Introduction to GPU Computing

James Wynne
User Assistance & Outreach
National Center for Computational Sciences
Titan Node Layout

Host

Device

GDDR5 memory

Cray Gemini

HT3

HT3

AM0 Opteron

DDR3 memory
Intro into GPU Computing

• Monte Carlo Method for π
  – Since we know from geometry that the ratio of the square and circle is \( \pi/4 \)
  
  – Scatter a large number of random “darts” on it

  – Find the ratio of darts within the circle, and total number of darts scattered
    \[ \pi = 4 \left( \frac{\text{dartsInCircle}}{\text{totalDarts}} \right) \]
Intro into GPU Computing

- Monte Carlo Method for $\pi$
Intro into GPU Computing

• Monte Carlo Method for $\pi$
  – It's an “embarrassingly parallel” application

> Each point’s computations does not rely on any other point’s data

> I.E. each iteration of the main loop is independent
Seed the RNG

```cpp
seedRandomNumberGenerator();

for (i = 0; i <= numDarts; i++)
{
    x = rand();
    y = rand();
    z = ((x*x)+(y*y));
    if (z <= 1)
        count++;
}
pi = (count/numDarts)*4;
```
seedRandomNumberGenerator();
for (i = 0; i <= numDarts; i++)
{
    x = rand();
    y = rand();
    z = ((x*x)+(y*y));
    if (z <= 1)
        count++;
}
pi = (count/numDarts)*4;
seedRandomNumberGenerator();

for (i = 0; i <= numDarts; i++)
{
    x = rand();
    y = rand();
    z = ((x*x)+(y*y));
    if (z <= 1)
        count++;
}

pi = (count/numDarts)*4;
seedRandomNumberGenerator();
for (i = 0; i <= numDarts; i++)
{
    x = rand();
    y = rand();
    z = ((x*x)+(y*y));
    if (z <= 1)
        count++;
}
pi = (count/numDarts)*4;
Serial

```java
seedRandomNumberGenerator();
for (i = 0; i <= numDarts; i++)
    numDarts = maximum number of points
{
    x = rand();
    y = rand();
    z = ((x*x)+(y*y));
    if (z <= 1)
        count++;
}
pi = (count/numDarts)*4;
```
serial

```
seedRandomNumberGenerator();

for (i = 0; i <= numDarts; i++)
{
  x = rand();
  y = rand();
  z = ((x*x)+(y*y));
  if (z <= 1)
    count++;
}

pi = (count/numDarts)*4;
```

numDarts = maximum number of points
seedRandomNumberGenerator();
for (i = 0; i <= numDarts; i++)
{
    x = rand();  
    y = rand();  
    z = ((x*x)+(y*y));  
    if (z <= 1)
        count++;
}
pi = (count/numDarts)*4;
Serial

seedRandomNumberGenerator();
for (i = 0; i <= numDarts; i++)
{
    x = rand();
    y = rand();
    z = ((x*x)+(y*y)); // Determine if the point is inside the circle
    if (z <= 1)
        count++;
}
pi = (count/numDarts)*4;
Serial

seedRandomNumberGenerator();
for (i = 0; i <= numDarts; i++)
{
    x = rand();
    y = rand();
    z = ((x*x)+(y*y));
    if (z <= 1) // If it is, increment count
        count++;
}
pi = (count/numDarts)*4;
seedRandomNumberGenerator();
for (i = 0; i <= numDarts; i++)
{
    x = rand();
    y = rand();
    z = ((x*x)+(y*y));
    if (z <= 1)
        count++;
}
pi = (count/numDarts)*4;    Calculate π by finding the ratio * 4
Beginning Parallelization

• Profile your code
  – Use tools to look for what takes the longest/most computationally intense
Beginning Parallelization

• Profile your code
  – Many tools exist to profile your code
    » CrayPat-lite
      • CPU code

  » NVIDIA Visual Profiler
    • Used to profile GPU kernels
Beginning Parallelization

- CrayPat-lite

```
!/a/s/w/stf007 >>> aprun -n1 ./a.out
CrayPat/X: Version 6.1.3 Revision 12145 11/18/13 21:56:10
Pi: 3.140652

CRYPAT-LITE PERFORMANCE STATISTICS

CrayPat/X: Version 6.1.3 Revision 12145 (xf 12007) 11/18/13 21:56:10
Experiment: lite sample_profile
Number of PEs (MPI ranks): 1
Numbers of PEs per Node: 1
Numbers of Threads per PE: 1
Number of Cores per Socket: 16
Execution start time: Mon Feb 3 15:07:17 2014
System name and speed: titan-ext5 2200 MHz

Wall Clock Time: 0.109849 secs
High Memory: 18.89 MBytes
MFLOPS (aggregate): 105.24 M/sec
I/O Write Rate: 0.35 MBytes/Sec
```
## Beginning Parallelization

- CrayPat-lite

### Table 1: Profile by Function Group and Function (top 3 functions shown)

<table>
<thead>
<tr>
<th>Samp%</th>
<th>Samp</th>
<th>Group</th>
<th>Imb.</th>
<th>Imb.</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0%</td>
<td>5.0</td>
<td>Total</td>
<td>--</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>60.0%</td>
<td>3.0</td>
<td>MATH</td>
<td>--</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>40.0%</td>
<td>2.0</td>
<td>erand48_r</td>
<td>--</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>20.0%</td>
<td>1.0</td>
<td>drand48</td>
<td>--</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>40.0%</td>
<td>2.0</td>
<td>ETC</td>
<td>--</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>40.0%</td>
<td>2.0</td>
<td>_drand48_iterate</td>
<td>--</td>
<td>--</td>
<td></td>
</tr>
</tbody>
</table>
Beginning Parallelization

- NVIDIA Visual Profiler
Beginning Parallelization

• Analyze your code profile
  – Most computationally intensive section
  – Takes the longest

• Common structures
  – Loops
    » No data dependence between loop iterations
Common Challenges

• Memory allocation and transfer
  – Allocation of memory on the device
  – Copy data over to device memory, do computation, copy back to host memory

• Must be worth the effort/time to copy it to the device

• Various ways to accelerate code
  – OpenACC
  – CUDA
  – Thrust
Data Flow

• Identify the portion of code for acceleration

```
seedRandomNumberGenerator();

for (i = 0; i <= numDarts; i++)
{
    x = rand();
    y = rand();
    z = ((x*x)+(y*y));
    if (z <= 1)
        count++;
}

pi = (count/numDarts)*4;
```
Data Flow

- Copy this kernel to the device to be executed

```cpp
seedRandomNumberGenerator();

for (i = 0; i <= numDarts; i++)
{
    x = rand();
    y = rand();
    z = ((x*x)+(y*y));
    if (z <= 1)
        count++;
}

pi = (count/numDarts)*4;
```
Beginning Parallelization

• Once finished, copy the resulting data back to the host

```cpp
seedRandomNumberGenerator();
for (i = 0; i <= numDarts; i++)
{
    x = rand();
    y = rand();
    z = ((x*x)+(y*y));
    if (z <= 1)
        count++;
}
pi = (count/numDarts)*4;
```
Introduction to GPU Computing

OpenACC
OpenACC

• OpenACC is similar to OpenMP
  – Utilizes compiler directives (#pragma acc)
  – Enclosed section will be offload to the device

• Designed to be a quick and simple way to accelerate code
  – Higher level
  – Tends to hide explicit accelerator functions
    » Memory allocation/copy
    » Accelerator initialization
  – Removes need for programmer to manage separate variables for host and device data

• As of Feb. 2014, only implemented in few compilers
  – PGI
  – Cray
OpenACC

• Pre-fill an array on host with random numbers
  – Can not run functions inside kernels

• Identify the region(s) to accelerate with OpenACC

• Use the `copyin()` clause to copy `randomnums[]` onto the accelerator

• Use `create()` clause to create our private variables on the device
OpenACC

```c
seedRandomNumberGenerator();
for (int i = 0; i<2*niter; i++)
    randomnums[i] = rand();

#pragma acc parallel copyin(randomnums[0:200000]) create(x,y,z)
{
    for (i = 0; i<100000; i++)
        int temp = i+i;
        x = randomnums[temp];
        y = randomnums[temp+1];
        z = (x*x)+(y*y);
        output[i] = z;
    for(i = 0; i<100000; i++)
        if output[i]<=1
            ++count;
}```

Seeds and generates the random numbers for use on the GPU.
OpenACC

seedRandomNumberGenerator();
for (int i = 0; i<2*niter; i++)
    randomnums[i] = rand();

#pragma acc parallel copyin(randomnums[0:200000]) create(x,y,z)
{
    for (i = 0; i<100000; i++)
        int temp = i+i;
        x = randomnums[temp];
        y = randomnums[temp+1];
        z = (x*x)+(y*y);
        output[i] = z;
    for(i = 0; i<100000; i++)
        if output[i]<=1
            ++count;
}
OpenACC

seedRandomNumberGenerator();
for (int i = 0; i<2*niter; i++)
    randomnums[i] = rand();

#pragma acc parallel copyin(randomnums[0:200000]) create(x,y,z)
{
    for (i = 0; i<100000; i++)
        int temp = i+i;
        x = randomnums[temp];
        y = randomnums[temp+1];
        z = (x*x)+(y*y);
        output[i] = z;

    for(i = 0; i<100000; i++)
        if output[i]<=1
            ++count;
}
OpenACC

seedRandomNumberGenerator();
for (int i = 0; i<2*niter; i++)
    randomnums[i] = rand();

#pragma acc parallel copyin(randomnums[0:200000]) create(x,y,z)
{
    for (i = 0; i<100000; i++)
        int temp = i+i;
        x = randomnums[temp];
        y = randomnums[temp+1];
        z = (x*x)+(y*y);
        output[i] = z;

    for(i = 0; i<100000; i++)
        if output[i]<=1
            ++count;
}

Compiler will recognize this loop as a separate reduction kernel and deal with it separately from the Monte Carlo kernel.
seedRandomNumberGenerator();
for (int i = 0; i<2*niter; i++)
    randomnums[i] = rand();

#pragma acc parallel copyin(randomnums[0:200000]) create(x,y,z)
{
    for (i = 0; i<100000; i++)
        int temp = i+i;
        x = randomnums[temp];
        y = randomnums[temp+1];
        z = (x*x)+(y*y);
        output[i] = z;
    for(i = 0; i<100000; i++)
        if output[i]<=1
            ++count;
}
OpenACC

• To compile:
  – PGI
    • $ module load cudatoolkit
    • $ CC -acc -Minfo=all -ta=nvidia:5.0,cc3x accpi.cpp
  – Cray
    • $ module load craype-accel-nvidia35
    • $ CC -hpragma=acc accpi.cpp
Introduction to GPU Computing

CUDA
Thread Model

- Kernel: Function that the threads execute
- Thread: A parallel instance of the kernel
- Thread Block: A group of threads that execute the same instruction simultaneously and have a common shared memory space
- Grid: A set of thread blocks

<table>
<thead>
<tr>
<th>Grid</th>
<th>Thread Block 0</th>
<th>Thread Block 1</th>
<th>Thread Block 2</th>
<th>Thread Block 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0</td>
<td>Thread 5</td>
<td>Thread 0</td>
<td>Thread 5</td>
<td>Thread 0</td>
</tr>
<tr>
<td>Thread 5</td>
<td>Thread 0</td>
<td>Thread 5</td>
<td>Thread 0</td>
<td>Thread 5</td>
</tr>
</tbody>
</table>

---

36
Memory Hierarchy

Thread Private
Memory Hierarchy

- Shared memory
- Thread Private
Memory Hierarchy

Grid 0

Grid 1

Device global memory

Shared memory

Thread Private
CUDA

– Outline of accelerated Monte Carlo code

  » Allocate memory on the Device

  » Generate random numbers (dart coordinates) on the Device

  » Launch the Monte Carlo operation as a CUDA Kernel

  » Copy the results back to the Host
CUDA

- `cudaMalloc((void**)&randomnums,(2*numDarts)*sizeof(float))`
  - Allocate space on the device for the random numbers (2 per point)

- `cudaMalloc((void**)&count_d,(numDarts)*sizeof(int))`
  - Allocate space on the device for the computed data

- All of these CUDA functions are called on the CPU host side, in your `main()`
CUDA

- Built in library in the CUDA Toolkit

- Allows creation of high quality random numbers (darts)
  - Via kernel launched on GPU

```c
createGenerator(rng);
seed(rng);
launchRNG(rng, array_d);
destroyRNG();
```

- Called from your host code (main())
__global__ void kernel(int* count_d, float* randomnums) {
    double i, x, y, z, tid, xidx, yidx = 0;
    tid = threadIdx.x;
    i = tid;
    xidx = (i+i);
    yidx = (xidx+1);
    x = randomnums[xidx];
    y = randomnums[yidx];
    z = ((x*x)+(y*y));
    if (z<=1)
        count_d[tid] = 1;
    else
        count_d[tid] = 0;
}
__global__ void kernel(int* count_d, float* randomnums)
  » __global__ tells the compiler that this function is a CUDA kernel

  tid = threadIdx.x;
  » Use of built in variables like this helps determine where each thread is in the block/grid

  x = randomnums[xidx];
  y = randomnums[yidx];
  » Load in 2 random numbers from the cuRand array
z = ((x*x)+(y*y));
if (z<=1)
    count_d[tid] = 1;
else
    count_d[tid] = 0;

» If this thread’s dart lands inside the circle, put a 1 into the count_d array, if not, then put a 0

– To launch the kernel on the Device:
    » kernel<<<numThreadsPerBlock,numBlocks>>>(count_d, randnums);
- cudaMemcpy(count, count_d, numDarts*sizeof(int), cudaMemcpyDeviceToHost);

> Copy the final array back to the host

- for(i = 0; i<numDarts; i++)
  reducedcount += count[i];

> Reduce the array into a single integer
CUDA

• To compile:
  
  -$ module load cudatoolkit

  -$ nvcc -arch=sm_35 cudapi.cu
Introduction to GPU Computing

Thrust
Thrust

• What is Thrust?
  – Template library for the GPU
  – Imitates C++ Standard Template Library (CPU)
  – Provides a library of pre-accelerated functions to use to write code for the GPU
    » Sorting
    » Reductions
    » Transformations
  – Included in the CUDA Toolkit 4.0 and above
Declaring a kernel of the operation to be performed inside a struct

- Seed and generate darts (random numbers) using Thrust

- Use these to do 1000 Monte Carlo operations

- Increment the counter for points inside the circle

In main()

- Use Thrust to launch this kernel on the accelerator to perform it roughly 1000 times
Thrust

• Define the kernel

```
struct montecarlo
{
    __host__ __device__
    float operator()(unsigned int thread_id)
    {
        thrust::default_random_engine rng(seedVal);
        thrust::uniform_real_distribution<float> u01(0,1);
        x = u01(rng);
        y = u01(rng);
        z = (x*x)+(y*y);
        if z<=1
            sum++;
        return sum;
    }
}
```
Thrust

• Define the kernel

```c
struct montecarlo
{
  __host__ __device__
  float operator()(unsigned int thread_id)
  {
    thrust::default_random_engine rng(seedVal);
    thrust::uniform_real_distribution<float> u01(0,1);
    x = u01(rng);
    y = u01(rng);
    z = (x*x)+(y*y);
    if z<=1
      sum++;
    return sum;
  }
};
```
__host__ __device__ means this is a GPU kernel
Thrust

- Define the kernel

```cpp
struct montecarlo
{
    __host__ __device__
    float operator()(unsigned int thread_id)
    {
        thrust::default_random_engine rng(seedVal);
        thrust::uniform_real_distribution<float> u01(0,1);
        x = u01(rng);
        y = u01(rng);
        z = (x*x)+(y*y);
        if z<=1
            sum++;
        return sum;
    }
};
```

Declare a Thrust RNG and declare a distribution for the random numbers (0-1)
Thrust

- Define the kernel

struct montecarlo
{
    __host__ __device__
    float operator()(unsigned int thread_id)
    {
        thrust::default_random_engine rng(seedVal);
        thrust::uniform_real_distribution<float> u01(0,1);
        x = u01(rng);
        y = u01(rng);
        z = (x*x)+(y*y);
        if (z<=1)
            sum++;
        return sum;
    }
};

Generate random numbers
Thrust

- In `main()`, launch the kernel on the GPU

```cpp
float count = 0.0;
count = thrust::transform_reduce(thrust::counting_iterator<int>(0),
     thrust::counting_iterator<int>(nDarts),
     montecarlo(),
     0.0,
     thrust::plus<float>())
```

- `thrust::transform_reduce()` will perform an operation (`montecarlo`) a set number of times (think: threads)
- The returned values is then reduced using `thrust::plus<float>()`
  - The initial value is set to 0.0

- The number of times it is executed (number of threads) is defined by thrust counting iterators
  - `thrust::counting_iterator<int>(0)`
  - `thrust::counting_iterator<int>(numDarts)`
Thrust

• To compile:
  − $ module load cudatoolkit
  − $ nvcc thrust.cu
## Summary

<table>
<thead>
<tr>
<th></th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
</table>
| **OpenACC** | - Simple syntax  
- Hides memory initialization/copy  
- Compiler does most of the work | - Only supported on few compilers  
- Can be difficult to work with |
| **CUDA**   | - High amount of control  
- Designed for NVIDIA GPUs  
- Well established  
- nvcc compiler comes with the CUDA Toolkit  
- Many libraries are available that work with CUDA | - Can be a pain to manually manage the memory copying/initializations  
- Only works on NVIDIA GPUs |
| **Thrust** | - Imitates the C++ STL  
- Commonly used functions are built in (e.g. reductions, random numbers)  
- Can define own kernel functions to use instead | - For those unfamiliar to templates and STL, can be confusing  
- More cryptic way to launch custom kernels |
Note

• All code used here can be found on the OLCF Github a
  – https://github.com/olcf/Accelerating-Serial-Code

• For more information on each of these APIs

• Other tutorials on GPU computing can be found
  – https://www.olcf.ornl.gov/support/tutorials/
QUESTIONS?

Introduction into GPU Computing