AGENDA

1 Intro to GPU Computing
2 OpenACC with hands-on
3 CUDA C/C++ with hands-on
4
5
Parallel programming—Why do you care?
The world IS parallel
Accelerator Programming—Why do you Care?
Power of 300 Petaflop CPU-only Supercomputer = Power for the city of San Francisco

HPC’s Biggest Challenge: Power
UNPRECEDENTED VALUE TO SCIENTIFIC COMPUTING

64 Sandy Bridge CPUs
58 ns/day

1 Tesla K40 GPU
102 ns/day

AMBER Molecular Dynamics Simulation
DHFR NVE Benchmark
3 WAYS TO ACCELERATE APPLICATIONS

Applications

Libraries
“Drop-in” Acceleration

OpenACC Directives
Easily Accelerate Applications

Programming Languages
Maximum Flexibility
GPU ACCELERATED LIBRARIES
“Drop-on” Acceleration for your Applications

Linear Algebra
FFT, BLAS, SPARSE, Matrix

NVIDIA cuFFT, cuBLAS, cuSPARSE

Numerical & Math
RAND, Statistics

NVIDIA Math Lib

ArrayFire

Data Struct. & AI
Sort, Scan, Zero Sum

Thrust

GPU AI
Board
Games

GPU AI
Path Finding

Visual Processing
Image & Video

NVIDIA NPP

NVIDIA cuRAND

NVIDIA Video
Encode

Sundog Software
3 WAYS TO ACCELERATE APPLICATIONS

Applications

Libraries
“Drop-in” Acceleration

OpenACC Directives
Easily Accelerate Applications

Programming Languages
Maximum Flexibility
OPENACC DIRECTIVES

Program myscience
... serial code ...

!$acc kernels
do k = 1,n1
do i = 1,n2
... parallel code ...
endo
doend
doincend kernels
End Program myscience

CPU

GPU

Simple Compiler hints

Compiler Parallelizes code

Works on many-core GPUs & multicore CPUs

Your original Fortran or C code
FAMILIAR TO OPENMP PROGRAMMERS

OpenMP

CPU

```c
main() {
    double pi = 0.0; long i;

    #pragma omp parallel for reduction(+:pi)
    for (i=0; i<N; i++)
    {
        double t = (double)((i+0.05)/N);
        pi += 4.0/(1.0+t*t);
    }

    printf("pi = %f\n", pi/N);
}
```

OpenACC

CPU

```c
main() {
    double pi = 0.0; long i;

    #pragma acc kernels
    for (i=0; i<N; i++)
    {
        double t = (double)((i+0.05)/N);
        pi += 4.0/(1.0+t*t);
    }

    printf("pi = %f\n", pi/N);
}
```

GPU
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"
**A VERY SIMPLE EXERCISE: SAXPY**

### SAXPY in C

```c
void saxpy(int n,
   float a,
   float *x,
   float *restrict 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);
...```

### SAXPY in Fortran

```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)
...```
GPU Architecture
GPU ARCHITECTURE

Two Main Components

- Global memory
  - Analogous to RAM in a CPU server
  - Accessible by both GPU and CPU
  - Currently up to 12 GB
  - ECC on/off options for Quadro and Tesla products

- Streaming Multiprocessors (SM)
  - Perform the actual computation
  - Each SM has its own: Control units, registers, execution pipelines, caches
GPU ARCHITECTURE

Streaming Multiprocessor (SM)

- Many CUDA Cores per SM
  - Architecture dependent
- Special-function units
  - \( \cos / \sin / \tan \), etc.
- Shared mem + L1 cache
- Thousands of 32-bit registers
GPU ARCHITECTURE

CUDA Core

- Floating point & Integer unit
  - IEEE 754-2008 floating-point standard
  - Fused multiply-add (FMA) instruction for both single and double precision
- Logic unit
- Move, compare unit
- Branch unit
GPU MEMORY HIERARCHY REVIEW

SM-0
Registers
L1
SMEM

SM-1
Registers
L1
SMEM

SM-N
Registers
L1
SMEM

L2

Global Memory
GPU ARCHITECTURE

Memory System on each SM

- Extremely fast, but small, i.e., 10s of Kb
- Programmer chooses whether to use cache as L1 or Shared Mem
  - L1
    - Hardware-managed—used for things like register spilling
    - Should NOT attempt to utilize like CPU caches
  - Shared Memory—programmer MUST synchronize data accesses!!!
    - User-managed scratch pad
    - Repeated access to same data or multiple threads with same data
GPU ARCHITECTURE

Memory system on each GPU board

- Unified L2 cache (100s of Kb)
  - Fast, coherent data sharing across all cores in the GPU
- ECC protection
  - DRAM
    - ECC supported for GDDR5 memory
  - All major internal memories are ECC protected
    - Register file, L1 cache, L2 cache
CUDA Programming model
ANATOMY OF A CUDA C/C++ APPLICATION

- **Serial** code executes in a **Host** (CPU) thread
- **Parallel** code executes in many **Device** (GPU) threads across multiple processing elements
CUDA C : C WITH A FEW KEYWORDS

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];
}
// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);

__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];
}
// Invoke parallel SAXPY kernel with 256 threads/block
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nbblocks, 256>>>(n, 2.0, x, y);
CUDA KERNELS

- Parallel portion of application: execute as a **kernel**
  - Entire GPU executes kernel, many threads
- CUDA threads:
  - Lightweight
  - Fast switching
  - 1000s execute simultaneously

<table>
<thead>
<tr>
<th>CPU</th>
<th>Host</th>
<th>Executes functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>Device</td>
<td>Executes kernels</td>
</tr>
</tbody>
</table>
CUDA KERNELS: PARALLEL THREADS

- A **kernel** is a function executed on the GPU as an array of threads in parallel.

- All threads execute the same code, can take different paths.

- Each thread has an ID:
  - Select input/output data
  - Control decisions

```c
float x = input[threadIdx.x];
float y = func(x);
output[threadIdx.x] = y;
```
CUDA Kernels: Subdivide into Blocks
CUDA Kernels: Subdivide into Blocks

Threads are grouped into blocks
CUDA Kernels: Subdivide into Blocks

- Threads are grouped into blocks
- Blocks are grouped into a grid
Threads are grouped into blocks

Blocks are grouped into a grid

A kernel is executed as a grid of blocks of threads
CUDA Kernels: Subdivide into Blocks

- Threads are grouped into blocks
- Blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads
Kernel Execution

- Each kernel is executed on one device
- Multiple kernels can execute on a device at one time

CUDA - enabled GPU

- Each thread is executed by a core

CUDA Streaming Multiprocessor

- Each block is executed by one SM and does not migrate
- Several concurrent blocks can reside on one SM depending on the blocks’ memory requirements and the SM’s memory resources

CUDA-enabled GPU

- Each kernel is executed on one device
- Multiple kernels can execute on a device at one time
Thread blocks allow cooperation

- Threads may need to cooperate:
  - Cooperatively load/store blocks of memory all will use
  - Share results with each other or cooperate to produce a single result
  - Synchronize with each other
THREAD BLOCKS ALLOW SCALABILITY

- Blocks can execute in any order, concurrently or sequentially
- This independence between blocks gives scalability:
  - A kernel scales across any number of SMs
Memory System Hierarchy
Thread:
  - Registers
Thread:
- Registers
- Local memory
MEMORY HIERARCHY

- Thread:
  - Registers
  - Local memory

- Block of threads:
  - Shared memory
MEMORY HIERARCHY : SHARED MEMORY

__shared__ int a[SIZE];

- Allocated per thread block, same lifetime as the block
- Accessible by any thread in the block
- Several uses:
  - Sharing data among threads in a block
  - User-managed cache (reducing gmem accesses)
MEMORY HIERARCHY

- Thread:
  - Registers
  - Local memory
- Block of threads:
  - Shared memory
- All blocks:
  - Global memory
MEMORY HIERARCHY : GLOBAL MEMORY

- Accessible by all threads of any kernel
- Data lifetime: from allocation to deallocation by host code
  - cudaMalloc (void** pointer, size_t nbytes)
  - cudaMemcpy (void* pointer, int value, size_t count)
  - cudaMemcpy (void* pointer)
CUDA memory management
MEMORY SPACES

CPU and GPU have separate memory spaces

- Data is moved across PCIe bus
- Use functions to allocate/set/copy memory on GPU just like standard C

Pointers are just addresses

- Can’t tell from the pointer value whether the address is on CPU or GPU
  - Must use cudaPointerGetAttributes(...)
- Must exercise care when dereferencing:
  - Dereferencing CPU pointer on GPU will likely crash
  - Dereferencing GPU pointer on CPU will likely crash
GPU MEMORY ALLOCATION / RELEASE

Host (CPU) manages device (GPU) memory

- cudaMemcpy (void ** pointer, size_t nbytes)
- cudaMemcpy (void * pointer, int value, size_t count)
- cudaMemcpy (void* pointer)

int n = 1024;

int nbytes = 1024*sizeof(int);

int * d_a = 0;

cudaMalloc( (void**)d_a, nbytes );
cudaMemset( d_a, 0, nbytes);
cudaFree(d_a);

Note: Device memory from GPU point of view is also referred to as global memory.
DATA COPIES

cudaMemcpy( void *dst,  void *src,  size_t nbytes,  
enum cudaMemcpyKind direction);

- returns after the copy is complete
- blocks CPU thread until all bytes have been copied
- doesn’t start copying until previous CUDA calls complete

enum cudaMemcpyKind

- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToHost
- cudaMemcpyDeviceToDevice

Non-blocking memcopies are provided
Basic kernels and execution
CUDA PROGRAMMING MODEL REVISITED

- Parallel code (kernel) is launched and executed on a device by many threads
- Threads are grouped into thread blocks
- Parallel code is written for a thread
  - Each thread is free to execute a unique code path
  - Built-in thread and block ID variables
THREAD HIERARCHY

- Threads launched for a parallel section are partitioned into thread blocks
  - Grid = all blocks for a given launch
  - Thread block is a group of threads that can:
    - Synchronize their execution
    - Communicate via shared memory
IDS AND DIMENSIONS

 Threads
  - 3D IDs, unique within a block

 Blocks
  - 2D IDs, unique within a grid

 Dimensions set at launch time
  - Can be unique for each grid

 Built-in variables
  - threadIdx, blockIdx
  - blockDim, gridDim

(Continued)
IDS AND DIMENSIONS

Threads
- 3D IDs, unique within a block

Blocks
- 2D IDs, unique within a grid

Dimensions set at launch time
- Can be unique for each grid

Built-in variables
- threadIdx, blockIdx
- blockDim, gridDim
LAUNCHING KERNELS ON GPU

Launch parameters (triple chevron <<<>>> notation)

- grid dimensions (up to 2D), dim3 type
- thread-block dimensions (up to 3D), dim3 type
- shared memory: number of bytes per block
  - for extern smem variables declared without size
  - Optional, 0 by default
- stream ID
  - Optional, 0 by default

```cpp
dim3 grid(16, 16);
dim3 block(16,16);
kernel<<<grid, block, 0, 0>>>(...);
kernel<<<32, 512>>>(...);
```
GPU KERNEL EXECUTION

- Kernel launches on a grid of blocks, $<\text{grid}, \text{block}>$ (arg1,...)
- Each block is launched on one SM
  - A block is divided into warps of 32 threads each (think 32-way vector)
  - Warps in a block are scheduled and executed.
    - All threads in a warp execute same instruction simultaneously (think SIMD)
  - Number of blocks/SM determined by resources required by the block
    - Registers, shared memory, total warps, etc.
- Block runs to completion on SM it started on, no migration.
A thread block consists of 32-thread warps

A warp is executed physically in parallel (SIMD) on a multiprocessor
Threads are executed by scalar processors.

Thread blocks are executed on multiprocessors.

Thread blocks do not migrate.

Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file).

A kernel is launched as a grid of thread blocks.
BLOCKS MUST BE INDEPENDENT

Any possible interleaving of blocks should be valid
  ▶ presumed to run to completion without pre-emption
  ▶ can run in any order
  ▶ can run concurrently OR sequentially

Blocks may coordinate but not synchronize
  ▶ shared queue pointer: OK
  ▶ shared lock: BAD ... any dependence on order easily deadlocks

Independence requirement gives scalability