



Optimize Algorithms for the GPU



- Maximize independent parallelism
- Maximize arithmetic intensity (math/bandwidth)
- Sometimes it's better to recompute than to cache
 - GPU spends its transistors on ALUs, not memory
- Do more computation on the GPU to avoid costly data transfers
 - Even low parallelism computations can sometimes be faster than transferring back and forth to host

© NVIDIA Corporation 2009

2

Use Parallelism Efficiently



- Partition your computation to keep the GPU multiprocessors equally busy
 - Many threads, many thread blocks
- Keep resource usage low enough to support multiple active thread blocks per multiprocessor
 - Registers, shared memory
- Avoid small branch-granularity

© NVIDIA Corporation 2009

3

Optimize Memory Access



- Coalesced vs. Non-coalesced = order of magnitude
 - Global/Local device memory
- Optimize for spatial locality in cached texture memory
- In shared memory, avoid high-degree bank conflicts

© NVIDIA Corporation 2009

4

Take Advantage of Shared Memory



- Hundreds of times faster than global memory
- Threads can cooperate via shared memory
- Use one / a few threads to load / compute data shared by all threads
- Use it to avoid non-coalesced access
 - Stage loads and stores in shared memory to re-order non-coalesceable addressing

© NVIDIA Corporation 2009

5

Outline



- Execution Configuration Optimizations
- Instruction Optimizations
- Memory Optimizations
 - Data transfers between host and device
 - Device memory optimizations

© NVIDIA Corporation 2009

6

Occupancy



- Thread instructions are executed sequentially, execute other warps to hide latencies
- Number of warps limited by multiprocessor resources
 - Warp limit (24-32)
 - Block limit (8)
 - Registers (8k-16k)
 - Shared memory (16kB)
- Occupancy = # warps running / warp resource limit.

Occupancy != Performance



- ... but low occupancy can hurt performance
- Read-after-write register dependency (~24 cycles)

```
x = y + 5;  
z = x + 3;
```
- Run at least 192 threads (6 warps) per multiprocessor
 - At least 25% occupancy (1.0/1.1), 18.75% (1.2/1.3)
- Memory read latency (400-600 cycles)
 - Hide with arithmetic instructions from other warps
 - Assuming kernel is not memory bound

Threads per Block Heuristics



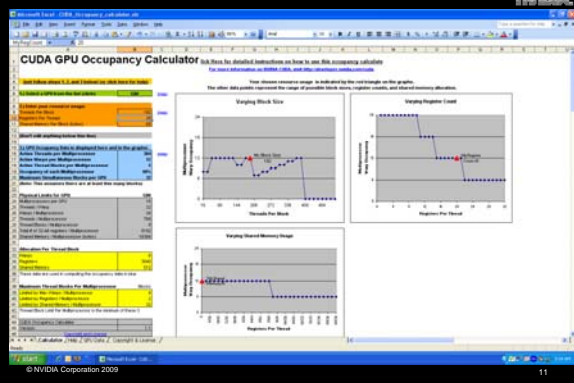
- # threads per block == 32 * n
 - Fully populate warps
- # threads per block >= 64
 - Avoid 8-block resource limit
- # threads per block <= 512 (384)
 - Continue hiding latencies while one block is doing `__syncthreads()`
- # threads per block == 64 * n
 - Help scheduler to avoid register memory bank conflicts

Blocks per Grid Heuristics



- # of blocks > # of multiprocessors
 - All multiprocessors busy
- # of blocks > # of multiprocessors * 2
 - Continue hiding latencies while one block is doing `__syncthreads()`
- # of blocks > 100 to scale to future devices
 - 1000 blocks per grid will scale across multiple generations

Occupancy Calculator



Parameterize Your Application



- Parameterization helps adaptation to different GPUs
- GPUs vary in many ways
 - # of multiprocessors
 - Memory bandwidth
 - Shared memory size
 - Register file size
 - Max. threads per block
- You can even make apps self-tuning (like FFTW and ATLAS)
 - "Experiment" mode discovers and saves optimal configuration

Outline



- Execution Configuration Optimizations
- **Instruction Optimizations**
- Memory Optimizations
 - Data transfers between host and device
 - Device memory optimizations

Arithmetic Instruction Throughput



- **int / float: add, shift, min, max, float: mul, mad**
 - 4 cycles per instruction per warp
- **Others are more expensive**
 - int multiply (32-bit)
 - `sinf(x)`, `expf(x)`, `powf(x,y)` (5 ulp or less)
- **Use fast math intrinsics**
 - `__mul24()` / `__umul24()`
 - `__sinf(x)`, `__expf(x)`, `__powf(x,y)`
 - Force intrinsics with `-use_fast_math` compiler option
- **See programming guide for details**

GPU results may not match CPU



- **Many variables: hardware, compiler, optimizations**
- **CPU operations aren't strictly limited to 0.5 ulp**
 - Operation sequence on 80-bit extended precision ALUs
- **Floating-point arithmetic is not associative!**
 - Parallelization potentially changes order of operations
- **Generally IEEE-754 compliant**
 - See programming guide for details

Control Flow Instructions



- **Divergence: threads of a warp take different branch**
 - Different execution paths must be serialized
 - Aim for warp branch granularity:

```
if (threadIdx.x / WARP_SIZE > 2)
```
- **Branch predication**
 - Remove branch, schedule all execution paths
 - Write results only for threads which meet branch condition

Outline



- Execution Configuration Optimizations
- Instruction Optimizations
- Memory Optimizations
 - **Data transfers between host and device**
 - Device memory optimizations

Host-Device Data Transfers



- **Device to host memory bandwidth much lower than device to device bandwidth**
 - 8 GB/s peak (PCI-e x16 Gen 2) vs. 141 GB/s peak (GTX 280)
- **Minimize transfers**
 - Intermediate data can be allocated, operated on, and deallocated without ever copying them to host memory
- **Group transfers**
 - One large transfer much better than many small ones
- **See the "bandwidthTest" CUDA SDK sample**

Overlapping Data Transfers and Computation



- **Overlap CPU computation with data transfers**
 - Allocate **pinned** host memory with "cudaMallocHost"

```

cudaMemcpyAsync(a_d, a_h, size, dir, 0);
kernel<<grid, block>>(a_d);
cpuFunction();
    
```

overlapped
- **Overlap GPU computation with data transfers**
 - Compute capability 1.1

```

cudaMemcpyAsync(dst, src, size, dir, stream1);
kernel<<grid, block, 0, stream2>>(...);
    
```

overlapped
- **stream** = sequence of operations that execute in order on the GPU
- stream-0 blocks and cannot be overlapped

© NVIDIA Corporation 2009

19

GPU/CPU Synchronization



- **Context based: cudaThreadSynchronize()**
 - Blocks until issued CUDA calls from a CPU thread complete
- **Stream based: cudaStreamSynchronize(stream)**
 - Blocks until CUDA calls issued to given stream complete
- **Stream based using events**
 - Insert events into stream, **record** when it reaches GPU

```

cudaEventRecord(event, stream)
    
```

 - Block until given event is recorded

```

cudaEventSynchronize(event)
    
```
- **Non-blocking polling versions: cudaXXQuery()**

© NVIDIA Corporation 2009

20

Zero copy



- **Access host memory directly from device code**
 - Transfers implicitly performed as needed by device code
 - Easier than asynchronous transfers
 - Faster if data is only read/written once
 - Always beneficial for integrated devices
- **Translate pinned memory pointer to device pointer**
 - `cudaHostGetDevicePointer`
 - Pass result to kernel

© NVIDIA Corporation 2009

21

Zero copy considerations



- Zero copy will always be a win for integrated devices that utilize CPU memory (you can check this using the **integrated** field in `cudaDeviceProp`)
- Zero copy will be faster if data is only read/written from/to global memory once, for example:
 - Copy input data to GPU memory
 - Run one kernel
 - Copy output data back to CPU memory
- Potentially easier and faster alternative to using `cudaMemcpyAsync`
 - For example, can both read and write CPU memory from within one kernel
- Note that current devices use pointers that are 32-bit so there is a limit of **4GB per context**

© NVIDIA Corporation 2009

22

Outline



- Execution Configuration Optimizations
- Instruction Optimizations
- Memory Optimizations
 - Data transfers between host and device
 - **Device memory optimizations**

© NVIDIA Corporation 2009

23

Theoretical Bandwidth



- **Device bandwidth of GTX 280**
 - $1.1 \text{ GHz} * (512\text{b} / 8) * 2 = 131.9 \text{ GB/s}$
 - Memory clock
 - Memory interface
 - DDR
- **Instruction throughput of GTX 280**
 - $1.3 \text{ GHz} * 240 = 311 \text{ GFLOPS}$
 - Peak (FMAD + MUL): 933 GFLOPS
- Arithmetic intensity should be larger than 10

© NVIDIA Corporation 2009

24

Coalescing

- Global memory requests per half-warp (16 threads)
- Compute capability ≤ 1.1 : aligned, in-order 64B requests
 - 1 transaction
 - 16 transactions
 - 16 transactions
- Compute capability ≥ 1.2 : aligned 32B, 64B, 128B requests
 - 2 transactions - 64B and 32B segment

© NVIDIA Corporation 2009 25

Coalescing Examples

```

global void offsetCopy(float *odata, float *idata, int offset)
{
    int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;
    odata[xid] = idata[xid];
}

```

- GTX 280
 - Compute capability 1.3
 - Peak bandwidth of 141 GB/s
- FX 5600
 - Compute capability 1.0
 - Peak bandwidth of 77 GB/s

© NVIDIA Corporation 2009 26

Coalescing Examples

```

global void strideCopy(float *odata, float *idata, int stride)
{
    int xid = (blockIdx.x * blockDim.x + threadIdx.x) * stride;
    odata[xid] = idata[xid];
}

```

- Strides are common and generally large
- Avoid using *shared memory* or *texture access*

© NVIDIA Corporation 2009 27

Shared Memory

- ~Hundred times faster than global memory
- Cache data to reduce global memory accesses
- Threads can cooperate via shared memory
- Use it to avoid non-coalesced access
 - Stage loads and stores in shared memory
 - Reorder access to allow coalescing

© NVIDIA Corporation 2009 28

Shared Memory Architecture

- Many threads accessing memory
 - Therefore, memory is divided into **banks**
 - Successive 32-bit words assigned to successive banks
- Each bank can service one address per cycle
 - 16 banks to service 16 threads per clock (ALUs are clocked twice as fast as shared memory)
- Multiple simultaneous accesses to a bank result in a **bank conflict**
 - Conflicting accesses are serialized

© NVIDIA Corporation 2009 29

Bank Addressing Examples

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

© NVIDIA Corporation 2009 30

Bank Addressing Examples

- 2-way Bank Conflicts
 - Linear addressing stride=2
- 8-way Bank Conflicts
 - Linear addressing stride=8

© NVIDIA Corporation 2009 31

Shared memory bank conflicts

- Shared memory as fast as registers **if there are no bank conflicts**
 - all threads of a half-warp access **different banks**
 - all threads of a half-warp read the **identical address** (broadcast)
- warp_serialize** profiler signal reflects conflicts
 - multiple threads in the same half-warp access the same bank
 - Bank conflict, must serialize the accesses
 - Cost = max # of simultaneous accesses to a single bank**

© NVIDIA Corporation 2009 32

Shared Memory Example: Transpose

- Each thread block works on a tile of the matrix
- Naïve implementation exhibits strided access to global memory

© NVIDIA Corporation 2009 33

Naïve Transpose

- Loads are coalesced, stores are not (strided by height)

```

__global__ void transposeNaive(float *odata, float *idata,
                               int width, int height)
{
    int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;

    int index_in = xIndex + width * yIndex;
    int index_out = yIndex + height * xIndex;

    odata[index_out] = idata[index_in];
}

```

© NVIDIA Corporation 2009 34

Coalescing through shared memory

- Access columns of a tile in shared memory to write contiguous data to global memory
- Requires `__syncthreads()` since threads access data in shared memory stored by other threads

© NVIDIA Corporation 2009 35

Coalescing through shared memory

```

__global__ void transposeCoalesced(float *odata, float *idata,
                                   int width, int height)
{
    dim3 tileIdx(blockIdx.x * TILE_DIM, blockIdx.y * TILE_DIM);

    int inIdx = tileIdx.x + threadIdx.x +
                (tileIdx.y + threadIdx.y) * width;
    int outIdx = tileIdx.y + threadIdx.x +
                 (tileIdx.x + threadIdx.y) * height;

    __shared__ float tile[TILE_DIM][TILE_DIM];

    // contiguous          coalesced
    tile[threadIdx.y][threadIdx.x] = idata[inIdx];

    __syncthreads();

    // coalesced          16-way bank conflict
    odata[outIdx] = tile[threadIdx.x][threadIdx.y];
}

```

© NVIDIA Corporation 2009 36

Coalescing through shared memory



```
__global__ void transposeCoalesced(float *odata, float *idata,
                                   int width, int height)
{
    dim3 tileIdx(blockIdx.x * TILE_DIM, blockIdx.y * TILE_DIM);

    int inIdx = tileIdx.x + threadIdx.x +
                (tileIdx.y + threadIdx.y) * width;
    int outIdx = tileIdx.y + threadIdx.x +
                 (tileIdx.x + threadIdx.y) * height;

    __shared__ float tile[TILE_DIM][TILE_DIM + 1];

    // contiguous          coalesced
    tile[threadIdx.y][threadIdx.x] = idata[inIdx];

    __syncthreads();

    // coalesced          no bank conflict
    odata[outIdx] = tile[threadIdx.x][threadIdx.y];
}

```

© NVIDIA Corporation 2009

37

Textures in CUDA

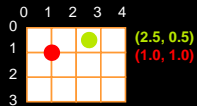


- Texture is an object for *reading* data
- Benefits:
 - Data is cached
 - Helpful when coalescing is a problem
 - Filtering
 - Linear / bilinear / trilinear interpolation
 - Dedicated hardware
 - Wrap modes (for "out-of-bounds" addresses)
 - Clamp to edge / repeat
 - Addressable in 1D, 2D, or 3D
 - Using integer or normalized coordinates

© NVIDIA Corporation 2009

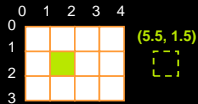
38

Texture Addressing



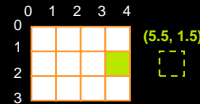
Wrap

- Out-of-bounds coordinate is wrapped (modulo arithmetic)



Clamp

- Out-of-bounds coordinate is replaced with the closest boundary



© NVIDIA Corporation 2009

39

CUDA Texture Types



- Bound to linear memory
 - Global memory bound to a texture
 - 1D, no filtering, no addressing modes
- Bound to CUDA arrays
 - Block linear CUDA array is bound to a texture
 - 1D, 2D, or 3D, filtering, addressing modes
 - No direct write from thread program (but D2D copy)
- Bound to pitch linear (CUDA 2.2)
 - Global memory address is bound to a texture
 - 2D, filtering, addressing modes

© NVIDIA Corporation 2009

40

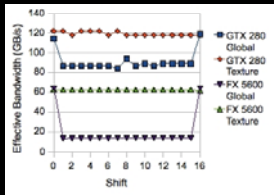
Texture Example



```
texture <float> texRef;

__global__ void textureShiftCopy(float *odata, float *idata, int shift)
{
    int xid = blockIdx.x * blockDim.x + threadIdx.x + shift;
    odata[xid] = tex1Dfetch(texRef, xid);
}

```



© NVIDIA Corporation 2009

41

Summary



- GPU hardware can achieve great performance on data-parallel computations if you follow a few simple guidelines:
 - Use parallelism efficiently
 - Coalesce memory accesses if possible
 - Take advantage of shared memory
 - Explore other memory spaces
 - Texture
 - Constant
 - Reduce bank conflicts

© NVIDIA Corporation 2009

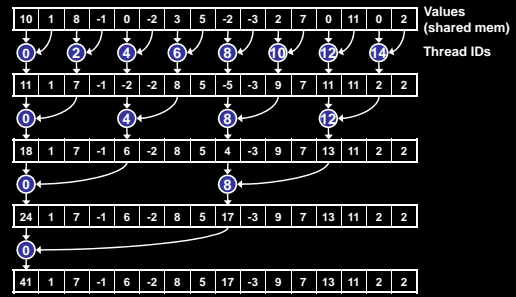
42

Outline



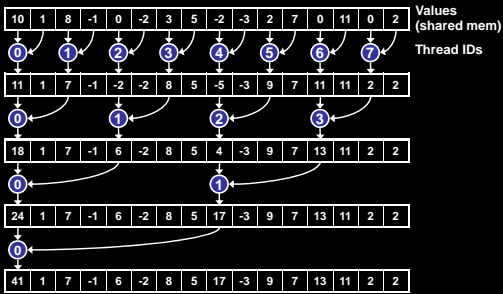
- SDK / Toolkit
- Execution Configuration Optimizations
- Instruction Optimizations
- Memory Optimizations
 - Data transfers between host and device
 - Device memory optimizations
- Example: Parallel Reduction

Parallel Reduction (1.0x)



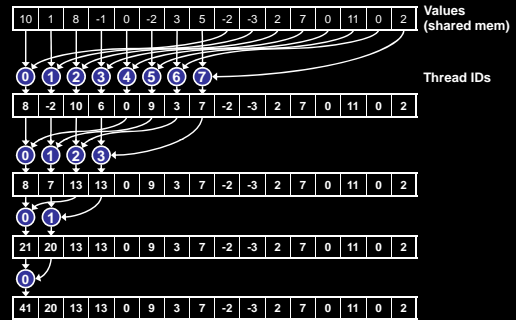
issue: divergent warps

Parallel Reduction (2.3x)



issue: bank conflicts

Parallel Reduction (4.7x)



Parallel Reduction (30x)



- Reduce idle warps
 - Preamble with serial reduction
 - Reduce multiple elements per thread
- Reduce branches
 - Unroll tail of loop where only 1 warp participates
 - Unroll everything (using templates)
- Optimized version is part of CUDPP

Graphics Hardware 07: Scan Primitives for GPU Computing
(Shubho Sengupta, Mark Harris, Yao Zhang, John Owens)