Parallel Programming with GPUs

Professor S. Oliveira, University of Iowa. Computer Science and Mathematics Departments and AMCS program member.
SIMD
MIMD Shared Memory

- Virtual view - single globally addressable memory
- May be a single memory or several memories
- Access time can vary (NUMA) based on how close a memory is to the referencing processor
MIMD Distributed Memory

- Each processor has its own private memory
- Data must be explicitly sent from one processor to another over the network to share data
Symmetric Multi-Processing

- Shared memory within a node
  - Multiple sockets
  - Multicore chip in each socket
- Distributed memory across nodes
Multicore Processor
Tools for parallel Programming

- In 1997 MPI (message Passing Interface) for distributed memory systems. In 2008 OpenMPI (open source version) was made available.
- Pthreads or OpenMP (1997) for shared memory programming on computers and processors. OpenMP application programming interface (API) was developed by the OpenMP ARB (arch. Review board).
- CUDA (2003) for host-device architectures GPU’s and also recently Intel’s OpenCL. Graphics GPU’s used OpenGL which is good for graphics application but very difficult for numerical software development.
Massively Parallel Processor with GPUs

- A quiet revolution and potential build-up
  - Calculation: 367 GFLOPS vs. 32 GFLOPS
  - Memory Bandwidth: 86.4 GB/s vs. 8.4 GB/s
  - Until 2007, programmed through graphics API

- GPU in every PC and workstation – massive volume and potential impact
CPUs and GPUs have fundamentally different design philosophies.
Architecture of a CUDA GPU (2007 on)

- Host
- Input Assembler
- Thread Execution Manager

- Parallel Data Cache
  - Texture
  - Load/store

- Global Memory

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2010
ECE 408, University of Illinois, Urbana-Champaign
Arrays of Parallel Threads

• A CUDA kernel is executed by an array of threads
  – All threads run the same code (SPMD)
  – Each thread has an ID that it uses to compute memory addresses and make control decisions

```c
float x = input[threadID];
float y = func(x);
output[threadID] = y;
...```

© 2009 David Kirk/ NVIDIA and Wen-mei W. Hwu
Recorded for the Virtual School of Computational Science and Engineering
Cooperation

- Divide monolithic thread array into multiple blocks
  - Threads within a block cooperate via **shared memory**, **atomic operations** and **barrier synchronization**
  - Threads in different blocks cannot cooperate

© 2009 David Kirk/ NVIDIA and Wen-mei W. Hwu
Recorded for the Virtual School of Computational Science and Engineering
Block IDs and Thread IDs

- Each thread uses IDs to decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D

- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …
CUDA Memory Model Overview

- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
  - Long latency access
- We will focus on global memory for now
CUDA Device Memory

- `cudaMalloc()`
  - Allocates object in the device **Global Memory**
  - Requires two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** allocated object

- `cudaFree()`
  - Frees object from device **Global Memory**
CUDA Device Memory Allocation (cont.)

- Code example:
  - Allocate a 64 * 64 single precision float array
  - Attach the allocated storage to Md
  - “d” is often used to indicate a device data structure

```
TILE_WIDTH = 64;
Float* Md;
int size = TILE_WIDTH * TILE_WIDTH * sizeof(float);
cudaMalloc((void**)&Md, size);
cudaFree(Md);
```

© 2009 David Kirk/ NVIDIA and Wen-mei W. Hwu
Recorded for the Virtual School of Computational Science and Engineering
CUDA Host-Device Data Transfer

• cudaMemcpy() - synchronous
  • memory data transfer
  • Requires four parameters
    • Pointer to destination
    • Pointer to source
    • Number of bytes copied
    • Type of transfer
      • Host to Host
      • Host to Device
      • Device to Host
      • Device to Device

• Asynchronous transfer
  • cudaMemcpyAsync()
CUDA Host-Device Data Transfer

- Code example:
  - Transfer a 64 * 64 single precision float array
  - M is in host memory and Md is in device memory
  - `cudaMemcpyHostToDevice` and `cudaMemcpyDeviceToDeviceToHost` are symbolic constants

```c
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
cudaMemcpy(M, Md, size, cudaMemcpyDeviceToDeviceToHost);
```
• Declspecs
  • global, device, shared, constant

• Keywords
  • threadIdx, blockIdx

• Intrinsics
  • __syncthreads

• Runtime API
  • Memory, symbol, execution management

• Function launch

```c
__device__ float filter[N];
__global__ void convolve (float *image) {
  __shared__ float region[M];
  ...
  region[threadIdx] = image[i];
  __syncthreads()
  ...
  image[j] = result;
}
```

// Allocate GPU memory
`cudaMalloc((void **)&myimage, bytes);`

// 100 blocks, 10 threads per block
`convolve<<<100, 10>>>(myimage);`
__device__ functions cannot have their address taken

For functions executed on the device:
- No recursion
- No static variable declarations inside the function
- No variable number of arguments
CUDA Function Declarations

<table>
<thead>
<tr>
<th></th>
<th>Executed on the:</th>
<th>Only callable from the:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__device__</code></td>
<td>device</td>
<td>device</td>
</tr>
<tr>
<td><code>float DeviceFunc()</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>__global__</code></td>
<td>device</td>
<td>host</td>
</tr>
<tr>
<td><code>void KernelFunc()</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>__host__</code></td>
<td>host</td>
<td>host</td>
</tr>
<tr>
<td><code>float HostFunc()</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- `__global__` defines a kernel function
  Must return `void`
- `__device__` and `__host__` can be used together
Calling a Kernel Function – Thread Creation

- A kernel function must be called with an *execution configuration*:

```cpp
__global__ void KernelFunc(...);
dim3 DimGrid(100, 50); // 5000 thread blocks
dim3 DimBlock(4, 8, 8); // 256 threads per block
size_t SharedMemBytes = 64; // 64 bytes of shared memory
KernelFunc <<< DimGrid, DimBlock, SharedMemBytes >>>(...);
```

- Any call to a kernel function is asynchronous – host can continue processing after the kernel call
int main( void ) {
    cudaDeviceProp  prop;
    int dev;

    cudaGetDevice( &dev );
    printf( "ID of current CUDA device: %d\n", dev );

    memset( &prop, 0, sizeof( cudaDeviceProp ) );
    prop.major = 1;
    prop.minor = 3;
    cudaChooseDevice( &dev, &prop );
    printf( "ID of CUDA device closest to revision 1.3: %d\n", dev );

    cudaSetDevice( dev );
}
// Example 1: Uses 1 thread per block
#define N 100 // size of vectors

__global__ void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x;    // tid = block index for 1 thread/block (N blocks)
    int tid = threadIdx.x;  // tid = thread index for 1 block kernel launch
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}

int main( void ) {
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;

    // allocate the memory on the GPU
    cudaMalloc( (void**)&dev_a, N * sizeof(int) );
    cudaMalloc( (void**)&dev_b, N * sizeof(int) );
    cudaMalloc( (void**)&dev_c, N * sizeof(int) );

// copy the arrays 'a' and 'b' to the GPU
    cudaMemcpy( dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice );

    add<<<N,1>>>( dev_a, dev_b, dev_c );  // kernel launch: 1 thread/block, N blocks
    add<<<1,N>>>( dev_a, dev_b, dev_c );   // kernel launch: N threads/block, 1 block

    // copy the array 'c' back from the GPU to the CPU
    cudaMemcpy( c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost );

    // and display the results...

    // free the memory allocated on the GPU
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );

    return 0;
General case: B blocks, T threads/block

#define N (32 * 1024)
#define B 128
#define T 64

__global__ void add( int *a, int *b, int *c ) {
    int tid = threadIdx.x + blockIdx.x*blockDim.x;
    while (tid < SIZE) {
        c[tid] = a[tid] + b[tid];
        tid += blockDim.x*gridDim.x;
    }
}

Max # blocks = 65,535
Max # threads/block usually = 512
Choosing 128 threads/block: B = (N+127)/128
Using the kernel

int main( void ) {
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;
    // allocate memory on device
    cudaMalloc( (void**)&dev_a, N * sizeof(int) );
    cudaMalloc( (void**)&dev_b, N * sizeof(int) );
    cudaMalloc( (void**)&dev_c, N * sizeof(int) );

    // initialize vectors a & b...
    // copy a & b to device (dev_a & dev_b)
    cudaMemcpy( dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice );

    add<<<B,T>>>( dev_a, dev_b, dev_c );  // kernel launch: T threads/block, B blocks

    // copy from device and use results
    cudaMemcpy( c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost );
}

__global__ void dot(float *a, float *b, float *c) {
    __shared__ float cache[threadsPerBlock];
    int tid = threadIdx.x + blockIdx.x*blockDim.x;
    int cacheIndex = threadIdx.x;

    float temp = 0;
    while ( tid < N ) {
        temp += a[tid]*b[tid];
        tid += blockDim.x * gridDim.x;
    }
    cache[cacheIndex] = temp;

    __syncthreads();}
// parallel reduction: threadsPerBlock must be a power of two!
int i = blockDim.x / 2;
while ( i != 0 ) {
    if ( cacheIndex < i )
        cache[cacheIndex] += cache[cacheIndex + i];
    __syncthreads();
    i /= 2;
}
// cache[0] is now the sum of a[i] * b[i] for all i for this thread block
if ( cacheIndex == 0 )
    c[blockIdx.x] = cache[0];
More on this...

22C:177 High performance and Parallel Computing
(cross-listed with 22M:178) Fall 2011

Covers:

• MPI
• OpenMP
• CUDA (OpenCL?)

See http://www.cs.uiowa.edu/~oliveira/C177-F11/C177-F11-description.html
Dr. Oliveira Current Research

• Granular flow simulation on GPU’s simulating interactions between rocks or balls as they slide and roll; highly dynamic sparse interact’ns
• Clustering for large databases: algorithms for grouping PPN functional modules, grouping genes activated simultaneously in CNS of rats
• Issues about GPUs
  Redesigning algorithms; ML algorithms
• Other applications using MPI and OpenMP
  Applications involving PDEs, ODEs, linear and no linear solvers, preconditioners (recursive algorithms)
Some references......

- *CUDA by Example: An Introduction to General-Purpose GPU Programming*, Sanders & Kandrot (2010)
- *Parallel Programming with MPI*, Pacheco, Morgan Kauffman (1996)