AMDC</td>
<td>AM</td>
<td>JAVAR</td>
<td>Modula-2*</td>
<td>pC</td>
</tr>
<tr>
<td>AppLeS</td>
<td>DCE++</td>
<td>javaRMI</td>
<td>Multipol</td>
<td>pC++</td>
</tr>
<tr>
<td>Anoeba</td>
<td>DDD</td>
<td>javaPG</td>
<td>MPI</td>
<td>PCN</td>
</tr>
<tr>
<td>ARTS</td>
<td>DICE</td>
<td>JavaSpace</td>
<td>MPI++</td>
<td>PCP-</td>
</tr>
<tr>
<td>Athapascan-0b</td>
<td>DOLIB</td>
<td>JavaSpace</td>
<td>Mumin</td>
<td>PEACE</td>
</tr>
<tr>
<td>Aurora</td>
<td>DOPC</td>
<td>JDLD</td>
<td>Nano-Threads</td>
<td>PCU</td>
</tr>
<tr>
<td>Automap</td>
<td>DOSMOS</td>
<td>Joyce</td>
<td>NESL</td>
<td>PET</td>
</tr>
<tr>
<td>bb_threads</td>
<td>DRl</td>
<td>Khoros</td>
<td>NetClasses++</td>
<td>PETSc</td>
</tr>
<tr>
<td>Blaze</td>
<td>DSM-Threads</td>
<td>Karma</td>
<td>Nexus</td>
<td>PENNY</td>
</tr>
<tr>
<td>BSP</td>
<td>Ease</td>
<td>KOAN/Fortran-S</td>
<td>Nimrod</td>
<td>Phosphorus</td>
</tr>
<tr>
<td>BlockComm</td>
<td>ECO</td>
<td>LAM</td>
<td>NOW</td>
<td>POET</td>
</tr>
<tr>
<td>C++</td>
<td>Eiffel</td>
<td>Lilac</td>
<td>Objective Linda</td>
<td>Polar</td>
</tr>
<tr>
<td><strong>C</strong></td>
<td>Elidean</td>
<td>Linda</td>
<td>Occam</td>
<td>POOMA</td>
</tr>
<tr>
<td>C++ in C</td>
<td>EMUL</td>
<td>JAVA</td>
<td>Omega</td>
<td>POOL-T</td>
</tr>
<tr>
<td>Cashmere</td>
<td>EPL</td>
<td>WWWinda</td>
<td>OpenMP</td>
<td>PRESTO</td>
</tr>
<tr>
<td>C4++</td>
<td>Eclisburr</td>
<td>ISETL-Linda</td>
<td>Orca</td>
<td>P-RIO</td>
</tr>
<tr>
<td>CC++</td>
<td>Express</td>
<td>PatLin</td>
<td>OOF90</td>
<td>Prospero</td>
</tr>
<tr>
<td>Chu</td>
<td>Falcon</td>
<td>Eilean</td>
<td>P++</td>
<td>Proteus</td>
</tr>
<tr>
<td>Charlotte</td>
<td>Filaments</td>
<td>F-4-Linda</td>
<td>PSL</td>
<td>QPC++</td>
</tr>
<tr>
<td>Charm</td>
<td>FM</td>
<td>Glenda</td>
<td>p4-Linda</td>
<td>PVM</td>
</tr>
<tr>
<td>Charm++</td>
<td>FLASH</td>
<td>POSYBL</td>
<td>Pablo</td>
<td>PSI</td>
</tr>
<tr>
<td>Cilk</td>
<td>The FORCE</td>
<td>Objective-Linda</td>
<td>PADE</td>
<td>PSDM</td>
</tr>
<tr>
<td>Cilk</td>
<td>Fork</td>
<td>LIPS</td>
<td>Quake</td>
<td>Quark</td>
</tr>
<tr>
<td>CM-Fortran</td>
<td>Fortran-M</td>
<td>Locust</td>
<td>Quicks</td>
<td>Quick Threads</td>
</tr>
<tr>
<td>Converse</td>
<td>FX</td>
<td>Lparx</td>
<td>Quicks</td>
<td>Sage++</td>
</tr>
<tr>
<td>Code</td>
<td>GAMMA</td>
<td>Lucid</td>
<td>Quick</td>
<td>SCANAL</td>
</tr>
<tr>
<td>COOL</td>
<td>Glenda</td>
<td>Maisie</td>
<td>Random</td>
<td>SAM</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PARALLEL PROGRAMMING LANGUAGES: 90S**
<table>
<thead>
<tr>
<th>PARALLEL PROGRAMMING LANGUAGES: 00S</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM++</td>
</tr>
<tr>
<td>ArBB</td>
</tr>
<tr>
<td>BSP</td>
</tr>
<tr>
<td>C++11</td>
</tr>
<tr>
<td>C++AMP</td>
</tr>
<tr>
<td>Charm++</td>
</tr>
<tr>
<td>Chapel</td>
</tr>
<tr>
<td>Cilk++</td>
</tr>
<tr>
<td>CnC</td>
</tr>
<tr>
<td>coArray Fortran</td>
</tr>
<tr>
<td>Codelets</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
MPI AND OPENMP

The two most common HPC parallel programming standards

- MPI (Message Passing Interface) is the standard for parallel programming on distributed memory parallel architectures (e.g. clusters)

- OpenMP (Open MultiProcessing) is the standard for parallel programming on shared memory parallel architectures (e.g. single multicore, multisocket servers)

- MPI and OpenMP can interoperate (hybrid parallel programming)

- CUDA and other GPU accelerated computing models can interoperate with MPI and OpenMP (with some care)
SMALL CHANGES, BIG SPEED-UP

Application Code

Rest of Sequential CPU Code

GPU

Use GPU to Parallelize

Compute-Intensive Functions

CPU
3 WAYS TO ACCELERATE APPLICATIONS

- Applications
  - Libraries
    - “Drop-in” Acceleration
  - OpenACC Directives
    - Easily Accelerate Applications
  - Programming Languages
    - Maximum Performance
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
DROP-IN ACCELERATION WITH GPU LIBRARIES

Up to 10x speedups out of the box

Automatically scale with multi-GPU libraries

75% of developers use GPU libraries to accelerate their application
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
cuDNN  GPU-ACCELERATED DEEP LEARNING

High performance routines for Convolutional Neural Networks

- Optimized for current and future NVIDIA GPUs
- Integrated in major open-source frameworks
  - Caffe, Torch7, Theano
- Flexible and easy-to-use API
- Also available for ARM / Jetson TK1

Baseline Caffe compared to Caffe accelerated by cuDNN on K40

- Caffe (GPU) 11x
- Caffe (cuDNN) 14x

Baseline Caffe compared to Caffe accelerated by cuDNN on K40

*CPU is 24 core E5-2697v2 @ 2.4GHz Intel MKL 11.1.3
3 STEPS TO CUDA-ACCELERATED APPLICATION

Step 1: Substitute library calls with equivalent CUDA library calls

```
saxpy ( ... ) ➔ cudblasSaxpy ( ... )
```

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

```c
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*d_x[] + d_y[]
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);

DROP-IN ACCELERATION (STEP 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();
DROP-IN ACCELERATION (STEP 3)

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();

MOVE IN ACCELERATION (STEP 4)
Transfer data to GPU
Read data back GPU
5X-10X SPEEDUP USING NVIDIA LIBRARIES

BLAS | LAPACK | SPARSE | FFT | Math | Deep Learning | Image Processing

19

[Graphs and charts showing performance improvements for different libraries and operations.]
Linear Performance Scaling with XT libraries

- **cuBLAS-XT**
  Machine learning, O&G, Material Science, Defense, Supercomputing

- **cuFFT-XT**
  O&G, Molecular Dynamics, Defense

- **AmgX**
  CFD, Supercomputing, O&G Reservoir Sim
CUSOLVER (FROM CUDA 7)

- **cusolverDN**
  - Dense Cholesky, LU, SVD, (batched) QR
  - Optimization, Computer vision, CFD

- **cusolverSP**
  - Sparse direct solvers & Eigensolvers
  - Newton’s method, Chemical kinetics

- **cusolverRF**
  - Sparse refactorization solver
  - Chemistry, ODEs, Circuit simulation

---

**cusolverDN Speedup over CPU**

- SPOTRF
- DPOTRF
- CPOTRF
- ZPOTRF

**cusolverSP Speedup over CPU**

- 1138_bus
- ex9
- Chem97ZtZ
- nasa1824
- Muu

---

cuSOLVER 7.0, MKL 11.0.4, SuiteSparse 3.6.0  
K40, i7-3930K CPU @ 3.20GHz
3 WAYS TO ACCELERATE APPLICATIONS

Applications

Libraries

OpenACC Directives

Easily Accelerate Applications

Programming Languages

“Drop-in” Acceleration

Maximum Flexibility
OPENACC DIRECTIVES

Simple Compiler hints

Compiler Parallelizes code

Works on many-core GPUs & multicore CPUs

Your original Fortran or C code

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

NVIDIA  CRAY  PGI  CAPS
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:

```c
!$acc data copy(util1,util2,util3) copyin(ip,scp2,scp2i)
    !$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
Directives: Easy & Powerful

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.

-- Developer at the Global Manufacturer of Navigation Systems
Exploit GPU with LESS effort; maintain ONE legacy source code

Examples: REAL-WORLD application tuning using directives

ELAN
Computational Electro-Magnetics
- Goals: optimize w/ less effort, preserve code base
- Kernels 6.5X to 13X faster than 16-core Xeon
- Overall speedup 3.2X

COSMO
Weather
- Goal: preserve physics code (22% of runtime), augmenting dynamics kernels in CUDA
- Physics speedup 4.2X vs. multi-core Xeon

GAMESS
CCSD(T)
Molecular Modeling
- Goals: 3X speedup (2 kernels = 98% of runtime); scale to 1536 nodes
- Overall speedup 3.1X vs. 8-core Interlagos

Results from EMGS, MeteoSwiss/CSCS, NCSA/Cray/NVIDIA
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

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

```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.

```cpp
template <typename T>
struct Functor {
  __device__ Functor(T 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
}
```

// 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

```
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

```
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, Numba Pro

C# .NET
- GPU.NET

Numerical Analytics
- MATLAB
- Wolfram Mathematica 8
NVIDIA CUDA EVOLUTION
CUDA EVOLUTION (2)

CUDA 6
April-2014
- Unified Memory
  Simpler Programming & Memory Model
- Multi-GPU aware libraries
  Automatic Scaling to >1 GPU per node
  Operate directly on large datasets that reside in CPU memory
- Drop-in FFTW and BLAS libraries
  Accelerate FFT and BLAS with no code changes

CUDA 6.5
Q3 2014
- CUDA FORTRAN Tools support
- cuFFT Callbacks
  Improves performance
- Better Error detection and Reporting
  XID 13
- RDMA/P2P Topology viewer
- TESLA

CUDA 7
Q1 2015
- Power8 Support
- C++11
- CUDA C JIT
  Compile CUDA Kernels at run-time
- PGI Supported as host C++ Compiler
- Hyper-Q for Multi GPU support
- TESLA

CUDA tools for Hyper-Q/MPI
GPUDirect RDMA & OpenMPI Optimizations
  Reduce inter-node latency
  Improvements for MPI Application Scaling
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
\( \alpha \): scalar

GPU SAXPY in multiple languages and libraries

A menagerie* of possibilities, not a tutorial

*technically, a program chrestomathy: http://en.wikipedia.org/wiki/Chrestomathy
OPENACC COMPILER DIRECTIVES

Parallel C Code

```c
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

```fortran
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

```c
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

```c
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
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;
cudadeviceCopy(d_x, x, N, cudaMemcpyHostToDevice);
cudadeviceCopy(d_y, y, N, cudaMemcpyHostToDevice);

// Perform SAXPY on 1M elements
saxpy<<<4096,256>>>(N, 2.0, d_x, d_y);
cudadeviceCopy(y, d_y, N, cudaMemcpyDeviceToHost);

Serial C++ Code

```cpp
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

```cpp
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**

```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**

```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**

```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**

```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://numpy.scipy.org

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.

CUDA C, C++, Fortran

New Language Support

LLVM Compiler For CUDA

NVIDIA GPUs

New Processor Support

x86 CPUs
Further Improvements
UNIFIED MEMORY

Dramatically Lower Developer Effort

Developer View Today

- System Memory
- GPU Memory

Developer View With Unified Memory

- 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

```c
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

```c
void sortfile(FILE *fp, int N) {
    char *data;
    cudaMallocManaged(&data, N);
    fread(data, 1, N, fp);
    qsort<<<...>>>(data,N,1,compare);
    cudaMemcpy(data, &data, N, cudaMemcpyDeviceToHost);
    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
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

N3960 Technical Specification Working Draft:

Prototype:
https://github.com/n3554/n3554
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."
NUMBA PYTHON COMPILER

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

```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
Approach: apply a closure to a set of arrays

```java
// 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;
      }
    });
```

foreach iterations parallelized over GPU threads

- Threads run closure `execute()` method
What is MATLAB?

- High level language and development environment for:
  - Algorithm and application development
  - Data analysis
  - Mathematical modeling
  - GPU computing *
- Extensive math, engineering, and plotting functionality
- Add-on products for image and video processing, communications, signal processing, financial modeling, and more
- Over 1.3 million users worldwide

* Requires Parallel Computing Toolbox
Running MATLAB code on the GPU

- 200+ built-in MATLAB functions that are supported on the GPU
  - Random number generation
  - FFT
  - Matrix multiplications
  - Solvers
  - Convolutions
  - Min/max
  - SVD
  - Cholesky and LU factorization

- Additional support in toolboxes
  - Image Processing
    - Morphological filtering, 2-D filtering, ...
  - Communications
    - Turbo, LDPC, and Viterbi decoders, ...
  - Signal Processing
    - Cross correlation, FIR filtering, ...

- Use `arrayfun` to execute custom functions on the GPU (more efficiently than doing each operation within function individually)
Resources
ON-LINE

• nvidia.com/cuda
• developer.nvidia.com/cuda-zone
• MOOC on Udacity.com “Intro to Parallel Programming”
• Lots of other online courses from Universities
• Lots of forums and portals e.g. gpgpu.org, StackOverflow …
MANUALS

In order of complexity

• CUDA By Examples, Addison-Wesley/Pearson
• Programming Massively Parallel Processors, 2nd ed., Morgan Kaufmann
• CUDA Application Design & Development, Morgan Kaufmann
• CUDA Fortran
• The CUDA Handbook, Addison-Wesley/Pearson