GPU Compiler
Optimizations

Fuli Ma, Kit Morton, Austin Sharp, Chris Schultz
Objective

To show high level optimization techniques used by compilers for GPU architectures.

Keep in mind...

Different Architectures $\rightarrow$ Different Optimizations
Outline

Background
Overview of GPU Architecture
Optimizations
Future Possibilities
Why GPU Programming?

- Massive parallelism
  - 1000s of threads vs 2-8 threads
- Take advantage of all available resources
  - For heavy computation tasks
Why Compiler Optimizations?

- GPU programming models may be unfamiliar to many programmers
- GPU architectures differ
- Performance increase
Optimization Review

● Pipeline Scheduling
  ○ Rearrange instructions
  ○ Avoid RAW and Control Hazards

● Loop Unrolling
  ○ More scheduling opportunities
  ○ Eliminate branch instructions

● Memory-based Optimizations
  ○ Merging Arrays (spatial locality)
  ○ Loop Interchange (temporal locality)
  ○ Loop Fusion (temporal locality)
  ○ Blocking (temporal locality)
Background

- Kernel
- GPU Architecture
  - Warp / Block
  - Memory Hierarchy
Kernel

- Essentially a function that runs over many threads at once
- All threads run the same code
  - Different data
  - D-cache more important than I-cache
Kernel

Example

```c
__global__ void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x;  // handle the data at this index
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```
GPU Architecture

- Threads (FUs)
- Blocks
- Memory Hierarchy
Threads

- One thread corresponds to one FU
- Executes a kernel function
- More lightweight than threads in OS
  - Not managed by operating system
  - Eliminates context switching
  - No I/O handling
Blocks

- Group of threads
  - Different amongst architectures
- Each runs the same kernel
- Threads within a block have unique IDs
- All threads in a block execute same instructions on different data.
Memory (Fast to Slow)

- Register file
- Shared Memory
- Constant Memory
- Texture memory
- Local memory and Global memory
Register File

● Each thread can read and write to registers
● Data is only visible to the thread who wrote it
  ○ Logically, registers are visible per thread
  ○ Architecturally registers are shared within block
  ○ Registers allocated to a thread as needed

● Register pressure
  ○ If not enough for task, pushed out to local memory
Shared Memory

- Visible to all the threads within the block
  - Clears once kernel completes
- This allows for thread-to-thread communication
Shared Memory

● **N-shared memory banks**
  ○ $N = \text{number of threads in a block}$

● Extra space can be used as cache for global memory
Constant Memory

- Read only
- Will not change throughout a kernel execution
- Performance gained when threads in a block read the same location
Texture Memory

- Read only
- When threads inside a block read adjacent memory locations, performance gain
  - Can pull all thread requests in one pull vs. having to pull from separate locations
Local Memory

- Same scope as register files
  - Acts as an overflow for register pressure
- Off chip
- Physically, shares space with global memory
Global Memory

- Is visible to all blocks and host
- Lasts the entire duration
- Physically, is shared with local memory
Optimizations

- Divergence
- Memory Optimizations
- Loop Unrolling
- Source-to-source compilers
Terms

GPGPU - General Purpose Graphics Processing Unit
CUDA - Compute Unified Device Architecture, NVidia’s parallel programming framework
SIMD engines - Single-program multiple-data engine (AMD)
SM - Streaming Multiprocessor (NVidia), consisting of SPs and FUs
SP - Scalar Processors (NVidia)
SC - Stream Core (AMD)
Warp - Execution of kernel on a SP (NVidia)
Divergence

- GPUs fetch a single instruction per cycle.
- When a thread takes divergent paths, paths wait for other threads to fetch instructions.

Optimization Techniques
  - Thread Regrouping
  - Data Reorganization
  - Block Subdivision
Thread Regrouping

- Threads with similar execution paths are arranged into the same warp
- Eliminates thread ID based divergence
Data Reorganization

- Data is reorganized such that all threads in a block (warp) will take the same execution path.
- Requires knowledge of input data.
Warp Subdivision

- Threads in a warp are divided into two sub-warps depending upon divergent paths.
- Sub-warps run independently but are scheduled to maximize computational throughput.
- Does not eliminate divergence but minimizes additional instruction fetches.
Memory Optimization

- Memory Coalescing
  - Memory accessed by thread index
- Vectorization
  - Combining memory accesses for a single thread
- Thread and Thread-block Merging
  - Improves memory sharing while decreasing parallelism
- Data Prefetching
- Avoid Partition Camping
Memory Coalescing

- Reducing global → shared memory transactions
  - Accessing global memory can take up to 100s of instructions
- Sequential and aligned access
  - More recent hardware can adapt to this issue
Memory Coalescing
Sequential and Aligned
Memory Coalescing

Aligned, but not sequential
Memory Coalescing

Sequential but not aligned
Memory Coalescing

Example: Matrix Addition

(a)

```c
__global__ void MatrixAdd(int *in1, int *in2, int *out) {
    int t1 = in1[threadIdx.x * gridDim.x + blockIdx.x];
    int t2 = in2[threadIdx.x * gridDim.x + blockIdx.x];
    out[threadIdx.x * gridDim.x + blockIdx.x] = t1 + t2;
}
```

(b)

```c
__global__ void MatrixAdd(int *in1, int *in2, int *out) {
    int t1 = in1[blockIdx.x * blockDim.x + threadIdx.x];
    int t2 = in2[blockIdx.x * blockDim.x + threadIdx.x];
    out[blockIdx.x * blockDim.x + threadIdx.x] = t1 + t2;
}
```
Example: Matrix Addition

Both matrices are inside global memory
Example: Non-coalescable

```
__global__ void MatrixAdd(int *in1, int *in2, int *out) {
    int t1 = in1[threadIdx.x * blockDim.x + blockIdx.x];
    int t2 = in2[threadIdx.x * blockDim.x + blockIdx.x];
    out[threadIdx.x * blockDim.x + blockIdx.x] = t1 + t2;
}
```

Will take 4 total global memory accesses since data is split up amongst cache lines
Example: Coalescable

```
__global__ void MatrixAdd(int *in1, int *in2, int *out)
{
    int t1 = in1[blockIdx.x * blockDim.x + threadIdx.x];
    int t2 = in2[blockIdx.x * blockDim.x + threadIdx.x];
    out[blockIdx.x * blockDim.x + threadIdx.x] = t1 + t2;
}
```

Will take 2 total global memory accesses since data grabbed by blocks are aligned on the same cache line
Vectorization

- Similar to coalescing
  - Trying to pull related data in one pull
- Combining multiple thread memory accesses into one instruction
- Hardware dependent
  - For NVidia GPUs, vectorization can only be done for texture memory
Vectorization

Example: Matrix Vector Multiplication (naive)

```c
#define A(y,x) A[(y)* width+(x)]
#define globalDimY 1

__global__ void mv_naive(float *A, float *B, float *C, int width) {
    float sum = 0;
    for (int i=0; i< width; i=i+1) {
        float a;
        float b;
        a = A(idx, i);
        b = B[i];
        sum += a*b;
    }
    C[idx] = sum;
}
```
Vectorization

Example: Matrix Vector Multiplication (After vectorization)

```c
#define A(y,x) read_imagef(A, imageSampler, (int2)(y,x))
#define B(y,x) read_imagef(B, imageSampler, (int2)(y,x))
#define globalDimY 1
__kernel void mv_vec(__read_only image2d_t A,
                     __read_only image2d_t B, __global float* C, int width) {
    float sum = 0;
    for (int i=0; i< width/4; i=i+1) {
        float a;
        float b;
        a = A(idx, i);
        b = B(0, i);
        sum += a.x*b.x;
        sum += a.y*b.y;
        sum += a.z*b.z;
        sum += a.w*b.w;
    }
    C[idx] = sum;
}
```
Vectorization

<table>
<thead>
<tr>
<th>R</th>
<th>G</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>G</td>
<td>B</td>
<td>A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R</th>
<th>G</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>G</td>
<td>B</td>
<td>A</td>
</tr>
</tbody>
</table>

Block 0

Block 1
Thread and Thread-Block Merging

- **Thread merge**
  - Combine multiple threads into one
  - Data register reuse
- **Thread-block merge**
  - Shared memory reuse
- Both decrease parallelism
- Degree of optimization depends on hardware
Data Prefetching

- Overlap memory accesses with computation
- I.E. prefetch data for future loop iterations
- Prefetched data are not bound to registers or shared memory, avoid hurting thread-level parallelism
Avoid Partition Camping

- “GPGPU prefers threads to distribute global memory accesses to different partitions of offchip memory”
- Address offsets
  - Separate data with padding for parallel accesses
- Remap block identifiers
Loop Unrolling

- Benefits are much the same as for CPUs
  - More ILP
  - Fewer instructions
  - Fewer branches
  - Don’t overrun I-cache size!!

- It’s hard to estimate how much to unroll
  - Cycling of threads within a warp can avoid latency
  - Threads in a warp share architectural registers
Loop Unrolling

- One method: static analysis of optimized code
  - Compile CUDA code with different degrees of unrolling
  - Disassemble
  - Estimate performance via static analysis
Loop Unrolling

- Eventually too many registers can be used
  - Logical registers are pushed to local memory
  - Can significantly reduce performance
- Fewer threads in a block
- Can’t cycle threads while waiting for IO
Source-to-Source Compilers

- Translates high level language into CUDA or OpenCL
  - High level multi-processing languages: HMPP
  - Sequential languages: C
- Uses directives to show parallelizable parts
- Utilizes optimization techniques described previously
Why Source-to-Source?

- Be able to massively parallelize a program with little to no effort
- Takes time for programmers to familiarize with CUDA/OpenCL
Source-to-Source

Unrolling example

```c
#pragma hmppec unroll 2, contiguous
for (i = 0; i < N; i++)
{
    B[i] = A[i];
}

(a) Transformation using 'contiguous' unroll, the codelet on the left shows the code with pragma, and the codelet on the right shows the resulting transformation.

#pragma hmppec unroll 2, split
for (i=0; i < N; i++)
{
    B[i] = A[i];
}

(b) Transformation using 'split' unroll, the codelet on the left shows the code with pragma, and the codelet on the right shows the resulting transformation.

#pragma hmppec tile i:2
for (i=0; i < N; i++)
{
    B[i] = A[i];
}

(c) Transformation using tiling, the codelet on the left shows the code with pragma, and the codelet on the right shows the resulting transformation.
```
Source-to-Source

Figures 8: Speedup of auto-tuned HMPP-generated CUDA kernels and hand-written CUDA codes over default HMPP CUDA on 2D/3D convolution and PolyBench codes using single and double precision.
Source-to-Source

Nvidia Example vs Source-to-source compiler

Figure 5. Execution time of FFT for different size of input data
Summary

- Existing optimization techniques for CPUs still apply to GPGPU programs
- Unique architecture of GPU creates new needs for optimization
- Some optimization techniques are unique to the GPU architecture
Future Possibilities

- Non-graphics oriented hardware
- Use of read-only memory outside graphics
- OpenCL vs CUDA
- Adopting solutions from HPC and vice-versa
Thank you!

Questions?