Outline

- Evolution of GPU Computing
- Parallelism
- Heterogeneous Computing Concepts
- Using GPUs to Accelerate Applications
  - Accelerated Libraries
  - Compiler Directives
  - Programming Languages
Once upon a time . . .

- **GPU**: Graphical Processing Unit
  - Originated as specialized hardware for 3D games.

- **Why a different processor?**
  - Rendering is the most computationally intense part of a game.
  - CPU is not an ideal device for computer graphics rendering
  - Freed CPU allows more complex AI, dynamic world generation, realistic dynamics.
Programmable GPUs

• Graphics research diverged from OpenGL pipeline.

• 2001 - Programmable vertex and fragment shaders. (Geforce3)
  • OpenGL 2.0 (2004) updates pipeline
  • GPGPU era begins
GPGPU: “Because It’s There”

Warning: Nerdy Movie Reference
GPU Computing Era: G80 and Fermi

G80
- Unified Shader Architecture
- Double Precision
- CUDA Introduced

Fermi (GF100)
- ECC
- Enhanced Double precision
- Memory hierarchy and expanded caching
12 years later

- **NVIDIA Kepler**
  - 1.31 tflop double precision
  - 3.95 tflop single precision
  - 250 gb/sec memory bandwidth
  - 2,688 Functional Units (cores)

≈ #1 on Top500 in 1997

- Can still play video games

NVIDIA GeForce Titan

NVIDIA GK110 - Kepler
Not A Video Game

18,866 NVIDIA GK110 + AMD Interlagos
Rpeak: 27,112.5 TFLOPS

Top500 Rank: #1 Nov’12
Science Uses GPUs

Medical Imaging
U of Utah

Molecular Dynamics
U of IL, 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
Outline

- Evolution of GPU Computing
- Parallelism
- Heterogeneous Computing Concepts
- Using GPUs to Accelerate Applications
  - Accelerated Libraries
  - Compiler Directives
  - Programming Languages
Latency vs. Throughput

**Throughput**
- Work per unit time
- Matrix operations, FFTs, signal processing, data analytics, rendering/visualization

**Latency**
- Time per unit work
- Databases, web servers, transaction servers, Internet infrastructure.
A GPU is a Throughput Optimized Processor

- GPU Achieves high throughput by parallel execution
  - 2,688 cores (GK110)
  - Millions of resident threads
- GPU Threads are much lighter weight than CPU threads like pThreads
- Processing in Parallel is how GPU achieves performance
A GPU is a Throughput Optimized Processor

- GPU Achieves high throughput by parallel execution
  - 2,688 cores (GK110)
  - Millions of resident threads
- GPU Threads are much lighter weight than CPU threads like pThreads
- Processing in parallel is how **HPC** achieves performance
Parallelism

- Parallel Poll
  - Ideal Serial Processor: IPC 1.0, 8ghz, perfect pipeline: 8gflops/core

Yellowstone: 1.5 Pflops @ 2.6ghz (#13 on Top 500 Nov’12!)
- 4,512 Nodes
- 2 Sockets per node
- 8 cores per socket
- Each Core: 2 x 256bit-wide pipelined FP units @ 2.6 ghz
  - Theoretical Peak: 20.8 gflops / core
Exposing Parallelism is . . .

. . . a necessary part of GPU Computing

. . . essential to utilize current HPC systems

. . . going to be more important on future systems
Outline

- Evolution of GPU Computing
- Parallelism
- Heterogeneous Computing Concepts
- Using GPUs to Accelerate Applications
  - Accelerated Libraries
  - Compiler Directives
  - Programming Languages
Heterogeneous Computing

- The CPU and its memory (host)
- The GPU and its memory (device)
```cpp
#include <iostream>
#include <algorithm>

using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];

    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    __syncthreads();

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS; offset <= RADIUS; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}

void fill_ints(int *x, int n) {
    fill_n(x, n, 1);
}

int main(void) {
    int *in, *out;
    // host copies of a, b, c
    int *d_in, *d_out;
    // device copies of a, b, c
    int size = (N + 2*RADIUS) * sizeof(int);
    // Alloc space for host copies and setup values
    in  = (int*)malloc(size); fill_ints(in,  N + 2*RADIUS);
    out = (int*)malloc(size); fill_ints(out, N + 2*RADIUS);
    // Alloc space for device copies
    cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

    // Launch stencil_1d() kernel on GPU
    stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS, d_out + RADIUS);

    // Copy result back to host
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(in); free(out); cudaFree(d_in); cudaFree(d_out);
    return 0;
}
```

**Heterogeneous Computing**

- **Parallel Code:** GPU-based stencil computation
- **Serial Code:** CPU-based data preparation and result collection
**GPU Computing**

- **GPU**
  - Compute-Intensive Functions
  - Use GPU to Parallelize

- **Application Code**

- **CPU**
  - Rest of Sequential CPU Code
Simple Processing Flow

1. Copy input data from CPU memory to GPU memory
Simple Processing Flow

1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute.
   Results stored in GPU memory.
Simple Processing Flow

1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute. Results stored in GPU memory.
3. Copy results from GPU memory to CPU memory
Outline

- Evolution of GPU Computing
- Parallelism
- Heterogeneous Computing Concepts
- Using GPUs to Accelerate Applications
  - Accelerated Libraries
  - Compiler Directives
  - Programming Languages
Simple Example: SAXPY

- BLAS1 function
- \( Y = a \times X + Y \)
- \( Y \) and \( X \) are length \( N \) vectors
- \( A \) is a scalar
- single precision data (float)

```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);
```
3 Ways to Accelerate Applications

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

Applications

Libraries

OpenACC Directives

Easily Accelerate Applications

Programming Languages

Maximum Flexibility

“Drop-in” Acceleration
GPU Accelerated Libraries

- NVIDIA cuBLAS
- NVIDIA cuRAND
- NVIDIA cuSPARSE
- NVIDIA NPP
- GPU VSIPL: Vector Signal Image Processing
- CULA tools: GPU Accelerated Linear Algebra
- MAGMA: Matrix Algebra on GPU and Multicore
- NVIDIA cuFFT
- Rogue Wave Software: IMSL Library
- ArrayFire: Matrix Computations
- CUSP: Sparse Linear Algebra
- Thrust: C++ STL Features for CUDA
SAXPY in cuBLAS

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
3 Ways to Accelerate Applications

Applications

Libraries

OpenACC Directives

Easily Accelerate Applications

Programming Languages

“Drop-in” Acceleration

Maximun Flexibility

Languages
OpenACC Compiler Directives

Simple Compiler hints

Compiler Parallelizes code

Works on many-core GPUs & multicore CPUs

Your original Fortran or C code

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

CPU

GPU
SAXPY with OpenACC 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)
...```

OpenACC Will be covered Friday

- Exposing Parallelism with OpenACC
- Profiling and Tuning OpenACC Applications
- Coding and profiling hands on

A Shameless Plug
3 Ways to Accelerate Applications

- Libraries
  - “Drop-in” Acceleration
- OpenACC Directives
  - Easily Accelerate Applications
- Programming Languages
  - Maximum Flexibility
CUDA C/C++

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…

CUDA C++ features enable sophisticated and flexible applications and middleware

```cpp
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
}
```

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
  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
```

CUDA C/C++ will be covered Thursday

- Optimization and GPU Architecture
- Hands on development, optimization, and profiling

YASP: Yet Another Shameless Plug
SAXPY in CUDA C

**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);
```

**CUDA 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;

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

cudaMemcpy(y, d_y, N, cudaMemcpyDeviceToHost);
```

SAXPY in 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
```

**CUDA Fortran**

```fortran
module mymodule contains
  subroutine saxpy(n, a, x, y)
    attributes(global) 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
```

Takeaways

- GPUs accelerate science
  - Increase throughput on parallel computations common to scientific applications.
  - GPUs have proven their worth in accelerating real science applications.

- Three ways to make use of GPUs in your application:
  1. Accelerated Libraries
  2. Compiler Directives – OpenACC in C, C++, Fortran
     - Tutorial - Friday
  3. Programming Languages- CUDA C/C++, CUDA Fortran
     - Tutorial – Thursday
Final Words:

- We live in a parallel universe
- Think in parallel about your problem in parallel
Conference Progress: 60%

Plenary Sessions: 100%