MIC-GPU: High-Performance Computing for Medical Imaging on Programmable Graphics Hardware (GPUs)

CUDA Memory Optimization

Klaus Mueller and Sungsoo Ha
Stony Brook University
Computer Science
Stony Brook, NY

Optimizing Memory Usage

Minimizing data transfers with low bandwidth
- Minimizing host & device transfer
- Maximizing usage of shared memory
- Re-computing can sometimes be cheaper than transfer

Organizing memory accesses based on the optimal memory access patterns
- Important for global memory access (low bandwidth)
- Shared memory accesses are usually worth optimizing only in case they have a high degree of bank conflicts

Global Memory Coalescing

Warp & global memory
- Threads execute by warp (32)
- Memory read/write by half warp (16)
- Global memory is considered to be partitioned into segments of size equal to 32, 64, or 128 bytes and aligned to these sizes.
- Block width must be divisible by 16 for coalescing
- Check your hardware (Compute Capability 1.x)
- Greatly improves throughput (Can yield speedups of >10)

Global Memory Coalescing

Compute Capability 1.0 or 1.1
- Aligned 64 or 128 bytes segment
- Sequential warp
- Divergent warp
- See some good patterns in CUDA document and CUDA SDK samples

Compute Capability 1.2 or higher
- 32, 64 or 128 bytes segment
- Any pattern as long as inside segment
Sobel Filter Effect

Before:  

After:  

Example: Sobel Filter

- Discrete convolution with Sobel mask

Input image

Filter mask

Output image

Ideally each thread will compute one output pixel

Output

R/W Global Memory

Bad access pattern

- Global memory only. No texture memory or shared memory. Hundreds of clock cycles, compared to 1 or 2 for reading from shared memory
- Unstructured read
- No cache, up to 12 global memory reads per thread

Reduce Global Memory Read

__device__ unsigned char ComputeSobel(

unsigned char ul,
unsigned char um,
unsigned char ur,
unsigned char ml,
unsigned char mm, //not used
unsigned char mr,
unsigned char ll,
unsigned char lm,
unsigned char lr,
float fScale) {

short Horz = ur + 2*mr + lr - ul - 2*ml - ll;
short Vert = ul + 2*um + ur - ll - 2*lm - lr;
short Sum = (short) ((fScale*(abs(Horz)+abs(Vert)));
if ( Sum < 0 ) return 0; else if ( Sum > 255 ) return 255;
return (unsigned char) Sum;
Reading Texture Memory

Take advantage of CUDA (texture memory)
- Using cache (texture memory) to enhance performance
- Each kernel can compute more than one pixels. This can help to exploit locality for cache
- Texture memory itself is optimized for coalescing

unsigned char *pSobel = (unsigned char *) (((char *) pSobelOriginal)+blockIdx.x*Pitch);
for ( int i = threadIdx.x; i < w; i += blockDim.x ) {
    unsigned char pix00 = tex2D( tex, (float) i-1, (float) blockIdx.x-1 );
    unsigned char pix01 = tex2D( tex, (float) i+0, (float) blockIdx.x-1 );
    unsigned char pix02 = tex2D( tex, (float) i+1, (float) blockIdx.x-1 );
    unsigned char pix10 = tex2D( tex, (float) i-1, (float) blockIdx.x+0 );
    unsigned char pix11 = tex2D( tex, (float) i+0, (float) blockIdx.x+0 );
    unsigned char pix12 = tex2D( tex, (float) i+1, (float) blockIdx.x+0 );
    unsigned char pix20 = tex2D( tex, (float) i-1, (float) blockIdx.x+1 );
    unsigned char pix21 = tex2D( tex, (float) i+0, (float) blockIdx.x+1 );
    unsigned char pix22 = tex2D( tex, (float) i+1, (float) blockIdx.x+1 );
    pSobel[i] = ComputeSobel(pix00, pix01, pix02, pix10, pix11, pix12,
                           pix20, pix21, pix22, fScale );}

Improve Caching?

Disadvantage
- Only using hardware cache to handle spatial locality
- A pixel may be still loaded 9 times in total due to cache miss

Take advantage of CUDA Shared Memory
- Shared memory can be as fast as register! As a user-controlled cache.
  1. Together with texture memory, load a block of the image into shared memory
  2. Each thread compute a consecutive rows of pixels (sliding window)
  3. Writing result to global memory

Returning Example : Sobel Filter

Computing all pixels inside one block (without apron)
Each thread will compute a number of consecutive rows of pixel

Applying vertical and horizontal masks

2 5 6 2 5 6
4 2 7 4 2 7
5 2 6 5 2 6

13 4 18 13
**Reading Shared Memory**

- Shared memory + texture memory.

```c
__shared__ unsigned char shared[];
kernelparallel<<<blocks, threads, sharedMem>>>(…);

// copy a large tile of pixels into shared memory
__syncthreads();
......// copy a large tile of pixels into shared memory
......// read 9 pixels from shared memory
out.x = ComputeSobel(pix00, pix01, pix02, pix10, pix11, pix12, pix20, pix21, pix22, fScale );
......//read p00, p10, p20
out.y = ComputeSobel(pix01, pix02, pix00, pix11, pix12, pix10, pix21, pix22, pix20, fScale );
......//read p01, p11, p21
out.z = ComputeSobel( pix02, pix00, pix01, pix12, pix10, pix11, pix22, pix20, pix21, fScale );
......//read p02, p12, p22
out.w = ComputeSobel( pix00, pix01, pix02, pix10, pix11, pix12, pix20, pix21, pix22, fScale );
__syncthreads();
```

**Shared Memory Bank Conflicts**

**Shared memory banks**

- Shared memory is divided into 32 banks to reduce conflicts
- Each thread can access 32-bit from different banks simultaneously to achieve high memory bandwidth
- Conflict-free shared memory as fast as registers
- Linear
  ```c
  shared__float shared[32];
  float data = shared[BaseIndex + 1* tid];
  ``
- Random
  ```c
  shared__float shared[32];
  float data = shared[BaseIndex + 4 * tid];
  ``

**Shared Memory Broadcasting**

**Compute Capability 1.x**

4-way bank conflicts

```c
__shared__ char shared[32];
char data = shared[BaseIndex + tid];
```

No bank conflicts

```c
char data = shared[BaseIndex + 4 * tid];
```

**Compute Capability 2.x**

- Bank conflicts occur when multiple threads access different words in the same bank
Returning Example: Sum of Numbers

Add up a large set of numbers
- Normalization factor:
  \[ S = \sum_{i=0}^{n-1} v[i] \]
- Mean square error:
  \[ MSE = \sum_{i=0}^{n-1} (a[i] - b[i])^2 \]

Number of addition operations:


n-1 additions → How to optimize?

Non-parallel approach

Input numbers:
\[ 10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2 \]

- Non-parallel approach:
  - Generate only one thread
  - \( O(n) \) additions

Rule 1: Maximized parallel execution

Parallel Approach: Kernel 1

- Interleaved addressing: Kernel 1
  - 16 threads in a half wrap.
  - Only 8 of them are active in the first loop

- \( O(\log n) \) additions

Rule 2: Optimize memory usage

Parallel Approach: Kernel 1

CUDA code:

```c
global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // do reduction in shared mem
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) {
            sdata[tid] += sdata[tid+s];
        }
        __syncthreads();
    }

    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Rule 3: Maximize instruction throughput

- Inefficient statement, % operator is very slow
- Writing with global memory coalescing
Parallel Approach: Kernel 2

Refinement strategy:

Just replace divergent branch in inner loop:

```c
for (unsigned int s=1; s < blockDim.x; s *= 2) {
    if (tid % (2^s)) == 0) {
        sdata[tid] == sdata[tid + s];
    }
    __syncthreads();
}
```

With strided index and non-divergent branch:

```c
for (unsigned int s=1; s < blockDim.x; s *= 2) {
    int index = 2^s * tid;
    if (index < blockDim.x) {
        sdata[index] == sdata[index + s];
    }
    __syncthreads();
}
```

Parallel Approach: Kernel 3

Sequential addressing: Kernel 3

CUDA code:

```
Just replace strided indexing in inner loop:
```

```
for (unsigned int s=1; s < blockDim.x; s *= 2) {
    int index = 2^s * tid;
    if (index < blockDim.x) {
        sdata[index] == sdata[index + s];
    }
    __syncthreads();
}
```

With reversed loop and threadID-based indexing:

```
for (unsigned int s=blockDim.x; s > 0; s>>=1) {
    if (tid < s) {
        sdata[tid] == sdata[tid + s];
    }
    __syncthreads();
}
```

Law 2: Optimize memory usage

Law 3: Maximize instruction throughput
Toward Final Optimized Kernel

Performance for 4M numbers:

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Time (2^20 ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3.456 ms</td>
<td>4.854 GB/s</td>
<td>2.33x</td>
<td>2.33x</td>
</tr>
<tr>
<td>3</td>
<td>1.722 ms</td>
<td>9.741 GB/s</td>
<td>2.01x</td>
<td>4.66x</td>
</tr>
<tr>
<td>4</td>
<td>0.965 ms</td>
<td>17.377 GB/s</td>
<td>1.78x</td>
<td>8.34x</td>
</tr>
<tr>
<td>5</td>
<td>0.536 ms</td>
<td>31.289 GB/s</td>
<td>1.8x</td>
<td>15.01x</td>
</tr>
<tr>
<td>6</td>
<td>0.381 ms</td>
<td>43.996 GB/s</td>
<td>1.41x</td>
<td>21.16x</td>
</tr>
<tr>
<td>7</td>
<td>0.268 ms</td>
<td>62.671 GB/s</td>
<td>1.42x</td>
<td>30.04x</td>
</tr>
</tbody>
</table>

Further Optimizations

Rule 3: Maximize instruction throughput

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Halve the number of blocks, with two loads</td>
</tr>
<tr>
<td>5</td>
<td>Unrolling last loop</td>
</tr>
<tr>
<td>6</td>
<td>Completely unrolling loops</td>
</tr>
<tr>
<td>7</td>
<td>Multiple element per thread</td>
</tr>
<tr>
<td></td>
<td>See details changes in M. Harris, Optimizing parallel reduction with CUDA</td>
</tr>
</tbody>
</table>

Towards Final Optimized Kernel

Performance for 4M numbers:

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Time (2^20 ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3.456 ms</td>
<td>4.854 GB/s</td>
<td>2.33x</td>
<td>2.33x</td>
</tr>
<tr>
<td>3</td>
<td>1.722 ms</td>
<td>9.741 GB/s</td>
<td>2.01x</td>
<td>4.66x</td>
</tr>
<tr>
<td>4</td>
<td>0.965 ms</td>
<td>17.377 GB/s</td>
<td>1.78x</td>
<td>8.34x</td>
</tr>
<tr>
<td>5</td>
<td>0.536 ms</td>
<td>31.289 GB/s</td>
<td>1.8x</td>
<td>15.01x</td>
</tr>
<tr>
<td>6</td>
<td>0.381 ms</td>
<td>43.996 GB/s</td>
<td>1.41x</td>
<td>21.16x</td>
</tr>
<tr>
<td>7</td>
<td>0.268 ms</td>
<td>62.671 GB/s</td>
<td>1.42x</td>
<td>30.04x</td>
</tr>
</tbody>
</table>

Final optimized kernel:

Best Programming Practices

Three basic strategies

- Maximize parallel execution
- Optimize memory usage → achieve maximum memory bandwidth
- Optimize instruction bandwidth → maximize instruction throughput

Maximized parallel execution

- Minimize number of synchronization barriers → let it flow
- Minimize divergent flows
- Better synchronize within a block than across → group threads

Optimize memory usage

- Map poor coalescing patterns in global memory to shared memory
- Load data in coalesced chunks before computation begins → #1: Get good coalescing in global memory
- → #2: Get conflict-free data access in shared memory
Maximize instruction throughput

1. Instruction level
   → operator, branches and loops
2. Optimize execution configuration

Kernel will fail to launch if
- Number of threads per block >> max number of threads per block
- Requires too many registers or shared memory than available

Block
- At least as many blocks as multiprocessors (SMs)

Threads
- Chose number of threads/block as a multiple of the warp size
- Typically 192 or 256 threads per block
- But watch out for required registers and shared memory
- Check Visual profiler or Occupancy Calculator

Multiprocessor occupancy
- Ratio of number of active warps per SM over max number of warps
- Visual profiler or Occupancy Calculator
  - Choose thread block size based on shared memory and registers