INTRODUCTION TO GPU COMPUTING

Jeff Larkin, June 28, 2018
THE WORLD LEADER IN VISUAL COMPUTING
HPC's Biggest Challenge: Power
40 Years of Microprocessor Trend Data

Single-threaded perf: 1.5X per year

GPU-Computing perf: 1.1X per year

1000X by 2025
CUDA ECOSYSTEM 2018

CUDA DOWNLOADS IN 2017
3,500,000

CUDA REGISTERED DEVELOPERS
800,000

GTC ATTENDEES
8,000+
CUDA APPLICATION ECOSYSTEM
From Ease of Use to Specialized Performance

Applications  Frameworks  Libraries  Directives and Standard Languages  Specialized Languages

CUDA-C++  CUDA Fortran
FUNDAMENTALS OF GPU COMPUTING
ACCELERATED COMPUTING
10X PERFORMANCE & 5X ENERGY EFFICIENCY FOR HPC

CPU
Optimized for Serial Tasks

GPU Accelerator
Optimized for Parallel Tasks
ACCELERATED COMPUTING
10X PERFORMANCE & 5X ENERGY EFFICIENCY FOR HPC

**CPU**
Optimized for Serial Tasks

**CPU Strengths**
- Very large main memory
- Very fast clock speeds
- Latency optimized via large caches
- Small number of threads can run very quickly

**CPU Weaknesses**
- Relatively low memory bandwidth
- Cache misses very costly
- Low performance/watt
ACCELERATED COMPUTING

10X PERFORMANCE & 5X ENERGY EFFICIENCY FOR HPC

GPU Strengths
- High bandwidth main memory
- Latency tolerant via parallelism
- Significantly more compute resources
- High throughput
- High performance/watt

GPU Weaknesses
- Relatively low memory capacity
- Low per-thread performance
ACCELERATED COMPUTING
10X PERFORMANCE & 5X ENERGY EFFICIENCY FOR HPC

CPU
Optimized for Serial Tasks

GPU Accelerator
Optimized for Parallel Tasks
Speed v. Throughput

Which is better depends on your needs...

*Images from Wikimedia Commons via Creative Commons
Accelerator Nodes

CPU and GPU have distinct memories
- CPU generally larger and slower
- GPU generally smaller and faster

Execution begins on the CPU
- Data and computation are offloaded to the GPU

CPU and GPU communicate via PCIe
- Data must be copied between these memories over PCIe
- PCIe Bandwidth is much lower than either memories
HOW GPU ACCELERATION WORKS

Application Code

Compute-Intensive Functions

Small % of Code

Rest of Sequential CPU Code

GPU

CPU
3 WAYS TO PROGRAM GPUS

Applications

Libraries
“Drop-in” Acceleration

OpenACC Directives
Easily Accelerate Applications

Programming Languages
Maximum Flexibility
SIMPLICITY & PERFORMANCE

Accelerated Libraries
- Little or no code change for standard libraries; high performance
- Limited by what libraries are available

Compiler Directives
- High Level: Based on existing languages; simple and familiar
- High Level: Less control over performance

Parallel Language Extensions
- Expose low-level details for maximum performance
- Often more difficult to learn and more time consuming to implement
CODE FOR SIMPLICITY & PERFORMANCE

Libraries
• Implement as much as possible using portable libraries.

Directives
• Use directives to rapidly accelerate your code.

Languages
• Use lower level languages for important kernels.
GPU LIBRARIES
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
GPU ACCELERATED LIBRARIES
“Drop-in” Acceleration for Your Applications

Linear Algebra
FFT, BLAS, SPARSE, Matrix

Numerical & Math
RAND, Statistics

Data Struct. & AI
Sort, Scan, Zero Sum

Visual Processing
Image & Video
int N = 1 << 20; // 1M elements
x = (float *)malloc(N * sizeof(float));
y = (float *)malloc(N * sizeof(float));
initData(x, y);

// Perform SAXPY on 1M elements: y[] = a*x[] + y[]
saxpy(N, 2.0, x, 1, y, 1);
useResult(y);
DROP-IN ACCELERATION
With Automatic Data Management

int N = 1 << 20;  // 1M elements
x = (float *)malloc(N * sizeof(float));
y = (float *)malloc(N * sizeof(float));
initData(x, y);

// Perform SAXPY on 1M elements: y[] = a*x[] + y[]
saxpy(N, 2.0, x, 1, y, 1);
useResult(y);

int N = 1 << 20;  // 1M elements
cudaMallocManaged(&x, N * sizeof(float));
cudaMallocManaged(&y, N * sizeof(float));
initData(x, y);

// Perform SAXPY on 1M elements: y[] = a*x[] + y[]
saxpy(N, 2.0, x, 1, y, 1);
useResult(y);

**Step 1: Update memory allocation to be CUDA-aware**

Here, we use Unified Memory which automatically migrates between host (CPU) and device (GPU) as needed by the program.
DROP-IN ACCELERATION

With Automatic Data Management

Step 2: Call CUDA library version of API

Many standard libraries (BLAS, FFT, etc) have well-defined interfaces. CUDA will try to match interfaces as far as possible.
int N = 1 << 20; // 1M elements
x = (float *)malloc(N * sizeof(float));
y = (float *)malloc(N * sizeof(float));
initData(x, y);

// Perform SAXPY on 1M elements: y[] = a*x[] + y[]
saxpy(N, 2.0, x, 1, y, 1);
useResult(y);

Step 3: Manage Data Locality
If not using unified memory, the program moves the data up to the GPU and back.

int N = 1 << 20; // 1M elements
x = (float *)malloc(N * sizeof(float));
y = (float *)malloc(N * sizeof(float));
cudaMalloc(&d_x, N * sizeof(float));
cudaMalloc(&d_y, N * sizeof(float));
initData(x, 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: y[] = a*x[] + y[]
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);

// Bring the result back to the CPU
useResult(y);
GPU-accelerated Libraries for Computing

NVIDIA GPU-accelerated libraries provide highly-optimized functions that perform 2x-10x faster than CPU-only alternatives. Using drop-in interfaces, you can replace CPU-only libraries such as MKL, IPP and FFTW with GPU-accelerated versions with almost no code changes. The libraries can optimally scale your application across multiple GPUs.

With NVIDIA’s libraries, you get highly efficient implementations of algorithms that are regularly extended and optimized. Whether you are building a new application or trying to speed up an existing application, NVIDIA’s libraries provide the easiest way to get started with GPUs. You can download NVIDIA libraries as part of the CUDA Toolkit.

Download Now

COMPONENTS

Deep Learning
Signal, Image and Video

Linear Algebra
Parallel Algorithms
OPENACC DIRECTIVES
OpenACC is a directives-based programming approach to parallel computing designed for performance and portability on CPUs and GPUs for HPC.

```c
main()
{
    <serial code>
    #pragma acc kernels
    {
        <parallel code>
    }
}
```
OpenACC
Simple | Powerful | Portable
Fueling the Next Wave of Scientific Discoveries in HPC

main()
{
    <serial code>
    // automatically runs on GPU
    #pragma acc kernels
    {
        <parallel code>
    }
}

University of Illinois
PowerGrid - MRI Reconstruction

70x Speed-Up
2 Days of Effort

RIKEN Japan
NICAM - Climate Modeling

7-8x Speed-Up
5% of Code Modified

8000+
Developers
using OpenACC

http://www.cray.com/sites/default/files/resources/OpenACC_213462.12_OpenACC_Cosmo_CS_FNL.pdf
http://www.openacc.org/content/experiences-porting-molecular-dynamics-code-gpus-cray-xk7
SINGLE CODE FOR MULTIPLE PLATFORMS
OpenACC - Performance Portable Programming Model for HPC

POWER
Sunway
x86 CPU
x86 Xeon Phi
NVIDIA GPU
PEZY-SC

AWE Hydrodynamics CloverLeaf mini-App, bm32 data set

Systems: Haswell: 2x16 core Haswell server, four K80s, CentOS 7.2 (perf-hsw10), Broadwell: 2x20 core Broadwell server, eight P100s (dgx1-prd-01), Minsky: POWER8+NVLINK, four P100s, RHEL 7.3 (gsn1).
Benchmark: CloverLeaf v1.3 downloaded from http://uk-mac.github.io/CloverLeaf the week of November 7 2016; CloverLeaf_Serial; CloverLeaf_ref (MPI+OpenMP); CloverLeaf_OpenACC (MPI+OpenACC)
Data compiled by PGI November 2016, Volta data collected June 2017
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)
...
```

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);

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

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
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

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
  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

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)
```

**Numba Parallel Python**

```python
import numpy as np
from numba import vectorize

@vectorize(['float32(float32, float32, float32)'], target='cuda')
def saxpy(a, x, y):
    return a * x + y

N = 1048576

# Initialize arrays
A = np.ones(N, dtype=np.float32)
B = np.ones(A.shape, dtype=A.dtype)
C = np.empty_like(A, dtype=A.dtype)

# Add arrays onGPU
C = saxpy(2.0, X, Y)
```

[http://numpy.scipy.org](http://numpy.scipy.org)  [https://numba.pydata.org](https://numba.pydata.org)
ENABLING ENDLESS WAYS TO SAXPY

• Build front-ends for Java, Python, R, DSLs
• Target other processors like ARM, FPGA, GPUs, x86

CUDA Compiler Contributed to Open Source LLVM
CUDA TOOLKIT - DOWNLOAD TODAY!
Everything You Need to Accelerate Applications

CUDA DOCUMENTATION
- Installation Guide
- Programming Guide
- API Reference

GETTING STARTED RESOURCES
- Best Practices Guide
- CUDA Tools Guide
- Samples

INDUSTRY APPLICATIONS

developer.nvidia.com/cuda-toolkit