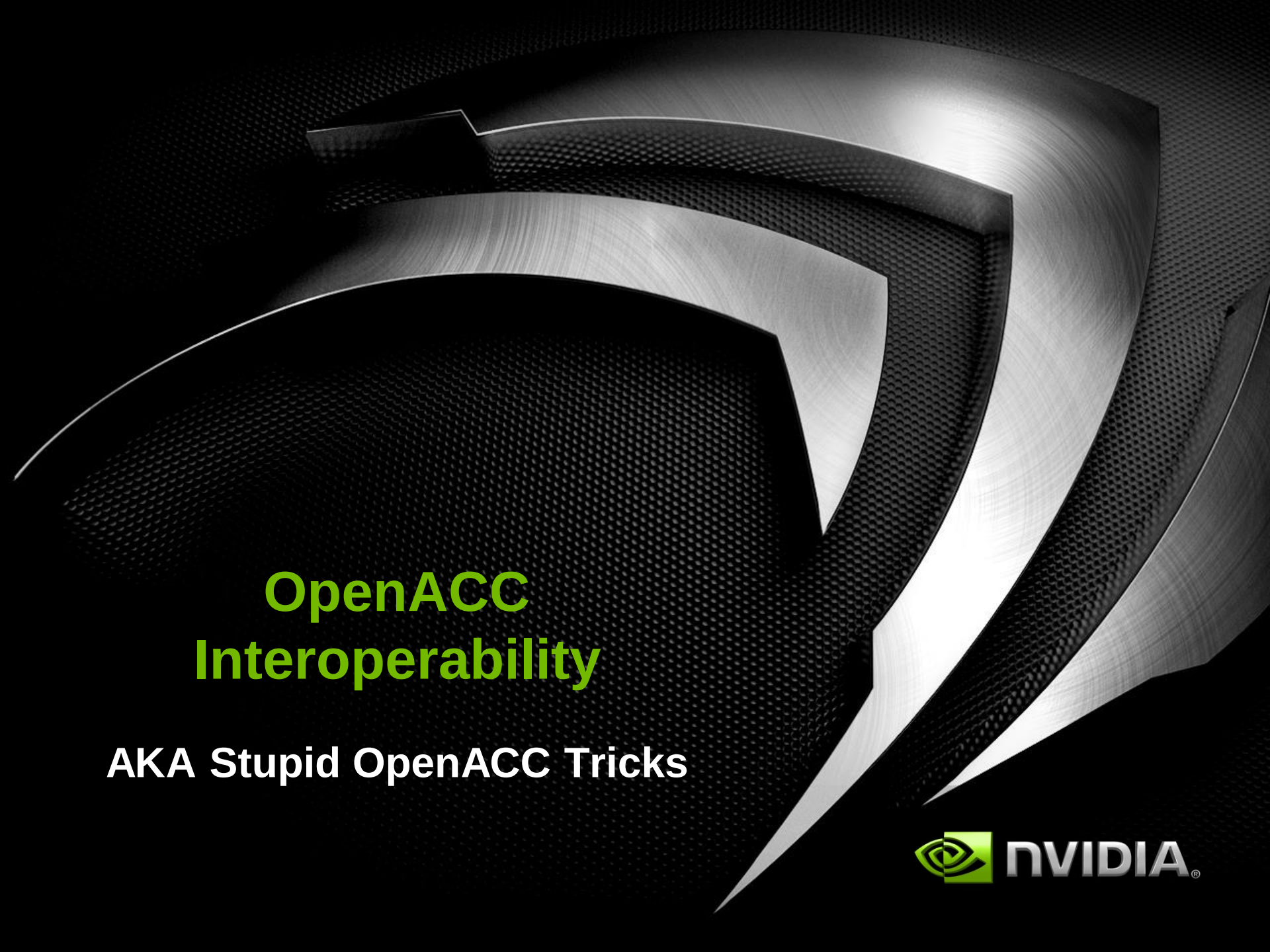

OpenACC
Interoperability
AKA Stupid OpenACC Tricks
NVIDIA

OpenACC is not an Island

- OpenACC allows very high level expression of parallelism and data movement.
- It's still possible to leverage low-level approaches such as CUDA C, CUDA Fortran, and GPU Libraries.

Why Interoperate?

- Don’t reinvent the wheel
  - Lots of CUDA code and libraries already exist and can be leveraged.
- Maximum Flexibility
  - Some things can just be represented more easily in one approach or another.
- Maximum Performance
  - Sometimes hand-tuning achieves the most performance.

CUDA C Primer

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);
```

- Serial loop over 1M elements, executes 1M times sequentially.
- Data is resident on CPU.

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);
```

- Parallel kernel, executes 1M times in parallel in groups of 256 elements.
- Data must be copied to/from GPU.

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

CUDA C Interoperability

OpenACC Main

```
program main
  integer, parameter :: N = 2**20
  real, dimension(N) :: X, Y
  real               :: A = 2.0

  !$acc data
  ! Initialize X and Y
  ...

  !$acc host_data use_device(x,y)
  call saxpy(n, a, x, y)
  !$acc end host_data
  !$acc end data

end program
```

CUDA C Kernel & Wrapper

```
__global__
void saxpy_kernel(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];
}

void saxpy(int n, float a, float *dx, float *dy)
{
  // Launch CUDA Kernel
  saxpy_kernel<<<4096,256>>>(N, 2.0, dx, dy);
}
```

- It's possible to interoperate from C/C++ or Fortran.
- OpenACC manages the data and passes device pointers to CUDA.

- CUDA kernel launch wrapped in function expecting device arrays.
- Kernel is launch with arrays passed from OpenACC in main.

CUDA C Interoperability (Reversed)

OpenACC Kernels

```
void saxpy(int n, float a, float *
restrict x, float * restrict y)
{
  #pragma acc kernels
deviceptr(x[0:n],y[0:n])
  {
    for(int i=0; i<n; i++)
    {
      y[i] += 2.0*x[i];
    }
  }
}
```

By passing a device pointer to an OpenACC region, it's possible to add OpenACC to an existing CUDA code.

CUDA C Main

```
int main(int argc, char **argv)
{
  float *x, *y, tmp;
  int n = 1<<20, i;

  cudaMalloc((void*)&x,(size_t)n*sizeof(float));
  cudaMalloc((void*)&y,(size_t)n*sizeof(float));

  ...

  saxpy(n, 2.0, x, y);
  cudaMemcpy(&tmp,y,(size_t)sizeof(float),
             cudaMemcpyDeviceToHost);
  return 0;
}
```

Memory is managed via standard CUDA calls.

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
```

- Serial loop over 1M elements, executes 1M times sequentially.
- Data is resident on CPU.

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
```

- Parallel kernel, executes 1M times in parallel in groups of 256 elements.
- Data must be copied to/from GPU (implicit).

http://developer.nvidia.com/cuda-fortran

CUDA Fortran Interoperability

OpenACC Main

```
program main
  use mymodule
  integer, parameter :: N =
2**20
  real, dimension(N) :: X, Y

  X(:) = 1.0
  Y(:) = 0.0

  !$acc data copy(y) copyin(x)
  call saxpy(N, 2.0, x, y)
  !$acc end data

end program
```

CUDA Fortran Kernel & Launcher

```
module mymodule
  contains
  attributes(global) &
  subroutine saxpy_kernel(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_kernel
  subroutine saxpy (n, a, x, y)
    use cudafor
    real, device :: x(:), y(:)
    real :: a
    integer :: n
    call saxpy_kernel<<<4096,256>>>(n, a, x, y)
  end subroutine saxpy
end module mymodule
```

- Thanks to the “device” attribute in saxpy, no host_data is needed.
- OpenACC manages the data and passes device pointers to CUDA.

- CUDA kernel launch wrapped in function expecting device arrays.
- Kernel is launch with arrays passed from OpenACC in main.

OpenACC with CUDA Fortran Main

Using the “deviceptr” data clause makes it possible to integrate OpenACC into an existing CUDA application.

CUDA C takes a few more tricks to compile, but can be done.

In theory, it should be possible to do the same with C/C++ (including Thrust), but in practice compiler incompatibilities make this difficult.

CUDA Fortran Main w/ OpenAcc Region

```
program main
  use cudafor
  integer, parameter :: N = 2**20
  real, device, dimension(N) :: X, Y
  integer :: i
  real :: tmp

  X(:) = 1.0
  Y(:) = 0.0

  !$acc kernels deviceptr(x,y)
  y(:) = y(:) + 2.0*x(:)
  !$acc end kernels

  tmp = y(1)
  print *, tmp
end program
```

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

CUBLAS Library & OpenACC

OpenACC can interface with existing GPU-optimized libraries (from C/C++ or Fortran).

This includes…

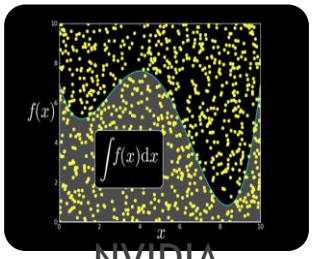
- CUBLAS
- Libsci_acc
- CUFFT
- MAGMA
- CULA
- …

OpenACC Main Calling CUBLAS

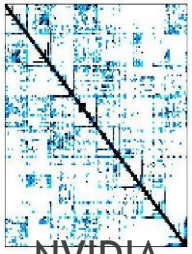
```
int N = 1<<20;
float *x, *y
// Allocate & Initialize X & Y
...

cublasInit();

#pragma acc data copyin(x[0:N]) copy(y[0:N])
{
  #pragma acc host_data use_device(x,y)
  {
    // Perform SAXPY on 1M elements
    cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);
  }
}

cublasShutdown();
```

Some GPU-accelerated Libraries

NVIDIA cuBLAS



NVIDIA cuRAND



NVIDIA cuSPARSE

NVIDIA NPP

GPU VSIPL

Vector Signal Image Processing

CULA tools

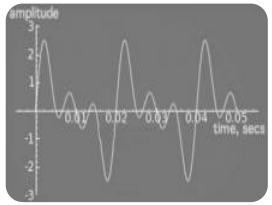
GPU Accelerated Linear Algebra

MAGMA

ICL UT

Matrix Algebra on GPU and Multicore

open source initiative



NVIDIA cuFFT

ROGUE WAVE SOFTWARE

IMSL Library

ArrayFire Matrix Computations

CUSP

Sparse Linear Algebra

open source initiative

Thrust

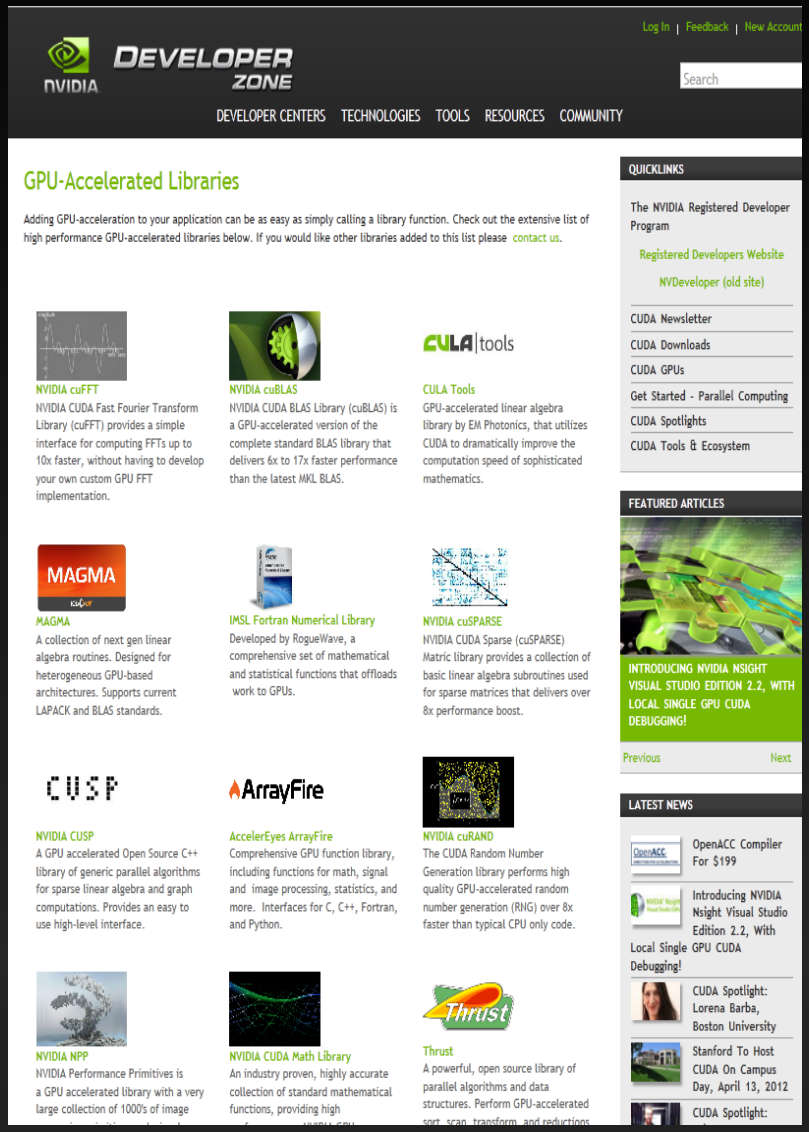
C++ STL Features for CUDA

open source initiative

Explore the CUDA (Libraries) Ecosystem

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

developer.nvidia.com/cuda-tools-ecosystem



Thrust C++ Template Library

Serial C++ Code with STL and Boost

```
int N = 1<<20;
std::vector<float> x(N), y(N);

...



// Perform SAXPY on 1M elements
std::transform(x.begin(), x.end(),
               y.begin(), y.end(),
                2.0f * _1 + _2);
```

www.boost.org/libs/lambda

Parallel C++ Code

```
int N = 1<<20;
thrust::host_vector<float> x(N), y(N);

...

thrust::device_vector<float> d_x = x;
thrust::device_vector<float> d_y = y;


// Perform SAXPY on 1M elements
thrust::transform(d_x.begin(), d_x.end(),
                  d_y.begin(),
d_y.begin(),
                  2.0f * _1 + _2);
```

http://thrust.github.com

Thrust C++ and OpenACC??

OpenACC Saxpy

```
void saxpy(int n, float a, float *
restrict x, float * restrict y)
{
#pragma acc kernels
deviceptr(x[0:n],y[0:n])
  {
    for(int i=0; i<n; i++)
    {
      y[i] += 2.0*x[i];
    }
  }
}
```

Thrust Main

```
int main(int argc, char **argv)
{
  int N = 1<<20;
  thrust::host_vector<float> x(N), y(N);
  for(int i=0; i<N; i++)
  {
    x[i] = 1.0f;
    y[i] = 1.0f;
  }

  thrust::device_vector<float> d_x = x;
  thrust::device_vector<float> d_y = y;
  thrust::device_ptr<float> p_x = &d_x[0];
  thrust::device_ptr<float> p_y = &d_y[0];

  saxpy(N,2.0,p_x.get(),p_y.get());

  y = d_y;
  return 0;
}
```

How to play well with others

My advice is to do the following:

1. Start with OpenACC
   - Expose high-level parallelism
   - Ensure correctness
   - Optimize away data movement last
2. Leverage other work that's available (even if it's not OpenACC)
   - Common libraries (good software engineering practice)
   - Lots of CUDA already exists
3. Share your experiences
   - OpenACC is still very new, best practices are still forming.
   - Allow others to leverage your work.