GPU Optimization

Chris Rossbach and Calvin Lin
cs380p
Outline

Over the last several classes:

Background from many areas
  Architecture
    Vector processors
    Hardware multi-threading
  Graphics
    Graphics pipeline
    Graphics programming models
  Algorithms
    parallel architectures → parallel algorithms

Programming GPUs
  CUDA
  Basics: getting something working
  Advanced: making it perform
Outline

Over the last several classes:

Background from many areas
   Architecture
      Vector processors
      Hardware multi-threading
   Graphics
      Graphics pipeline
      Graphics programming models
   Algorithms
      parallel architectures $\rightarrow$ parallel algorithms

Programming GPUs
   CUDA
   Basics: getting something working
Each SM has multiple vector units (4)
32 lanes wide $\rightarrow$ warp size
Each SM has multiple vector units (4)
32 lanes wide → warp size
Vector units use *hardware multi-threading*
Each SM has multiple vector units (4)
32 lanes wide → warp size
Vector units use **hardware multi-threading**
Execution → a grid of thread blocks (TBs)
Each TB has some number of threads
Each SM has multiple vector units (4)  
32 lanes wide → warp size  
Vector units use **hardware multi-threading**  
Execution → a grid of thread blocks (TBs)  
Each TB has some number of threads
Review

Each SM has multiple vector units (4)  
32 lanes wide $\rightarrow$ warp size  
Vector units use **hardware multi-threading**  
Execution $\rightarrow$ a grid of thread blocks (TBs)  
Each TB has some number of threads
Review

Each SM has multiple vector units (4)
32 lanes wide → warp size
Vector units use **hardware multi-threading**
Execution → a grid of thread blocks (TBs)
Each TB has some number of threads
GPU Memory Hierarchy
Global variables marked by `__constant__`

constant and can’t be changed in device.

Will be cached by Constant Cache

Located in global memory

Good for threads that access the same address

```
__constant__ int a=10;
__global__ void kernel()
{
    a++; //error
}
```
Texture Cache

SM

Tex

Read-only Data Cache

L2
Texture Cache

Save Data as Texture:
- Provides hardware accelerated filtered sampling of data (1D, 2D, 3D)
- Read-only data cache holds fetched samples
- Backed up by the L2 cache
Texture Cache

Save Data as Texture:
- Provides hardware accelerated filtered sampling of data (1D, 2D, 3D)
- Read-only data cache holds fetched samples
- Backed up by the L2 cache

Why use it?
- Separate pipeline from shared/L1
- Highest miss bandwidth
- Flexible, e.g. unaligned accesses
- What if your problem takes a large number of read-only points as input?
How many threads/blocks should I use?

// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
How many threads/blocks should I use?

// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
How many threads/blocks should I use?

// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;

• Usually things are correct if grid×block dims >= input size
• Getting good performance is another matter
void vecAdd()
{
    dim3 DGrid = ceil(n/256,1,1);
    dim3 DBlock = (256,1,1);
    addKernel<<<DGrid,DBlock>>>(A_d,B_d,C_d,n);
}
Internals

```c
__host__
void vecAdd()
{
    dim3 DGrid = ceil(n/256,1,1);
    dim3 DBlock = (256,1,1);
    addKernel<<<DGrid,DBlock>>>(A_d,B_d,C_d,n);
}

__global__
void addKernel(float *A_d,
               float *B_d,
               float *C_d,
               int n){
    int i = blockIdx.x * blockDim.x
            + threadIdx.x;
    if( i<n )
        C_d[i] = A_d[i] + B_d[i];
}
```
__host__
void vecAdd()
{
    dim3 DGrid = ceil(n/256,1,1);
    dim3 DBlock = (256,1,1);
    addKernel<<<DGrid,DBlock>>>(A_d,B_d,C_d,n);
}

__global__
void addKernel(float *A_d,
               float *B_d,
               float *C_d,
               int n){
    int i = blockIdx.x * blockDim.x
         + threadIdx.x;
    if( i<n )
        C_d[i] = A_d[i] + B_d[i];
}
__host__

```c
void vecAdd()
{
    dim3 DGrid = ceil(n/256,1,1);
    dim3 DBlock = (256,1,1);
    addKernel<<<DGrid,DBlock>>>(A_d,B_d,C_d,n);
}
```

__global__

```c
void addKernel(float *A_d, float *B_d, float *C_d, int n){
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if( i<n )
        C_d[i] = A_d[i] + B_d[i];
}
```
Kernel Launch
Kernel Launch

- Commands by host issued through *streams*
Kernel Launch

- Commands by host issued through \textit{streams}
Kernel Launch

- Commands by host issued through *streams*
  - Kernels in the same stream executed sequentially
Kernel Launch

- Commands by host issued through **streams**
  - Kernels in the same stream executed sequentially
  - Kernels in different streams may be executed concurrently
Kernel Launch

- Commands by host issued through **streams**
  - Kernels in the same stream executed sequentially
  - Kernels in different streams may be executed concurrently
- Streams mapped to GPU HW queues
Kernel Launch

• Commands by host issued through **streams**
  - Kernels in the same stream executed sequentially
  - Kernels in different streams may be executed concurrently

• Streams mapped to GPU HW queues
  - Done by “kernel management unit” (KMU)
Kernel Launch

- Commands by host issued through **streams**
  - Kernels in the same stream executed sequentially
  - Kernels in different streams may be executed concurrently
- Streams mapped to GPU HW queues
  - Done by “kernel management unit” (KMU)
  - Multiple streams mapped to each queue → serializes some kernels
Kernel Launch

- Commands by host issued through *streams*
  - Kernels in the same stream executed sequentially
  - Kernels in different streams may be executed concurrently
- Streams mapped to GPU HW queues
  - Done by “kernel management unit” (KMU)
  - Multiple streams mapped to each queue → serializes some kernels
- Kernel launch distributes thread blocks to SMs
Thread Blocks, Warps, Scheduling
Thread Blocks, Warps, Scheduling

Suppose one TB (threadblock) has 64 threads (2 warps)
Thread Blocks, Warps, Scheduling

Suppose one TB (threadblock) has 64 threads (2 warps)

- SMs split blocks into warps
- Unit of HW scheduling for SM
- 32 threads each
Thread Blocks, Warps, Scheduling

Suppose one TB (threadblock) has 64 threads (2 warps)

- **Thread Blocks**
  - SMs split blocks into warps
  - Unit of HW scheduling for SM
  - 32 threads each

- **SMs**
  - SM_0
  - SM_1
  - SM_12

- **Features**
  - Register File
  - Cores
  - L1 Cache/Shared Memory
Thread Blocks, Warps, Scheduling

Suppose one TB (threadblock) has 64 threads (2 warps)

**Thread Blocks**

**SMs**

- SMs split blocks into warps
- Unit of HW scheduling for SM
- 32 threads each
Thread Blocks, Warps, Scheduling

Suppose one TB (threadblock) has 64 threads (2 warps)
Thread Blocks, Warps, Scheduling

Suppose one TB (threadblock) has 64 threads (2 warps)

- 

SMs

- SM_0
- SM_1
- SM_12

- SMs split blocks into warps
- Unit of HW scheduling for SM
- 32 threads each

Remaining TBs are queued
SIMD vs. SIMT

Flynn’s Taxonomy

<table>
<thead>
<tr>
<th>Instruction Streams</th>
<th>Data Streams</th>
</tr>
</thead>
<tbody>
<tr>
<td>SISD</td>
<td>SIMD</td>
</tr>
<tr>
<td>MISD</td>
<td>MIMD</td>
</tr>
</tbody>
</table>
SIMD vs. SIMT

Flynn’s Taxonomy

<table>
<thead>
<tr>
<th>Instruction Streams</th>
<th>Data Streams</th>
</tr>
</thead>
<tbody>
<tr>
<td>SISD</td>
<td>SIMD</td>
</tr>
<tr>
<td>MISD</td>
<td>MIMD</td>
</tr>
</tbody>
</table>

Single Scalar Thread + Register File

e.g., SSE/AVX
SIMD vs. SIMT

Flynn’s Taxonomy

<table>
<thead>
<tr>
<th>Instruction Streams</th>
<th>Data Streams</th>
</tr>
</thead>
<tbody>
<tr>
<td>SISD</td>
<td>SIMD</td>
</tr>
<tr>
<td>MISD</td>
<td>MIMD</td>
</tr>
</tbody>
</table>

Single Scalar Thread

Loosely synchronized threads

e.g., pthreads

e.g., SSE/AVX
SIMD vs. SIMT

Flynn’s Taxonomy

- **SISD**: Single Instruction, Single Data
- **SIMD**: Single Instruction, Multiple Data
- **MISD**: Multiple Instruction, Single Data
- **MIMD**: Multiple Instruction, Multiple Data

**SIMT**: Synchronous operation
- **RF**: Register File
- **e.g., SSE/AVX**: Single Scalar Thread

**Loosely synchronized threads**
- **Multiple threads**
- **e.g., pthreads**

**SIMD**

**MIMD**

**RF**
A Taco Bar
A Taco Bar
A Taco Bar

- Where is the parallelism here?
GPU: a Multi-lane Taco Bar
GPU: a Multi-lane Taco Bar
GPU: a Multi-lane Taco Bar

1 Taco, please

• Where is the parallelism here?
GPU: a Multi-lane Taco Bar

1 Taco, please

- Where is the parallelism here?
  - There’s none!
  - This only works if you can keep every lane full at every step
  - Throughput == Performance
  - Goal: *Increase Occupancy!*
GPU: a Multi-lane Taco Bar

- Where is the parallelism here?
- There’s none!
- This only works if you can keep every lane full at every step
- Throughput == Performance
- Goal: **Increase Occupancy!**
GPU: a Multi-lane Taco Bar

- Where is the parallelism here?
- There’s none!
- This only works if you can keep every lane full at every step
- Throughput == Performance
- Goal: Increase Occupancy!
GPU: a Multi-lane Taco Bar

• Where is the parallelism here?

• There’s none!
• This only works if you can keep every lane full at every step
• Throughput == Performance
• Goal: Increase Occupancy!
GPU Performance Metric: *Occupancy*
GPU Performance Metric: *Occupancy*

\[
\text{Occupancy} = \frac{\#\text{Active Warps}}{\#\text{MaximumActive Warps}}
\]

Measures how well concurrency/parallelism is utilized
GPU Performance Metric: *Occupancy*

\[
\text{Occupancy} = \frac{\text{(#Active Warps)}}{\text{(Maximum Active Warps)}}
\]

Measures how well concurrency/parallelism is utilized

Occupancy captures:
- *Which resources* can be dynamically shared
- How to reason about resource demands of a kernel
- Enables device-specific tuning of kernel parameters
Hardware Resources Are Finite

SM – Stream Multiprocessor
SP – Stream Processor
Hardware Resources Are Finite

SM – Stream Multiprocessor
SP – Stream Processor

Limited the #thread blocks
Hardware Resources Are Finite

SM – Stream Multiprocessor
SP – Stream Processor
Hardware Resources Are Finite

SM – Stream Multiprocessor
SP – Stream Processor
Hardware Resources Are Finite

- Kernel Distributor
- SM Scheduler
- SM
- SM
- SM
- SM
- DRAM

SM – Stream Multiprocessor
SP – Stream Processor

Thread Block Control
- TB 0

Warp Schedulers

Warp Context
- SP
- SP
- SP
- SP
- SP
- SP
- SP
- SP
- SP
- SP
- SP
- SP

Register File

L1/Shared Memory

Limits the #thread blocks
Limits the #threads
Limits the #threads
Limits the #thread blocks
CUDA Occupancy
CUDA Occupancy

\[
\text{Occupancy} = \frac{\text{(#Active Warps)}}{\text{(#Maximum Active Warps)}}
\]

Measure of how well max capacity is utilized
CUDA Occupancy

**Occupancy** = \( \frac{\text{(Active Warps)}}{\text{(Maximum Active Warps)}} \)

Measure of how well max capacity is utilized

What is the performance impact of varying kernel resource demands?
CUDA Occupancy

**Occupancy** = (#Active Warps) / (#MaximumActive Warps)

Measure of how well max capacity is utilized

Limits on the numerator:
- Registers/thread
- Shared memory/thread block
- Number of scheduling slots: blocks, warps

What is the performance impact of varying kernel resource demands?
CUDA Occupancy

Occupy = (#Active Warps) / (#MaximumActive Warps)

Measure of how well max capacity is utilized

Limits on the numerator:
- Registers/thread
- Shared memory/thread block
- Number of scheduling slots: blocks, warps

Limits on the denominator:
- Memory bandwidth
- Scheduler slots

What is the performance impact of varying kernel resource demands?
Impact of Thread Block Size
Impact of Thread Block Size

Consider Fermi: 1536 threads/SM
Impact of Thread Block Size

Consider Fermi: 1536 threads/SM
At 512 threads/block, how many blocks can execute (per SM)?
Impact of Thread Block Size

Consider Fermi: 1536 threads/SM
At 512 threads/block, how many blocks can execute (per SM)? 3
Impact of Thread Block Size

Consider Fermi: 1536 threads/SM

At 512 threads/block, how many blocks can execute (per SM)?
With 128 threads/block?
Impact of Thread Block Size

Consider Fermi: 1536 threads/SM
At 512 threads/block, how many blocks can execute (per SM)?
With 128 threads/block?

3
12
Impact of Thread Block Size

Consider Fermi: 1536 threads/SM
   At 512 threads/block, how many blocks can execute (per SM)?
   With 128 threads/block?

Consider HW limit of 8 thread blocks/SM @ 128 threads/block:
   Suppose only 1024 active threads at a time
   Occupancy = 0.666 (1024/1536)
Impact of Thread Block Size

Consider Fermi: 1536 threads/SM
   At 512 threads/block, how many blocks can execute (per SM)?
   With 128 threads/block?

Consider HW limit of 8 thread blocks/SM @ 128 threads/block:
   Suppose only 1024 active threads at a time
   Occupancy = 0.666 (1024/1536)

To maximize utilization, thread block size should balance
   demand for thread blocks vs.
   thread slots
Impact of #Registers Per Thread
Impact of #Registers Per Thread

Assume 10 registers/thread and a thread block size of 256
Impact of #Registers Per Thread

Assume 10 registers/thread and a thread block size of 256
Number of registers per SM = 16K
Impact of #Registers Per Thread

Assume 10 registers/thread and a thread block size of 256
Number of registers per SM = 16K
A TB requires 2560 registers \( \rightarrow \) max of 6 thread blocks per SM
  
  Uses all 1536 thread slots (6 blocks \( \times \) 256 threads/block)

\[
2560 \text{ regs/block} \times 6 \text{ block/SM} = 15,360 \text{ registers}
\]
Impact of #Registers Per Thread

Assume 10 registers/thread and a thread block size of 256
Number of registers per SM = 16K
A TB requires 2560 registers → max of 6 thread blocks per SM
   Uses all 1536 thread slots (6 blocks * 256 threads/block)
   2560 regs/block * 6 block/SM = 15,360 registers
What is the impact of increasing number of registers by 2?
Impact of #Registers Per Thread

Assume 10 registers/thread and a thread block size of 256
Number of registers per SM = 16K
A TB requires 2560 registers \(\rightarrow\) max of 6 thread blocks per SM
   Uses all 1536 thread slots (6 blocks * 256 threads/block)
   \(2560\) \textit{regs/block} \(\times\) 6 \textit{block/SM} = 15,360 \textit{registers}
What is the impact of increasing number of registers by 2?
   Granularity of management is a thread block!
Impact of #Registers Per Thread

Assume 10 registers/thread and a thread block size of 256
Number of registers per SM = 16K
A TB requires 2560 registers → max of 6 thread blocks per SM
   Uses all 1536 thread slots (6 blocks * 256 threads/block)
   \[2560 \text{ regs/block} \times 6 \text{ block/SM} = 15,360 \text{ registers}\]

What is the impact of increasing number of registers by 2?
   Granularity of management is a thread block!
   Loss of concurrency of 256 threads!
   \[ (12 \text{ regs/thread} \times 256 \text{ threads/block} \times 5 \text{ blocks/SM} = 15360 \text{ registers}) \]
Impact of Shared Memory

Shared memory is allocated per thread block
  Can limit the number of thread blocks executing concurrently per SM
  gridDim and blockDim parameters impact demand for
  shared memory
  number of thread slots
  number of thread block slots
• Navigate the tradeoffs
  ❖ maximize core utilization and memory bandwidth utilization
  ❖ Device-specific

• **Goal**: Increase occupancy until one or the other is saturated
Pragmatic Strategy: Strike a Balance

- Navigate the tradeoffs
  - Maximize core utilization and memory bandwidth utilization

- Device-specific
  - Goal: Increase occupancy until one or the other is saturated

```
template < class T >
__host__ cudaError_t cudaOccupancyMaxActiveBlocksPerMultiprocessor ( int* numBlocks, T func, int blockSize, size_t dynamicSMemSize ) [inline]

Returns occupancy for a device function.

Parameters
numBlocks
  - Returned occupancy
func
  - Kernel function for which occupancy is calculated
blockSize
  - Block size the kernel is intended to be launched with
dynamicSMemSize
  - Per-block dynamic shared memory usage intended, in bytes
```
Parallel Memory Accesses

**Coalesced** main memory access (16/32x faster)

HW combines multiple warp memory accesses → single coalesced access

**Bank-conflict-free** shared memory access (16/32)

No alignment or contiguity requirements

- CC 1.3: 16 different banks per half warp or same word
- CC 2.x+3.0: 32 different banks + 1-word broadcast each
Parallel Memory Architecture

In a parallel machine, many threads access memory
Therefore, memory is divided into banks
Essential to achieve high bandwidth

Each bank can service one address per cycle
A memory can service as many simultaneous accesses as it has banks

Multiple simultaneous accesses to a bank result in a bank conflict
Conflicting accesses are serialized
Coalesced Main Memory Accesses

single coalesced access

one and two coalesced accesses*
Bank Addressing Examples

No Bank Conflicts
Linear addressing
stride == 1

• No Bank Conflicts
  • Random 1:1 Permutation
2-way Bank Conflicts
Linear addressing
stride == 2

8-way Bank Conflicts
• Linear addressing
  stride == 8

Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 8
Thread 9
Thread 10
Thread 11

Bank 0
Bank 1
Bank 2
Bank 3
Bank 4
Bank 5
Bank 6
Bank 7
Bank 15

Bank 0
Bank 1
Bank 2
Bank 3
Bank 4
Bank 5
Bank 6
Bank 7
Bank 15

Bank 7
Bank 8
Bank 9
Bank 15
Linear Addressing

Given:

```c
__shared__ float shared[256];
float foo =
    shared[baseIndex + s * threadIdx.x];
```

This is only bank-conflict-free if \( s \) shares no common factors with the number of banks 16 on G80, so \( s \) must be odd.
Summary

Understanding u-arch resources is critical for optimization
Need to balance threads, blocks, registers
Memory level parallelism is sensitive to your access patterns!
Often suffices to just explore parameter space