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

Application Code

Rest of Sequential CPU Code

GPU

Compute-Intensive Functions
Use GPU to Parallelize

CPU
3 WAYS TO ACCELERATE 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
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
```
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)

```c
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)

```c
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();
```

- Initialize CUBLAS
- Shut down CUBLAS
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();
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
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 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
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:

```bash
go90 -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
end do

$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
<table>
<thead>
<tr>
<th>Language</th>
<th>Tools</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>Thrust, CUDA C++</td>
</tr>
<tr>
<td>Python</td>
<td>PyCUDA, Copperhead</td>
</tr>
<tr>
<td>C#</td>
<td>GPU.NET</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++ 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());

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

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, 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++

CUDA Fortran

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

Thrust C++ Template Library
http://developer.nvidia.com/thrust

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

GPU.NET
http://tidepowerd.com

Mathematica
An Example:
6 Ways to SAXPY
SINGLE PRECISION ALPHA X PLUS Y (SAXPY)

Part of Basic Linear Algebra Subroutines (BLAS) Library

\[ z = \alpha x + y \]

\[ x, y, z : \text{vector} \]
\[ \alpha : \text{scalar} \]

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)
...```

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
**Standard C**

```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**

```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);
```

Serial C++ Code with STL and Boost

```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

```
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
    do i=1,n
      y(i) = a*x(i)+y(i)
    enddo
  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
```

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)

---

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)
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
Software Roadmap
UNIFIED MEMORY

Dramatically Lower Developer Effort

Developer View Today

Developer View With Unified Memory

System Memory

GPU 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
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);
}

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

```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.
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

- Complete set of parallel primitives: for_each, sort, reduce, scan, etc.

- ISO C++ committee voted unanimously to accept as official tech. specification working draft

```cpp
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);
```

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. ”

Oscar Hernandez
Oak Ridge National Laboratories
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
**Compile Java for GPUs**

- 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< joggingContext>() {
        public float execute(float x, float y) {
            return x + y;
        }
    });
```

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

---

Java Black-Scholes Options Pricing Speedup

[Graph showing speedup vs. Sequential Java]
THANKS!

Carlo Nardone
+39 335 5828197
cnardone@nvidia.com