Optimization Tradeoffs
Why talking about trade-offs

• Understand the reasons why there are trade-offs when optimizing a GPU program
• Understand typical trade-offs
• Analyze the performance of a program under different trade-offs
Why we have optimization trade-offs

- Large number of possible combination of optimizations and configurations
- Resources are constrained
  - Local memory
  - Register
  - Global memory bandwidth
- Interaction between threads
  - Optimization sensitive to small changes
• Global memory bandwidth
  – 86.4 GB/s
• Multiprocessor performance
  – 18 Flop*1.35 Ghz = 24.3 GFLOPS
• Peak performance
  – 24.3*16 = 388.8 GFLOPS
• Each flop operates on up to 8 bytes of data
  – 388.8*8 >> 86.4 Gbs
• Global memory bandwidth can be easily saturated
Another example

• The configuration of an application:
  – 256 threads per block
  – 10 registers per thread
  – 4KB shared memory per block
  – 48 blocks -> 3 blocks per multiprocessor

• Two simple changes
Change resource usage

• Increase register usage from 10 to 11
  – Now only two blocks can run on a multiprocessors, because
  – 8192 registers per multiprocessor
  – Each block uses 11*256=2816 registers
  – 3*2816 > 8192

• Increase the shared memory usage from 4KB to 5KB
  – Can still support 3 blocks
    • 16 KB shared memory per multiprocessor
    • 3*5KB < 16
A story of two dimensions

• Build a mindset of optimizing for CUDA

• Dimension 1: Reduce instruction count
  – Common sub-expression elimination
  – Strength reduction
  – ...

• Diminsion 2: Increase multi-processor occupancy
Methods of increasing occupancy

- Schedule sequences of independent instructions within a warp
- Increase number of threads in a thread block
- Assign more blocks to a multi-processor

- Generally, intra-thread optimizations only work when the high-occupancy is maintained.
Source of independent warps

- From a few large thread blocks
- From many small thread blocks
- Different impact on performance
  - Larger thread blocks have better data locality
  - At the same time, larger thread blocks have higher thread synchronization overhead
  - Larger thread blocks potentially waste more thread space per multi-processor
Two attacks

- Intra-thread optimization
  - Instruction count reduction
  - Instruction level parallelism
- Inter-thread balancing
  - Work re-distribution
  - Resource balancing

- The optimizations interact through their effects on register usage.
Example: Matrix Multiplication

• For(…)
  – ___shared___ float As[16][16];
  – ___shared___ float Bs[16][16];

  – As[ty][tx] = A[indexA];
  – Bs[ty][tx] = B[indexB];
  – indexA += 16;
  – IndexB += 16*widthB;
  – ___syncthreads();

  – For(I=0; I<16; I++)
    • {
      – Ctemp+=As[ty][I]*Bs[I][tx];
    }
  – ___syncthreads();

  – C[indexC] = Ctemp;
Intra-thread optimization 1

- Instruction count reduction
  - Strength reduction
  - Common subexpression elimination
  - Loop-invariant code motion

- Loop unrolling
  - Remove branch instruction
  - Array subscript calculation
Unroll matrix multiplication

For(…)

- __shared__ float As[16][16];
- __shared__ float Bs[16][16];

- As[ty][tx] = A[indexA];
- Bs[ty][tx] = B[indexB];
- indexA += 16;
- IndexB += 16*widthB;
- __syncthreads();

- Ctemp+=As[ty][I]*Bs[I][tx];
...
- Ctemp+=As[ty][15]*Bs[15][tx];

- __syncthreads();

- C[indexC] = Ctemp;
Intra-thread optimization 2

- Reduce instruction latency -> increase instruction level parallelism
- Unroll
  - Facilitate instruction scheduling
- Prefetch and software pipelining
  - Reduce global memory access latency
Prefetch in matrix multiplication

- Atemp = A[indexA];
- Btemp = B[indexB];
- For(...)
  - __shared__ float As[16][16];
  - __shared__ float Bs[16][16];
  - As[ty][tx] = Atemp;
  - Bs[ty][tx] = Btemp;
  - indexA += 16;
  - IndexB += 16*widthB;
  - __syncthreads();
  - Atemp = A[indexA];
  - Btemp = B[indexB];
  - For(I=0; I<16; I++)
    • { 
      •   Ctemp+=As[ty][I]*Bs[I][tx];
    • }
  - __syncthreads();
  - C[indexC] = Ctemp;
Inter-thread optimization 1

• Work re-distribution
  – Tile workload
    • Better amortize the global memory latency
    • Reduce the pressure of global memory bandwidth
  – CUDA does a imperfect job
    • Prefer intra-thread performance

• Divide a grid into several kernel invocations
  – Kind of count-intuitive
  – Effective when kernel use constant memory
    • More data per grid -> more constant cache conflicts
Tile matrix multiplication

- For(...)
  - __shared__ float As[16][16];
  - __shared__ float Bs[16][32];

- As[ty][tx] = A[indexA];
- Bs[ty][tx] = B[indexB];
- Bs[ty][tx+16]=B[indexB+16];
- indexA += 16;
- IndexB += 16*widthB;
- __syncthreads();

- For(I=0; I<16; I++)
  - Ctemp+=As[ty][I]*Bs[I][tx];
  - Dtemp+=As[ty][I]*Bs[I][tx+16];
- __syncthreads();

- C[indexC] = Ctemp;
- C[indexC+16] = Dtemp;
• Resource balancing
  – Balance usage of register, shared memory and global memory accesses
  – Sometimes counter-intuitive
Balance shared memory and global memory accesses

- For(...)
  - __shared__ float As[16][16];
  - __shared__ float Bs[16][16];

  - As[ty][tx] = A[indexA];
  - Bs[ty][tx] = B[indexB];
  - indexA += 16;
  - IndexB += 16*widthB;
  - __syncthreads();

  - For(I=0; I<16; I++)
    • {  
      – Ctemp+=As[ty][I]*Bs[I][tx];  
      – Ctemp+=As[ty][I]*B[I*widthB+tx]  
    • }
  - __syncthreads();

  - C[indexC] = Ctemp;
AoS vs SoA on the G80 architecture
struct S {
    float x;
    float y;
};
struct S myData[N]

- preventing coalesced reads

struct S {
    float x[N];
    float y[N];
};
struct S myData;

- Leading to coalesced reads

In this case a SoA seems preferable to an AoS
When is an AoS still preferable on the G800 architecture?

- Alignment specifiers and automatically aligned built-in types allow for 64 or 128-bit reads from global memory.
- Reduction of number of memory operations.
- By adding an alignment specifier to the SoA from the previous slide and reading from global memory into registers...

```c
struct __align__(8) S {
    float x;
    float y;
};
```

... we can improve the performance drastically

- SoA: contiguous reads for x and y (up to 600 cycles)
- AoS: one still contiguous 64-bit read to get x and y (up to 300 cycles)

- Even more obvious for 128-bit structures. SoA ~1200 vs AoS ~300 cycles
AoS and shared memory

• When reading AoS from global memory always think about the shared memory layout.
  – Share memory only supports 32 bit reads/writes
  – AoS that allow for good access to global memory will result in bank conflicts in shared memory.
    • Global memory: 64 bit or 128 bit
    • Shared memory: multiples of a stride of 3 → 96bit

→ Changing the layout of the date from an AoS in global memory to a SoA in shared memory might be beneficial
One step further: SoAoS

For structures that exceed the 128-bit alignment boundary

```c
struct __align__(16) S {
    float a;
    float b;
    float c;
    float d;
    float e;
    float f;
};

struct S myData[N];
```

- Each thread will have to perform 2 128-bit reads.
- The single reads are no longer contiguous.
- Idea: a Structure of Arrays of Structures (SoAoS).
One step further: SoAoS

struct __align__(16) S16 {
    float a;
    float b;
    float c;
    float d;
};

struct __align__(8) S8 {
    float e;
    float f;
};

struct S {
    struct S16 x[N];
    struct S8  y[N];
};

struct S myData;

- The single reads are now again contiguous across threads

- This is just an idea to show that there are many things to try that might lead to better performance for global memory access