Programming GPUs with CUDA

John Mellor-Crummey

Department of Computer Science
Rice University

johnmc@rice.edu
Why GPUs?

• Two major trends
  — GPU performance is pulling away from traditional processors
  — availability of general (non-graphics) programming interfaces

• GPU in every PC and workstation
  — massive volume, potentially broad impact

Figure Credits: NVIDIA CUDA Compute Unified Device Architecture Programming Guide
http://docs.nvidia.com/cuda/cuda-c-programming-guide
Why GPUs?

- Two major trends
  - GPU performance is pulling away from traditional processors

---

Figure Credits: NVIDIA CUDA Compute Unified Device Architecture Programming Guide
http://docs.nvidia.com/cuda/cuda-c-programming-guide
Why GPUs?

- **Two major trends**
  - GPU performance is pulling away from traditional processors

![Graph showing theoretical peak GB/s over years](image)

- > 700 GB/s
- < 100 GB/s
Why GPUs?

• Two major trends
  —GPU performance is pulling away from traditional processors
  —availability of general (non-graphics) programming interfaces

• GPU in every PC and workstation
  —massive volume, potentially broad impact

Figure Credits: NVIDIA CUDA Compute Unified Device Architecture Programming Guide
http://docs.nvidia.com/cuda/cuda-c-programming-guide
Power and Area Efficiency

[Graph showing DP Computational Efficiency with various points representing different processors and a circle highlighting a group labeled GPU]

http://www.realworldtech.com/compute-efficiency-2012/2
GPGPU?

• General Purpose computation using GPU
  — applications beyond 3D graphics
  — typically, data-intensive science and engineering applications

• Data-intensive algorithms leverage GPU attributes
  — large data arrays, streaming throughput
  — fine-grain “single instruction multiple threads” (SIMT) parallelism
  — low-latency floating point computation
GPGPU Programming in 2005

- Stream-based programming model
- Express algorithms in terms of graphics operations
  - use GPU pixel shaders as general-purpose SP floating point units
- Directly exploit
  - pixel shaders
  - vertex shaders
  - video memory

Example: GPUSort (Govindaraju, Manocha; 2005)

Figure Credits: Dongho Kim, School of Media, Soongsil University
//invert the other half of the bitonic array and merge
glBegin(GL_QUADS);
for(int start=0; start<num_quads; start++){
    glTexCoord2f(s+width,0);
    glVertex2f(s,0);
    glTexCoord2f(s+width/2,0);
    glVertex2f(s+width/2,0);
    glTexCoord2f(s+width/2,Height);
    glVertex2f(s+width/2,Height);
    glTexCoord2f(s+width,Height);
    glVertex2f(s,Height);
    s+=width;
}
glEnd();

(Govindaraju, Manocha; 2005)
CUDA

CUDA = Compute Unified Device Architecture

• Software platform for parallel computing on Nvidia GPUs
  — introduced in 2006
  — positioned Nvidia’s GPUs as versatile compute devices

• C plus a few simple extensions
  — write a program for one thread
  — instantiate for many parallel threads
  — familiar language; simple data-parallel extensions

• CUDA is a scalable parallel programming model
  — runs on any number of cores without recompiling

Slide credit: Patrick LeGresley, NVidia
NVIDIA PASCAL P100 (2016)

- 15.3B transistors
- 56 SMs
- 4096-bit HBM2 memory interface

- 64 CUDA cores per SM
  —CUDA core = programmable shader
- 3584 cores total

Figure credit: NVIDIA Tesla P100 Whitepaper
NVIDIA VOLTA V100 (SIMT)

- 21.1B transistors
- 84 Streaming Multiprocessors (SMs)
- Each SM
  - 64 FP32 cores
  - 64 INT32 cores
  - 32 FP64 cores
  - 8 tensor cores (64 FP16 FMA each/cycle)
  - 4 texture units
  - 4 warp schedulers
    - 32-thread groups (warp)
    - 4 warps issue and execute concurrently
- 7.8 TF DP; 125 Tensor TF

Independent thread scheduling enables threads in a warp to execute independently - a key to starvation freedom when threads synchronize.
SIMT Thread Scheduling on Volta

```c
__device__ void insert_after(Node *a, Node *b)
{
    Node *c;
    lock(a); lock(a->next);
    c = a->next;

    a->next = b;
    b->prev = a;

    b->next = c;
    c->prev = b;

    unlock(c); unlock(a);
}
```
Why CUDA?

• Business rationale
  —opportunity for Nvidia to sell more chips
    – extend the demand from graphics into HPC
  —insurance against uncertain future for discrete GPUs
    – both Intel and AMD integrating GPUs onto microprocessors

• Technical rationale
  —hides GPU architecture behind the programming API
    – programmers never write “directly to the metal”
      insulates programmers from details of GPU hardware
    – enables Nvidia to change GPU architecture completely, transparently
      preserves investment in CUDA programs
  —simplifies the programming of multithreaded hardware
    – CUDA automatically manages threads
CUDA Design Goals

• Support heterogeneous parallel programming (CPU + GPU)
• Scale to hundreds of cores, thousands of parallel threads
• Enable programmer to focus on parallel algorithms
  —not GPU characteristics, programming language, scheduling ...
CUDA Software Stack for Heterogeneous Computing

Figure Credit: NVIDIA CUDA Compute Unified Device Architecture Programming Guide 1.1
Key CUDA Abstractions

• Hierarchy of concurrent threads
• Lightweight synchronization primitives
• Shared memory model for cooperating threads
Hierarchy of Concurrent Threads

- Parallel kernels composed of many threads
  - all threads execute same sequential program
  - use parallel threads rather than sequential loops

- Threads are grouped into thread blocks
  - threads in block can sync and share memory

- Blocks are grouped into grids
  - threads and blocks have unique IDs
    - threadIdx: 1D, 2D, or 3D
    - blockIdx: 1D or 2D
  - simplifies addressing when processing multidimensional data

Slide credit: Patrick LeGresley, NVidia
CUDA Programming Example

Computing y = ax + y with a serial loop

```c
void saxpy_serial(int n, float alpha, float *x, float *y) {
    for (int i = 0; i < n; i++)
        y[i] = alpha * x[i] + y[i];
}
// invoke serial saxpy kernel
saxpy_serial(n, 2.0, x, y)
```

Computing y = ax + y in parallel using CUDA

```c
__global__
void saxpy_parallel(int n, float alpha, float *x, float *y) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) y[i] = alpha * x[i] + y[i];
}
// invoke parallel saxpy kernel (256 threads per block)
int nbblocks = (n + 255)/256
saxpy_parallel<<<nbblocks, 256>>>(n, 2.0, x, y)
```
Synchronization and Coordination

- Threads within a block may synchronize with barriers
  
  ... step 1 ...  
  __syncthreads();  
  ... step 2 ...  

- Blocks can coordinate via atomic memory operations  
  —e.g. increment shared counter with atomicInc()
CUDA Memory Model

**Thread**

- Per-thread Local Memory

**Block**

- Per-block Shared Memory

**Kernel 0**

- Sequential Kernels

**Kernel 1**

- Per-device Global Memory

Figure credits: Patrick LeGresley, NVidia
Memory Model (Continued)

Figure credit: Patrick LeGresley, NVidia
• Registers: each SM has 32KB of registers
  — each thread has private registers
  — max # of registers / kernel: 63
  — latency: ~1 cycle; bandwidth ~8,000 GB/s

• L1+Shared Memory: on-chip memory that can be used either as L1 cache to share data among threads in the same thread block
  — 64 KB memory: 48 KB shared / 16 KB L1; 16 KB shared / 48 KB L1
  — latency: 10-20 cycles. bandwidth ~1,600 GB/s

• Local Memory: holds "spilled" registers, arrays

• L2 Cache: 768 KB unified L2 cache, shared among the 16 SMs
  — caches load/store from/to global memory, copies to/from CPU host, and texture requests
  — L2 implements atomic operations

• Global Memory: Accessible by all threads as well as host (CPU). High latency (400-800 cycles), but generally cached
Minimal Extensions to C

- Declaration specifiers to indicate where things live
  - Functions
    - __global__ void KernelFunc(...); // kernel callable from host
      must return void
    - __device__ float DeviceFunc(...); // function callable on device
      no recursion
      no static variables within function
    - __host__ float HostFunc(); // only callable on host
  - Variables (later slide)

- Extend function invocation syntax for parallel kernel launch
  KernelFunc<<<500, 128>>>(...); // 500 blocks, 128 threads each

- Built-in variables for thread identification in kernels
  dim3 threadIdx; dim3 blockIdx; dim3 blockDim;
Invoking a Kernel Function

• Call kernel function with an execution configuration

```c
__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
  — explicit synchronization is needed to block

• `cudaThreadSynchronize()` forces runtime to wait until all preceding device tasks have finished
Example of CUDA Thread Organization

- Grid as 2D array of blocks
- Block as 3D array of threads

```cpp
__global__ void KernelFunction(...);
dim3 dimBlock(4, 2, 2);
dim3 dimGrid(2, 2, 1);
KernelFunction<<<dimGrid, dimBlock>>>(...);
```
CUDA Variable Declarations

<table>
<thead>
<tr>
<th><strong>device</strong> <strong>local</strong></th>
<th>int LocalVar;</th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong> <strong>shared</strong></td>
<td>int SharedVar;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>device</strong></td>
<td>int GlobalVar;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>device</strong> <strong>constant</strong></td>
<td>int ConstantVar;</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- __device__ is optional with __local__, __shared__, or __constant__
- Automatic variables without any qualifier reside in a register
  — except arrays: reside in local memory
- Pointers
  — allocated on the host and passed to the kernel
  - __global__ void Kernelfunc(float *ptr)
  — address obtained for a global variable: float *ptr = &GlobalVar
Using Per Block Shared Memory

- Share variables among threads in a block with shared memory
  
  ```c
  __shared__ int scratch[blocksize];
  scratch[threadIdx.x] = arr[threadIdx.x];
  // ...
  // ... compute on scratch values
  // ...
  arr[threadIdx.x] = scratch[threadIdx.x];
  ```

- Communicate values between threads
  
  ```c
  scratch[threadIdx.x] = arr[threadIdx.x];
  __syncthreads();
  int left = scratch[threadIdx.x - 1];
  ```
Features Available in GPU Code

• Special variables for thread identification in kernels
  ```
  dim3 threadIdx; dim3 blockIdx; dim3 blockDim;
  ```

• Intrinsics that expose specific operations in kernel code
  ```
  _syncthreads(); // barrier synchronization
  ```

• Standard math library operations
  —exponentiation, truncation and rounding, trigonometric functions, min/max/abs, log, quotient/remainder, etc.

• Atomic memory operations
  —atomicAdd, atomicMin, atomicAnd, atomicCAS, etc.
Runtime Support

- Memory management for pointers to GPU memory
  - `cudaMalloc()`, `cudaFree()`
- Copying from host to/from device, device to device
  - `cudaMemcpy()`, `cudaMemcpy2D()`, `cudaMemcpy3D()`
More Complete Example: Vector Addition

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

```
int main()
{
    ...
    // Run N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```
// allocate and initialize host (CPU) memory
float *h_A = ..., *h_B = ...;

// allocate device (GPU) memory
float *d_A, *d_B, *d_C;
cudaMalloc( (void**) &d_A, N * sizeof(float) );
cudaMalloc( (void**) &d_B, N * sizeof(float) );
cudaMalloc( (void**) &d_C, N * sizeof(float) );

// copy host memory to device
cudaMemcpy( d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevice ) ;
cudaMemcpy( d_B, h_B, N * sizeof(float), cudaMemcpyHostToDevice ) ;

// execute the kernel on N/256 blocks of 256 threads each
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);
Extended C Summary

- **Declspeps**
  - global, device, shared, local, 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
void *myimage = cudaMalloc(bytes)

// 100 blocks, 10 threads per block
convolve<<<100, 10>>>(myimage);
```
Compiling CUDA

C/C++ CUDA Application

NVCC

CPU Code

PTX Code

Generic

PTX to Target Translator

Target device code

Specialized

GPU

... (ellipsis)

GPU

Target device code

35
Ideal CUDA programs

- High intrinsic parallelism
  - e.g. per-element operations

- Minimal communication (if any) between threads
  - limited synchronization

- High ratio of arithmetic to memory operations

- Few control flow statements
  - SIMT execution
    - divergent paths among threads in a block may be serialized (costly)
    - compiler may replace conditional instructions by predicated operations to reduce divergence
// Host multiplication function
// Compute C = A * B
// hA is the height of A
// wA is the width of A
// wB is the width of B
void Mul(const float* A, const float* B, int hA, int wA, int wB, float* C)
{
    int size;

    // Load A and B to the device
    float* Ad;
    size = hA * wA * sizeof(float);
    cudaMalloc((void**)&Ad, size);
    cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
    float* Bd;
    size = wA * wB * sizeof(float);
    cudaMalloc((void**)&Bd, size);
    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);

    // Allocate C on the device
    float* Cd;
    size = hA * wB * sizeof(float);
    cudaMalloc((void**)&Cd, size);

    // Compute the execution configuration assuming
    // the matrix dimensions are multiples of BLOCK_SIZE
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(wB / dimBlock.x, hA / dimBlock.y);

    // Launch the device computation
    Muld<<<dimGrid, dimBlock>>>(Ad, Bd, wA, wB, Cd);

    // Read C from the device
    cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(Ad);
    cudaFree(Bd);
    cudaFree(Cd);
}
CUDA Matrix Multiply: Device Code

// Device multiplication function called by Mul()
// Compute C = A * B
// wA is the width of A
// wB is the width of B
__global__ void Mul(float* A, float* B, int wA, int wB, float* C)
{
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Index of the first sub-matrix of A processed by the block
    int aBegin = bx * BLOCK_SIZE + tx;
    int aEnd = aBegin + wA;

    // Step size used to iterate through the sub-matrices of A
    int aStep = BLOCK_SIZE;

    // Index of the first sub-matrix of B processed by the block
    int bBegin = by * BLOCK_SIZE + ty;
    int bEnd = bBegin + wB;

    // Step size used to iterate through the sub-matrices of B
    int bStep = BLOCK_SIZE;

    // The element of the block sub-matrix that is computed
    float Csub = 0;

    // Loop over all the sub-matrices of A and B required to
    // compute the block sub-matrix
    for (int a = aBegin, b = bBegin;
        a <= aEnd;
        a += aStep, b += bStep) {

        // Shared memory for the sub-matrix of A
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

        // Shared memory for the sub-matrix of B
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

        // Load the matrices from global memory to shared memory;
        // each thread loads one element of each matrix
        As[ty][tx] = A[a + wA * ty + tx];
        Bs[ty][tx] = B[b + wB * ty + tx];

        // Synchronize to make sure the matrices are loaded
        __syncthreads();

        // Multiply the two matrices together;
        // each thread computes one element
        // of the block sub-matrix
        for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += As[ty][k] * Bs[k][tx];

        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();
    }

    // Write the block sub-matrix to global memory;
    // each thread writes one element
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
Concurrent Kernel Execution in Fermi

Serial Kernel Execution

Concurrent Kernel Execution

http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf
Optimization Considerations

- **Kernel optimizations**
  - make use of shared memory
  - minimize use divergent control flow
    - SIMT execution must follow all paths taken within a thread group
  - use intrinsic instructions when possible
    - exploit the hardware support behind them

- **CPU/GPU interaction**
  - use asynchronous memory copies

- **Key resource considerations for Fermi GPU’s**
  - max dimensions of a block (1024, 1024, 64)
  - max 1024 threads per block
  - 32K registers per SM
  - 16 KB / 48KB cache
  - 48 KB / 16KB shared memory
  - see the programmer’s guide for a complete set of limits for compute capability 2.x
Portable CUDA Alternative: OpenCL

- Framework for writing programs that execute on heterogeneous platforms, including CPUs, GPUs, etc.
  - supports both task and data parallelism
  - based on subset of ISO C99 with extensions for parallelism
  - numerics based on IEEE 754 floating point standard
  - efficiently interoperated with graphics APIs, e.g. OpenGL

- OpenCL managed by non-profit Khronos Group

- Initial specification approved for public release Dec. 8, 2008
  - specification 1.2 released Nov 14, 2011
OpenCL Kernel Example: 1D FFT

```c
__kernel void fft1D_1024 (__global float2 *in, __global float2 *out,
    __local float *sMemx, __local float *sMemy) {
    int tid = get_local_id(0);
    int blockIdx = get_group_id(0) * 1024 + tid;
    float2 data[16];

    // starting index of data to/from global memory
    in = in + blockIdx;  out = out + blockIdx;

    globalLoads(data, in, 64); // coalesced global reads
    fftRadix16Pass(data);      // in-place radix-16 pass
    twiddleFactorMul(data, tid, 1024, 0);

    // local shuffle using local memory
    localShuffle(data, sMemx, sMemy, tid, (((tid & 15) * 65) + (tid >> 4)));
    fftRadix16Pass(data);      // in-place radix-16 pass
    twiddleFactorMul(data, tid, 64, 4); // twiddle factor multiplication
    localShuffle(data, sMemx, sMemy, tid, (((tid >> 4) * 64) + (tid & 15)));

    // four radix-4 function calls
    fftRadix4Pass(data); fftRadix4Pass(data + 4);
    fftRadix4Pass(data + 8); fftRadix4Pass(data + 12);

    // coalesced global writes
    globalStores(data, out, 64);
}
```
OpenCL Host Program: 1D FFT

```
// create a compute context with GPU device
context = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);

// create a work-queue
queue = clCreateWorkQueue(context, NULL, NULL, 0);

// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(float)*2*num_entries, srcA);
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(float)*2*num_entries, NULL);

// create the compute program
program = clCreateProgramFromSource(context, 1, &fft1D_1024_kernel_src, NULL);

// build the compute program executable
clBuildProgramExecutable(program, false, NULL, NULL);

// create the compute kernel
kernel = clCreateKernel(program, "fft1D_1024");

// create N-D range object with work-item dimensions
global_work_size[0] = n;
local_work_size[0] = 64;
range = clCreateNDRangeContainer(context, 0, 1, global_work_size, local_work_size);

// set the args values
clSetKernelArg(kernel, 0, (void *)&memobjs[0], sizeof(cl_mem), NULL);
clSetKernelArg(kernel, 1, (void *)&memobjs[1], sizeof(cl_mem), NULL);
clSetKernelArg(kernel, 2, NULL, sizeof(float)*(local_work_size[0]+1)*16, NULL);
clSetKernelArg(kernel, 3, NULL, sizeof(float)*(local_work_size[0]+1)*16, NULL);

// execute kernel
clExecuteKernel(queue, kernel, NULL, range, NULL, 0, NULL);
```
References


- http://defectivecompass.wordpress.com/2006/06/25/learning-from-gpusort


- http://www.khronos.org/opencl

- Vivek Sarkar. Introduction to General-Purpose computation on GPUs (GPGPUs), COMP 635, September 2007.