GPU Programming  

Maciej Halber
Aim

- Give basic introduction to CUDA C
  - How to write kernels
  - Memory transfer
- Talk about general parallel computing concepts
  - Memory communication patterns
- Talk about efficiency concerns when writing parallel programs
Parallel Computation - Why do we care?

• It is fast!

• It is scalable!

• It is ubiquitous! (Soon will be)
  • Nvidia Tegra K1

• End of Moore Law?

• Many applications
  • Our favorite - CNNs!
Parallel Computation - Who and Where?

- Intel Xeon Phi
- OpenMP, OpenACC
- GLSL, HLSL - compute shaders
- Major players
  - OpenCL
  - CUDA (focus of this talk)
Parallel Programming in a Nutshell

• A LOT of small programs (threads) running at the same time

  • *GPU doesn’t get out of a bed in the morning for fewer than a couple of thousand threads* - David Luebke

• Serial vs. Parallel Paradigm

  • Trade expressiveness for speed

  • Serial programs are closer to the way we think (?)
CUDA Background

- CUDA is NVidia authored framework which enables parallel programming model
  - Minimal extensions to C/C++ environment
  - Scales to 100’s of cores, 1000’s of parallel thread
  - Heterogeneous programming model (CPU and GPU are separate entities)
  - Programmers can focus on designing parallel algorithms
Host-Device Model

- Host - CPU + System Memory
  - Memory transfer
  - Launching kernels
- Device - GPU
  - Executing kernels fast!
- Similar to OpenGL Client/Server Model
Kernel is a function that is executed on GPU by an array of threads

- Not recursive
- `void` return type
- No static variables

Each thread of a kernel has its own index

Declared with `__global__` qualifier

- GPU only code uses `__device__`
- CPU only code uses `__host__` (implicit)
Kernel - Grid and Blocks

- The kernel invocation specify the way threads are organized
  - grid_size - how many blocks
  - block_size - how many threads
- Take in variable of type size_t or dim3
- Important variables:
  - dim3 threadIdx [.x, .y, .z]
  - dim3 blockIdx [.x, .y, .z]
  - dim3 blockDim [.x, .y, .z]
  - dim3 gridDim [.x, .y, 1]
Kernel - Grid and Blocks

- Immediate questions
  - Why do we divide computation into blocks?
  - When the blocks are run?
  - In what order?
  - How can threads cooperate?
Kernel - Hardware perspective

• In high level hardware perspective CUDA is essentially a bunch of Streaming Multiprocessors
  • Executing single kernel at a time
• Each SM has number of simple processors (CUDA Cores) that can run several threads
  • Single block per single Streaming Multiprocessor (SM)
    • All the threads in a block run on the same SM at the same time
    • All blocks in a kernel finish before any blocks from the next are run
Kernel - Hardware perspective

• Consequences:
  • Efficiency - once a block is finished, new task can be immediately scheduled on a SM
  • Scalability - CUDA code can run on arbitrary number of SM (future GPUs!)
  • No guarantee on the order in which different blocks will be executed
  • Deadlocks - when block X waits for input from block Y, while block Y has already finished

• Take home point:
  • Threads in the same block cooperate to solve (sub) problems (via shared memory)
  • Threads in different blocks should not cooperate at all.
Kernel Example

• Square all numbers in the input vector

```c
__global__ void square( float * d_out, float * d_in ) {
    int idx = threadIdx.x;
    float f = d_in[idx];
    d_out[idx] = f * f;
}
```

• Calling the `square` kernel

```c
square<<<1, ARRAY_SIZE>>> (d_out, d_in);
```
Functions available for GPU code

- Huge range of arithmetic functions
- All `<math.h>` header is available
  - And more - `lgammaf(float x)`
- Random number generation is more tricky
  - CURAND library!
Random Number Generation

- Must include `<curand.h>`
- `curandCreateGenerator(curandGenerator_t * generator, curandRngType_t rng_type);`
- `curandSetPseudoRandomGeneratorSeed(curandGenerator_t generator, unsigned long long seed );`
- `curandGenerateUniform(curandGenerator_t generator, float *outputPtr, size_t num );`
- For your own kernels, include `<curand_kernel.h>`
  - `curand_init(unsigned long long seed, unsigned long long sequence, unsigned long long offset, curandState *state)`
  - `curand_uniform(curandState *state )`
  - `curand_normal(curandState *state )`
Memory Model

- **Thread - Registers**
  - Local variables, visible to single thread
  - Fastest

- **Blocks - Shared memory**
  - Special keyword `__shared__`
  - Visible to all threads in a single block
  - Very fast

- **Kernel - Global memory**
  - Visible to all threads on a device
  - Slowest, but still quite fast (much faster than host / device transfer)
Memory

- Notice \( d_\) and \( h_\) in front of the in and out pointers?
  - A common convention to differentiate between pointers to host / device memory
- Before doing computation we need to copy data from host to device
- Then we invoke the kernel
- And after we copy data back from device to host

```c
// allocate GPU memory
cudaMalloc((void**) &d_in, ARRAY_BYTES);
cudaMalloc((void**) &d_out, ARRAY_BYTES);

// transfer the array to the GPU
cudaMemcpy(d_in, h_in, ARRAY_BYTES, cudaMemcpyHostToDevice);

// launch the kernel
square<<<1, ARRAY_SIZE>>>(d_out, d_in);

// copy back the result array to the CPU
cudaMemcpy(h_out, d_out, ARRAY_BYTES, cudaMemcpyDeviceToHost);
```
GPU Memory Allocation, Copying, Release

• Should look familiar if you did some C!
  
  • `cudaMalloc( void ** pointer, size_t nbytes )`
  
  • `cudaMemset( void *pointer, int value, size_t count )`
  
  • `cudaFree( void *pointer)`
  
  • `cudaMemcpy( void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction )`
    
    • `cudaMemcpyHostToDevice`
    
    • `cudaMemcpyDeviceToHost`
    
    • `cudaMemcpyDeviceToDevice`
Streams

- `cudaMemcpy(...)` blocks execution of CPU code until finished
  - Kernel can not be launched
- Possible to interleave kernel launches and memory transfer using streams and `cudaMemcpyAsync(...)`
  - Launches memory transfer and goes back executing CPU code
  - Synchronization might be necessary
- Need to specify on which stream kernel should operate

```c
cudaStream_t stream1;
cudaStreamCreate( &stream1 );

cudaMemcpyAsync( d_array1, h_array1, ARRAY_BYTES,
            cudaMemcpyHostToDevice, stream1 );

kernelLaunch<<<N_BLOCKS, N_THREADS, 0, stream1>>>( d_array );

cudaMemcpyAsync( h_array1, d_array1, ARRAY_BYTES,
            cudaMemcpyDeviceToHost, stream1 );

cudaStreamDestroy(stream1);
```
Multiple GPUs

• What if we have multiple GPUs?
  • We can launch multiple kernels in parallel!
• `cudaSetDevice` (int dev) sets the current GPU
  • All subsequent calls will use this device
  • Can line up few asynchronous memory transfers and switch a GPU
• Can copy memory between devices, without involving host!
  • `cudaMemcpyPeerAsync` (void* dst_addr, int dst_dev, void* src_addr, int src_dev, size_t num_bytes, cudaStream_t stream)
  • Synchronization between devices is a huge topic
Synchronization

• Threads can access each other’s results through shared and global memory

• Remember all threads run asynchronously!
  • What if thread reads a result before other threads writes it?

• How to ensure correctness?

• CUDA provides few synchronization mechanisms
  • Barrier - `__syncthreads()`
  • Atomic operations
Barrier

- **__syncthreads()**
- Makes sure all threads are at the same point in execution lifetime
- Example:
  - Needed when copying the data from global to shared memory
  - Need to make sure all threads will access the correct values in memory

```c
__global__ void syncExample( int * d_in, int * d_out, int size ) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    extern __shared__ int sh_array[];

    // read in from array
    sh_array[idx] = d_in[idx];

    // barrier
    __syncthreads();

    // do a shift and modify operation
    if ( idx < size - 1 ) {
        int temp = sh_array[ idx + 1 ];
        __syncthreads();
        sh_array[ idx ] = temp * 2;
    }

d_out[ idx ] = sh_array[idx];
```
Atomic Operations

- CUDA also offers atomic operations
  - `atomicAdd(…)`
  - `atomicSub(…)`
  - `atomicMin(…)`

- No magic here:
  - Atomic operations serialize memory access, so expect performance hit
  - Still useful when developing algorithm
    - Correctness
    - Saves development time
  - Only specific specific operations, data types
    - A custom atomic function can be made using `atomicCAS(…)`
    - Example: [http://stackoverflow.com/questions/17411493/custom-atomic-functions](http://stackoverflow.com/questions/17411493/custom-atomic-functions)
Histogram computation

• Results using naive and atomic implementations

<table>
<thead>
<tr>
<th>Running naive histo</th>
<th>Running simple histo</th>
</tr>
</thead>
<tbody>
<tr>
<td>bin 0: count 1</td>
<td>bin 0: count 103</td>
</tr>
<tr>
<td>bin 1: count 1</td>
<td>bin 1: count 81</td>
</tr>
<tr>
<td>bin 2: count 1</td>
<td>bin 2: count 96</td>
</tr>
<tr>
<td>bin 3: count 1</td>
<td>bin 3: count 101</td>
</tr>
<tr>
<td>bin 4: count 1</td>
<td>bin 4: count 96</td>
</tr>
<tr>
<td>bin 5: count 1</td>
<td>bin 5: count 96</td>
</tr>
<tr>
<td>bin 6: count 1</td>
<td>bin 6: count 113</td>
</tr>
<tr>
<td>bin 7: count 1</td>
<td>bin 7: count 119</td>
</tr>
<tr>
<td>bin 8: count 1</td>
<td>bin 8: count 107</td>
</tr>
<tr>
<td>bin 9: count 1</td>
<td>bin 9: count 112</td>
</tr>
</tbody>
</table>

• Many more optimal ways to do histogram
  • Per thread histogram, then reduce the local histograms into full global one
Efficiency concerns

- In parallel computing we care about performance!
- Couple layers of optimization practices
  - Good practices, High-level strategies
  - Architecture specific optimization
  - Micro-optimization (Ninja)
Efficiency concerns

- In do parallel computing we care about performance!
- Couple layers of optimization practices
  - Good practices, High-level strategies
  - Architecture specific optimization
  - Micro-optimization (Ninja)
- General rule:

  Computation is fast, Memory I/O is slow
Good practices

• Minimize time spend on memory transfer per thread
  • Move frequently-accessed data to fast memory
• Maximize time spend on computation per thread
  • Give threads actual work to do!
• Avoid thread divergence
  • Warps
• Memory coalescing
• Optimal block size (bit architecture specific)
Warps

• Important to understand!

• Similar to SIMD instructions on CPU, Nvidia coins SIMT

• A wrap is a number of data elements GPU can perform single operation on in parallel

• All current CUDA enabled devices have a warp size of 32
  • Single multiply will be done on 32 values
  • Good to have your data size as multiple of 32!
Thread divergence

• Branching code will lead to thread divergence
  ‣ if (…) {} else {}
  ‣ for loops
• How it occurs:
  ‣ GPU is performing a single operation on 32 values
  ‣ If half of the threads in a wrap evaluate true, then the other half need to wait before executing
• In practice, be aware of it, but do not lose sleep over it!
Global memory coalescing

- GPU never reads just single value from global memory
  - Reads in chunks of data
  - GPU is most efficient when threads read or write from contiguous memory locations
- Strided memory access is okay, but only if the stride is low
  - With big stride can be very bad.
- Random is considered very bad!
Correct block size

• Choosing a correct block size might lead to better performance

• Note that single Streaming Multiprocessor executes single kernel at a time (without streams)!

• We might want to know what is the maximum number of threads per SM to decide how to initialize kernel

• To get that information we call deviceQuery utility, which will printout information about device that we are using
deviceQuery Output

CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GT 650M"
  CUDA Driver Version / Runtime Version: 5.5 / 5.5
  CUDA Capability Major/Minor version number: 3.0
  Total amount of global memory: 1024 MBytes (107341444 bytes)
  ( 2) Multiprocessors, (192) CUDA Cores/MP:
    384 CUDA Cores
  GPU Clock rate: 900 Mhz (0.90 GHz)
  Memory Clock rate: 2508 Mhz
  Memory Bus Width: 128-bit
  L2 Cache Size: 262144 bytes
  Maximum Texture Dimension Size (x,y,z):
    1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers:
    1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers:
    2D=(16384, 16384), 2048 layers
  Total amount of constant memory: 65536 bytes
  Total amount of shared memory per block: 49152 bytes
  Total number of registers available per block: 65536
  Warp size: 32
  Maximum number of threads per multiprocessor: 2048
  Maximum number of threads per block: 1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size (x,y,z):
    (2147483647, 65535, 65535)
  Maximum memory pitch: 2147483647 bytes
  Texture alignment: 512 bytes
  Concurrent copy and kernel execution: Yes with 1 copy engine(s)
  Run time limit on kernels: Yes
  Integrated GPU sharing Host Memory: No
  Support host page-locked memory mapping: Yes
  Alignment requirement for Surfaces: Yes
  Device has ECC support: Disabled
  Device supports Unified Addressing (UVA): Yes
  Device PCI Bus ID / PCI location ID: 1 / 0
  Compute Mode:
    < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 5.5, CUDA Runtime Version = 5.5, NumDevs = 1, Device0 = GeForce GT 650M
Result = PASS
Correct block size

• Not always correct to use maximum block size!
  
  • Most likely 512 or 1024 on your devices

• If we want to architecture agnostic - smaller is safer. Usual value is 256

• How many depends on how much sharing needs to happen between threads in a block
  
  • Might require some benchmarking
Data types and Intrinsic functions

- Use floats if you don’t need double precision
  - NVidia Tesla
    - Peak double precision performance - 1.32 Tflops
    - Peak single precision performance - 4.29 Tflops
- If really not that much concerned with the precision, can use CUDA intrinsic functions:
  - `__sinf(x)`
  - `__powf(x, y)`
Memory Communication Patterns

- We can categorize our memory i/o into few simple categories
  - Map
  - Scatter
  - Gather
    - Stencil
  - Transpose
- How to map tasks and memory together
- Useful for understanding parallel algorithms descriptions you might see in the future
Map

- Tasks read from and write to specific data elements
- One-to-one correspondence between input and output
- Each task does independent work - very efficient on GPU
Gather

- Tasks gather input elements from different locations to compute the result
- Each thread
  - reads from $n$ locations in the input
  - writes to a single location in the output
- Many-to-one correspondence between input and output
Stencil

- Tasks read input from a fixed neighborhood in an array
  - 2D von Neumann, 2D Moore, etc…

- Needs to generate result for each element in an output array
- Specialized gather
  - several-to-one correspondence between input and output
Scatter

- Tasks compute where to write output
- Each thread
  - reads from a single location in the input
  - writes to $n$ locations in the output
- One-to-many correspondence between input and output
Transpose

- Task re-order data elements in memory
- Standard transpose - array, matrix, image...
- Also for data structures
  - Array of Structures vs Structure of Arrays

- If you do a lot of operation on float, it might be better to transpose your data
- Transpose is a special case of gather / scatter
Getting started with CUDA C

• Download link: https://developer.nvidia.com/cuda-downloads


• Udacity CUDA Course: https://developer.nvidia.com/udacity-cs344-intro-parallel-programming
How to build

- `nvcc <filename>.cu [-o <executable>]`
  - Builds release mode

- `nvcc -g <filename>.cu`
  - Builds debug mode

- `-arch=sm_xx`
  - Specifies SM functionality version - check your deviceQuery
  - By default the basic sm_10

- .zip contains CMAKE file to generate makefiles, xcode projects, eclipse projects etc.
Debugging

- Tricky if you have single GPU - OS uses hardware acceleration
- cuda-gdb
  - Just like your usual gdb
  - On Mac need to log in as >console
Debugging

• Extremely useful `checkCudaErrors(...)` function
  • Part of Udacity course source code
  • Each function returns error code
  • If error occurs, it will report it and abort the program

```
invalid argument cudaMemcpy( d_dummy, h_inputImageRGBA, sizeof(uchar4) * numPixels, cudaMemcpyHostToDevice )
```

• `-arch=sm_20` and up allows for `printf()` in kernels
• CUDA has also a lot of useful profiling tools!
  • `nvprof`
  • `nvvp`
NSight Visual Profiler

• Profile perspective - load your executable
• Tons of useful information!

• Memory transfer is expensive!
Image Blurring Example
Image Blurring Example

- Two versions:
  - Using image as uchar4 array - strides memory access
  - Splitting image into channels and blurring separately

- Stride is not too dramatic, so simple AOS performs faster

- Theory still holds - if you just work on single channel, coalesced version is faster
  - un-coalesced 6.31 msecs
  - coalesced 5.29 msecs
Getting started with CUDA Matlab

- Extremely easy if you have Parallel Computing Toolbox
- Functions that you are familiar with are actually overloaded, given an input of a type GPUArray code will be executed on the GPU, without much effort.
  - Get data back using `gather()` function
- Can go beyond that using cudaKernel
  - compile CUDA code to .ptx
    - `nvcc -ptx kernels.cu`
  - Get it in Matlab using parallel.gpu.CUDAKernel
    - `feval()` to run the kernel on GPU
    - Again, gather to get data back `gather()`
CUDA Libraries

- CUFFT
- CUBLAS
  - Basic Linear Algebra Subroutines
- CURAND
- CUSPARSE
  - Linear Algebra for Sparse Matrices
- NVIDIA Performance Primitives (NPP)
  - Image, video, signal processing primitives
- Thrust
  - C++ library of parallel algorithms and data structures
Thanks!
Parallel algorithms - Reduce

• How to add all numbers in an array?

• Inputs:
  • Set of elements
  • Reduction operator

• Operator must be binary and associative

• Operator must be associative

• Serial implementation -> simple for loop
  • Results in an unbalanced tree
Parallel algorithms - Reduce

- We can do better! We want to expose more concurrency
  - Perfectly balanced tree
- Can be thought of as memory communication pattern -> All-to-One
- Reducing 1 Million elements
  - 1024 Blocks x 1024 Threads
  - 1 Block x 1024 Threads
- Let’s see an example!
Parallel algorithms - Scan

- More abstract concept - not very common in serial “world”, very useful in parallel computation
- Inputs:
  - Set of elements
  - Reduction operator
    - Associative
    - Binary
  - Identity Element
- Two flavors
  - Exclusive vs. Inclusive

Input: [13 8 2 7]
Exclusive: [0 13 21 23]
Inclusive: [13 21 23 30]
Hillis/Steele Inclusive Scan

- Each step skip $2^d$ steps, until $2^d < n$
Belloch Exclusive Scan

- Two stages: Reduce and Down Sweep

1. 2. 3. 4. 5. 6. 7. 8.

3. 7. 11. 15.


36.
Bleloch Exclusive Scan

• Down sweep takes different operator

• We start by addend
Belloch Exclusive Scan
Belloch Exclusive Scan

1 → 3
2 → 4
3 → 7
4 → 5
5 → 6
6 → 11
7 → 15
8 → 26
10 → 36
0
Belloch Exclusive Scan
Belloch Exclusive Scan
Scan Applications

- Exclusive sum scan is the cumulative distribution function
- Sorting
  - Radix Sort
  - Merge Sort
- Sparse matrix vector multiplication
- Why do we care?
  - Want process list of input creating list of outputs
  - First output and second input create second output and so on…
    - Serial in nature
  - If we can characterize our computation as a scan, then we can parallelize many problems, that otherwise would be a poor fit for GPU

See:
**Compact**

- Wish to compute indices of relevant objects
- Can use Exclusive Sum Scan
- Scatter input into output using the addresses

<table>
<thead>
<tr>
<th>Input Array</th>
<th>2 1 3 4 2 7 8 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicates</td>
<td>T F F T T F T F</td>
</tr>
<tr>
<td>Addresses</td>
<td>0 - - 1 2 - 3 -</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scan In Array</th>
<th>1 0 0 1 1 0 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addresses</td>
<td>0 1 1 1 2 3 3 4</td>
</tr>
</tbody>
</table>
Scan Applications

- Sparse matrix vector multiplication
  - Value vector - non zero values
  - Column Index - what columns these vectors came from
  - Row Pointer - each row starts with some non zero value, store position of those in value vector

<table>
<thead>
<tr>
<th>a</th>
<th>0</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>d</td>
<td>e</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>f</td>
</tr>
</tbody>
</table>

Value Vector: [a b c d e f]
Column Index: [0 2 0 1 2 2 2]
Row Pointer: [0 2 5]
Global memory coalescing

• Example of coalescing - image blurring

• We calculate per channel average given a stencil

• Better to have structure of 3 arrays, than array of 3 values

• Let’s use nvpp!