An Introduction to GPUs, CUDA and OpenCL

Bryan Catanzaro, NVIDIA Research
Overview

- Heterogeneous parallel computing
- The CUDA and OpenCL programming models
- Writing efficient CUDA code
- Thrust: making CUDA C++ productive
Heterogeneous Parallel Computing

Latency-Optimized CPU
- Fast Serial Processing

Throughput-Optimized GPU
- Scalable Parallel Processing
Why do we need heterogeneity?

- Why not just use latency optimized processors?
  - Once you decide to go parallel, why not go all the way
  - And reap more benefits

- For many applications, throughput optimized processors are more efficient: faster and use less power
  - Advantages can be fairly significant
Why Heterogeneity?

- Different goals produce different designs
  - Throughput optimized: assume work load is highly parallel
  - Latency optimized: assume work load is mostly sequential

- To minimize latency experienced by 1 thread:
  - lots of big on-chip caches
  - sophisticated control

- To maximize throughput of all threads:
  - multithreading can hide latency ... so skip the big caches
  - simpler control, cost amortized over ALUs via SIMD
### Latency vs. Throughput

<table>
<thead>
<tr>
<th>Specifications</th>
<th>Westmere-EP</th>
<th>Fermi (Tesla C2050)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing Elements</td>
<td>6 cores, 2 issue, 4 way SIMD @3.46 GHz</td>
<td>14 SMs, 2 issue, 16 way SIMD @1.15 GHz</td>
</tr>
<tr>
<td>Resident Strands/Threads (max)</td>
<td>6 cores, 2 threads, 4 way SIMD: 48 strands</td>
<td>14 SMs, 48 SIMD vectors, 32 way SIMD: 21504 threads</td>
</tr>
<tr>
<td>SP GFLOP/s</td>
<td>166</td>
<td>1030</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>32 GB/s</td>
<td>144 GB/s</td>
</tr>
<tr>
<td>Register File</td>
<td>~6 kB</td>
<td>1.75 MB</td>
</tr>
<tr>
<td>Local Store/L1 Cache</td>
<td>192 kB</td>
<td>896 kB</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>1.5 MB</td>
<td>0.75 MB</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>12 MB</td>
<td>-</td>
</tr>
</tbody>
</table>

Westmere-EP (32nm)

Fermi (40nm)
- Single Instruction Multiple Data architectures make use of data parallelism
- We care about SIMD because of area and power efficiency concerns
  - Amortize control overhead over SIMD width
- Parallelism exposed to programmer & compiler
SIMD: Neglected Parallelism

- OpenMP / Pthreads / MPI all neglect SIMD parallelism
- Because it is difficult for a compiler to exploit SIMD
- How do you deal with sparse data & branches?
  - Many languages (like C) are difficult to vectorize

- Most common solution:
  - Either forget about SIMD
    - Pray the auto-vectorizer likes you
  - Or instantiate intrinsics (assembly language)
  - Requires a new code version for every SIMD extension
Can we just ignore SIMD?

- Neglecting SIMD is becoming more expensive
  - AVX: 8 way, MIC: 16 way, Nvidia: 32 way, AMD GPU: 64 way

- This problem composes with thread level parallelism
- We need a programming model which addresses both SIMD and threads
CUDA is a programming model designed for:
- Throughput optimized architectures
- Wide SIMD parallelism
- Scalability

CUDA provides:
- A thread abstraction to deal with SIMD
- Synchronization & data sharing between small groups of threads

CUDA programs are written in C++ with minimal extensions

OpenCL is inspired by CUDA, but HW & SW vendor neutral
- Similar programming model, C only for device code
Hierarch of Concurrent Threads

- Parallel **kernels** composed of many threads
  - all threads execute the same sequential program

- Threads are grouped into **thread blocks**
  - threads in the same block can cooperate

- Threads/blocks have unique IDs
Hello World: Vector Addition

//Compute vector sum C=A+B
//Each thread performs one pairwise addition
__global__ void vecAdd(float* a, float* b, float* c) {
    int i = blockIdx.x * blockDim.x + threadIdx.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);
}
What is a CUDA Thread?

- Independent thread of execution
  - has its own program counter, variables (registers), processor state, etc.
  - no implication about how threads are scheduled

- CUDA threads might be **physical** threads
  - as mapped onto NVIDIA GPUs

- CUDA threads might be **virtual** threads
  - might pick 1 block = 1 physical thread on multicore CPU
What is a CUDA Thread Block?

- Thread block = a (data) parallel task
  - all blocks in kernel have the same entry point
  - but may execute any code they want

- Thread blocks of kernel must be independent tasks
  - program valid for *any interleaving* of block executions
CUDA Supports:

- Thread parallelism
  - each thread is an independent thread of execution

- Data parallelism
  - across threads in a block
  - across blocks in a kernel

- Task parallelism
  - different blocks are independent
  - independent kernels executing in separate streams
Synchronization

- Threads within a block may synchronize with barriers

  ```
  ... Step 1 ...
  __syncthreads();
  ... Step 2 ...
  ```

- Blocks **coordinate** via atomic memory operations
  - e.g., increment shared queue pointer with `atomicInc()`

- Implicit barrier between **dependent kernels**

  ```
  vec_minus<<<nbblocks, blksize>>>(a, b, c);
  -------------------------------
  vec_dot<<<nbblocks, blksize>>>(c, c);
  ```
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 ... can easily deadlock

- Independence requirement gives scalability
Scalability

- How do we write code that scales for parallel processors of different sizes?

- CUDA allows one binary to target all these chips
- Thread blocks bring scalability!
Memory model

Thread

Per-thread Local Memory

Block

Per-block Shared Memory
Memory model

Sequential Kernels

Kernel 0

Kernel 1

...
Memory model

- Host Memory
- Device 0 Memory
- Device 1 Memory

`cudaMemcpy()`
int main() {
    int N = 256 * 1024;
    float* h_a = malloc(sizeof(float) * N);
    //Similarly for h_b, h_c. Initialize h_a, h_b

    float *d_a, *d_b, *d_c;
    cudaMalloc(&d_a, sizeof(float) * N);
    //Similarly for d_b, d_c

    cudaMemcpy(d_a, h_a, sizeof(float) * N, cudaMemcpyHostToDevice);
    //Similarly for d_b

    //Run N/256 blocks of 256 threads each
    vecAdd<<<N/256, 256>>>(d_a, d_b, d_c);

    cudaMemcpy(h_c, d_c, sizeof(float) * N, cudaMemcpyDeviceToHost);
}
CUDA: Minimal extensions to C/C++

- Declaration specifiers to indicate where things live
  
  \_\_global\_ void KernelFunc(...);  // kernel callable from host
  \_\_device\_ void DeviceFunc(...);  // function callable on device
  \_\_device\_ int GlobalVar;  // variable in device memory
  \_\_shared\_ int SharedVar;  // in per-block shared memory

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

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

- Intrinsics that expose specific operations in kernel code
  \_\_syncthreads();  // barrier synchronization
Using per-block shared memory

- Variables shared across block

```c
__shared__ int *begin, *end;
```

- Scratchpad memory

```c
__shared__ int scratch[BLOCKSIZE];
scratch[threadIdx.x] = begin[threadIdx.x];
// ... compute on scratch values ...
begin[threadIdx.x] = scratch[threadIdx.x];
```

- Communicating values between threads

```c
scratch[threadIdx.x] = begin[threadIdx.x];
__syncthreads();
int left = scratch[threadIdx.x - 1];
```

- Per-block shared memory is faster than L1 cache, slower than register file

- It is relatively small: register file is 2-4x larger
CUDA: Features available on GPU

- Double and single precision (IEEE compliant)
- Standard mathematical functions
  - `sinf`, `powf`, `atanf`, `ceil`, `min`, `sqrtf`, etc.
- Atomic memory operations
  - `atomicAdd`, `atomicMin`, `atomicAnd`, `atomicCAS`, etc.
- These work on both global and shared memory
OpenCL

- OpenCL has broad industry support
- OpenCL’s data parallel execution model mirrors CUDA, but with different terminology
- OpenCL has rich task parallelism model
  - Runtime walks a dependence DAG of kernels/memory transfers
CUDA and OpenCL correspondence

- Thread ↔ Work-item
- Thread-block ↔ Work-group
- Global memory ↔ Global memory
- Constant memory ↔ Constant memory
- Shared memory ↔ Local memory
- Local memory ↔ Private memory
- __global__ function ↔ __kernel function
- __device__ function ↔ no qualification needed
- __constant__ variable ↔ __constant__ variable
- __device__ variable ↔ __global__ variable
- __shared__ variable ↔ __local__ variable

More information:
OpenCL and SIMD

- SIMD issues are handled separately by each runtime
- AMD CPU Runtime
  - No vectorization
    - Use float4 vectors in your code (float8 when AVX appears?)
- Intel CPU Runtime
  - Vectorization optional, using float4/float8 vectors good idea
- Nvidia GPU Runtime
  - Full vectorization, like CUDA
    - Prefers scalar code per work-item
- AMD GPU Runtime
  - Full vectorization
Imperatives for Efficient CUDA Code

- Expose abundant fine-grained parallelism
  - need 1000’s of threads for full utilization on GPU

- Maximize on-chip work
  - on-chip memory orders of magnitude faster

- Minimize execution divergence
  - SIMT execution of threads in 32-thread warps

- Minimize memory divergence
  - warp loads and consumes complete 128-byte cache line
A many core processor ≡ A device for turning a compute bound problem into a memory bound problem

Kathy Yelick, Berkeley

- Lots of processors, only one socket
- Memory concerns dominate performance tuning
Memory is SIMD too

- Virtually all processors have SIMD memory subsystems

  ![Cache Line Width Diagram](image)

- This has two effects:
  - Sparse access wastes bandwidth
    - 2 words used, 8 words loaded: \( \frac{1}{4} \) effective bandwidth
  - Unaligned access wastes bandwidth
    - 4 words used, 8 words loaded: \( \frac{1}{2} \) effective bandwidth
Coalescing

- GPUs and CPUs both perform memory transactions at a larger granularity than the program requests (e.g., a cache line)
- GPUs have a “coalescer”, which examines memory requests dynamically and coalesces them
- To use bandwidth effectively, when threads load, they should:
  - Present a set of unit strided loads (dense accesses)
  - Keep sets of loads aligned to vector boundaries
- Multidimensional arrays are usually stored as monolithic vectors in memory
- Care should be taken to assure aligned memory accesses for the necessary access pattern
SoA, AoS

- Different data access patterns may also require transposing data structures

Array of Structs

Structure of Arrays

- The cost of a transpose on the data structure is often much less than the cost of uncoalesced memory accesses
Making CUDA Programming Productive

- Libraries are critical to parallel computing
  - FFT
  - BLAS
  - Sort
  - Scan
  - Reduce

- Heterogeneity makes performance portability challenging

- Low-level programming models like CUDA and OpenCL can result in overfitting to a particular piece of hardware

- And if you’re like me, often make your code slow
  - My SGEMM isn’t as good as NVIDIA’s
A C++ template library for CUDA
- Mimics the C++ STL

Containers
- On host and device

Algorithms
- Sorting, reduction, scan, etc.
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>
#include <cstdlib>

int main(void)
{
    // generate 32M random numbers on the host
    thrust::host_vector<int> h_vec(32 << 20);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);

    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;

    // sort data on the device (846M keys per sec on GeForce GTX 480)
    thrust::sort(d_vec.begin(), d_vec.end());

    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

    return 0;
}
Containers

Concise and readable code

- Avoids common memory management errors

```c++
// allocate host vector with two elements
thrust::host_vector<int> h_vec(2);

// copy host vector to device
thrust::device_vector<int> d_vec = h_vec;

// write device values from the host
d_vec[0] = 13;
d_vec[1] = 27;

// read device values from the host
std::cout << "sum: " << d_vec[0] + d_vec[1] << std::endl;
```
Iterators

Pair of iterators defines a range

```cpp
// allocate device memory
device_vector<int> d_vec(10);

// declare iterator variables
device_vector<int>::iterator begin = d_vec.begin();
device_vector<int>::iterator end = d_vec.end();
device_vector<int>::iterator middle = begin + 5;

// sum first and second halves
int sum_half1 = reduce(begin, middle);
int sum_half2 = reduce(middle, end);

// empty range
int empty = reduce(begin, begin);
```
Iterators act like pointers

```cpp
// declare iterator variables
device_vector<int>::iterator begin = d_vec.begin();
device_vector<int>::iterator end = d_vec.end();

// pointer arithmetic
begin++;

// dereference device iterators from the host
int a = *begin;
int b = begin[3];

// compute size of range [begin,end)
int size = end - begin;
```
Iterators

Encode memory location

Automatic algorithm selection

// initialize random values on host
host_vector<int> h_vec(100);
generate(h_vec.begin(), h_vec.end(), rand);

// copy values to device
device_vector<int> d_vec = h_vec;

// compute sum on host
int h_sum = reduce(h_vec.begin(), h_vec.end());

// compute sum on device
int d_sum = reduce(d_vec.begin(), d_vec.end());
Algorithms

- Elementwise operations
  - for_each, transform, gather, scatter ...
- Reductions
  - reduce, inner_product, reduce_by_key ...
- Prefix-Sums
  - inclusive_scan, inclusive_scan_by_key ...
- Sorting
  - sort, stable_sort, sort_by_key ...
// allocate memory
device_vector<int>  A(10);
device_vector<int>  B(10);
device_vector<int>  C(10);

// transform A + B -> C
transform(A.begin(), A.end(), B.begin(), C.begin(), plus<int>());

// transform A - B -> C
transform(A.begin(), A.end(), B.begin(), C.begin(), minus<int>());

// multiply reduction
int product = reduce(A.begin(), A.end(), 1, multiplies<int>());
Standard data types

```cpp
// allocate device memory
device_vector<int> i_vec = ...;
device_vector<float> f_vec = ...;

// sum of integers
int i_sum = reduce(i_vec.begin(), i_vec.end());

// sum of floats
float f_sum = reduce(f_vec.begin(), f_vec.end());
```
Interoperability

Convert iterators to raw pointers & use with CUDA code

```cpp
// allocate device vector
thrust::device_vector<int> d_vec(4);

// obtain raw pointer to device vector’s memory
int * ptr = thrust::raw_pointer_cast(&d_vec[0]);

// use ptr in a CUDA C kernel
my_kernel<<< N / 256, 256 >>>(N, ptr);

// Note: ptr cannot be dereferenced on the host!
```
Productivity Implications

Consider a serial reduction

```c
// sum reduction
int sum = 0;
for(i = 0; i < n; ++i)
    sum += v[i];
```
Productivity Implications

Consider a serial reduction

```c
// product reduction
int product = 1;
for (i = 0; i < n; ++i)
    product *= v[i];
```
Productivity Implications

Consider a serial reduction

// max reduction
int max = 0;
for (i = 0; i < n; ++i)
    max = std::max(max, v[i]);
Productivity Implications

- Compare to low-level CUDA

```c
int sum = 0;
for(i = 0; i < n; ++i)
    sum += v[i];
```

```c
__global__
void block_sum(const float *input,
               float *per_block_results,
               const size_t n)
{
    extern __shared__ float sdata[];

    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

    // load input into __shared__ memory
    float x = 0;
    if(i < n)
    {
        x = input[i];
    }
    ...
```
Leveraging Parallel Primitives

Use `sort` liberally

<table>
<thead>
<tr>
<th>data type</th>
<th>std::sort</th>
<th>tbb::parallel_sort</th>
<th>thrust::sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>25.1</td>
<td>68.3</td>
<td>3532.2</td>
</tr>
<tr>
<td>short</td>
<td>15.1</td>
<td>46.8</td>
<td>1741.6</td>
</tr>
<tr>
<td>int</td>
<td>10.6</td>
<td>35.1</td>
<td>804.8</td>
</tr>
<tr>
<td>long</td>
<td>10.3</td>
<td>34.5</td>
<td>291.4</td>
</tr>
<tr>
<td>float</td>
<td>8.7</td>
<td>28.4</td>
<td>819.8</td>
</tr>
<tr>
<td>double</td>
<td>8.5</td>
<td>28.2</td>
<td>358.9</td>
</tr>
</tbody>
</table>

Intel Core i7 950

NVIDIA GeForce 480
Input-Sensitive Optimizations
Thrust on github

- Quick Start Guide
- Examples
- Documentation
- Mailing list (thrust-users)
Summary

- Heterogeneous parallel computing is here
  - We need both latency and throughput optimized processing

- Programming models like CUDA and OpenCL enable us to capitalize on heterogeneity

- CUDA and OpenCL encourage SIMD friendly, highly scalable algorithm design and implementation

- Thrust is a productive, efficient C++ library for CUDA development
Questions?

Bryan Catanzaro
bcatanzaro@nvidia.com

http://research.nvidia.com
Backup
SIMD & Control Flow

- Nvidia GPU hardware handles control flow divergence and reconvergence
  - Write scalar SIMD code, the hardware schedules the SIMD execution
  - One caveat: \_\_syncthreads() can’t appear in a divergent path
    - This will cause programs to hang
  - Good performing code will try to keep the execution convergent within a warp
    - Warp divergence only costs because of a finite instruction cache
Mapping CUDA to Nvidia GPUs

- CUDA is designed to be functionally forgiving

- However, to get good performance, one must understand how CUDA is mapped to Nvidia GPUs

- Threads: each thread is a SIMD vector lane

- Warps: A SIMD instruction acts on a “warp”
  - Warp width is 32 elements: LOGICAL SIMD width

- Thread blocks: Each thread block is scheduled onto an SM
  - Peak efficiency requires multiple thread blocks per SM
The GPU is very deeply pipelined to maximize throughput.

This means that performance depends on the number of thread blocks which can be allocated on a processor.

Therefore, resource usage costs performance:
- More registers $\Rightarrow$ Fewer thread blocks
- More shared memory usage $\Rightarrow$ Fewer thread blocks

It is often worth trying to reduce register count in order to get more thread blocks to fit on the chip:
- For Fermi, target 20 registers or less per thread for full occupancy.
Occupancy (Constants for Fermi)

- The Runtime tries to fit as many thread blocks simultaneously as possible on to an SM
  - The number of simultaneous thread blocks \(B\) is \(\leq 8\)
  - The number of warps per thread block \(T\) is \(\leq 32\)
  - Each SM has scheduler space for 48 warps \(W\)
    - \(B \times T \leq W=48\)
  - The number of threads per warp \(V\) is 32
  - \(B \times T \times V \times \text{Registers per thread} \leq 32768\)
  - \(B \times \text{Shared memory (bytes) per block} \leq 49152/16384\)
    - Depending on Shared memory/L1 cache configuration
  - Occupancy is reported as \(B \times T / W\)