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

SPIE Medical Imaging

CUDA Memory Optimization

Klaus Mueller, Ziyi Zheng, Eric Papenhausen

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

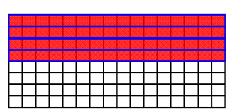
Organizing memory accesses based on the optimal memory access patterns

- 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

Warps & global memory

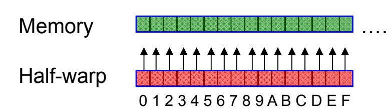
Block



- 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

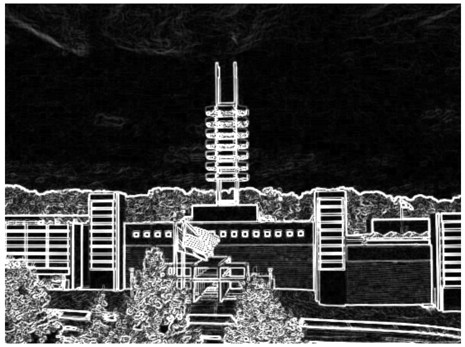
Compute Capability 1.2 or higher

- 32, 64 or 128 bytes segment
- Any pattern as long as inside segment

Sobel Filter Effect

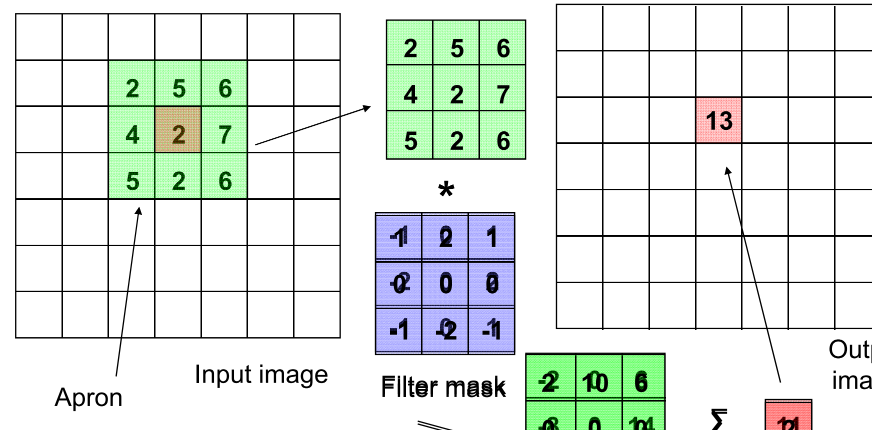
Before:

After:



Example: Sobel Filter

- Discrete convolution with Sobel mask

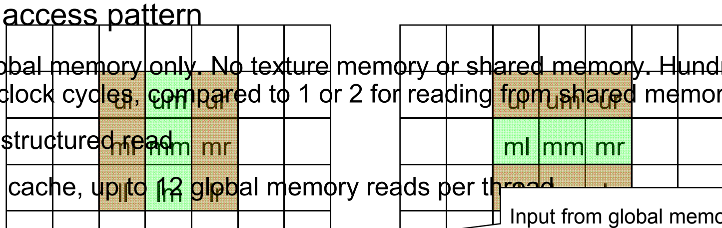


Ideally each thread will compute one output pixel

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



Input from global memory

Output to another global memory

```
__global__ void
SobelBadKernel(unsigned char* Input, unsigned char* output,unsigned int width, unsigned int
   height)
{
….//calculate the index for ur, ul, um, ml, mr, ll, lm, lr.
 float Horz=Input[ur] +Input[lr] +2.0*Input[mr] -2.0*Input[ml] -Input[ul] -Input[ll] ;
 float Vert=Input[ur] +Input[ul] +2.0*Input[um] -2.0*Input[lm] -Input[ll] -Input[lr] ;
 output[resultindex]   = abs(Horz)+abs(Vert);
}
```


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;}
```

2	5	6
4	2	7
5	2	6

Reduce 12 reads into 8 reads

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

Reading Texture Memory

- Texture memory only.
- No shared memory

```
unsigned char *pSobel = (unsigned char *) (((char *) pSobelOrig
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 );}
```

Global memory as output. Need consider coalescing when write back

Read from texture memory

Improve Caching?

Disadvantage

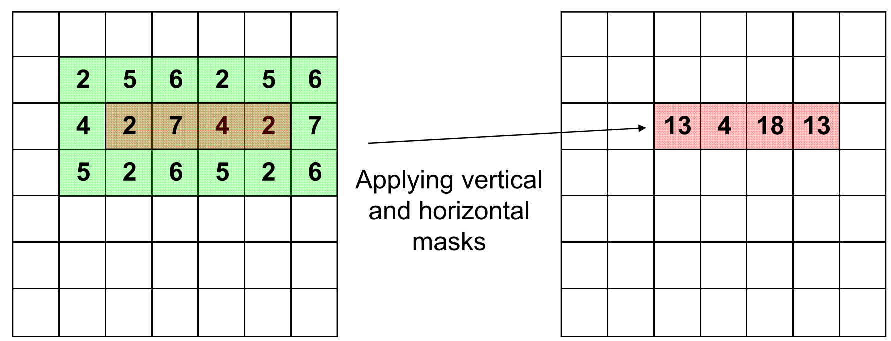
- Only using hardware cache to handle spatial locality
- A pixel may be still loaded 9 times in total due to cache miss

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



Each thread will compute a number of consecutive rows of pixel

Computing all pixels inside one block (without apron)

Reading Shared Memory

- Shared memory + texture memory.

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

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

```
……// copy a large tile of pixels into shared memory
  __syncthreads();
……// 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();
```

Loading data under current window, 9 reads

Sliding window right, reuse 6, update 3

Sliding window right, reuse 6, update 3

Sliding window right, reuse 6, update 3

Shared Memory Bank Conflicts

Shared memory banks

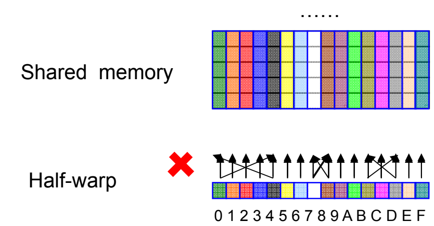
- Shared memory is divided into 32 banks to reduce conflicts
- Each thread can access 32-bit from different banks simultaneously to achieve high memory bandwidth
- Conflict-free shared memory as fast as registers
- Linear

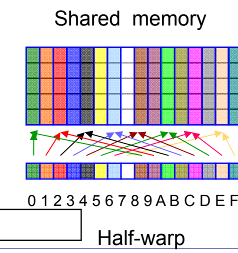
```
shared__ float shared[32];
float data = shared[BaseIndex + 1* tid];
```

Thread ID

1, 3, 5 ,7 …… (Any odd number)

- Random





Shared Memory Bank Conflicts

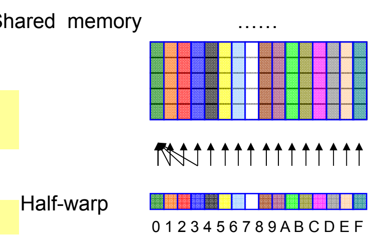
Compute Capability 1.x

4-way bank conflicts

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

No bank conflicts

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



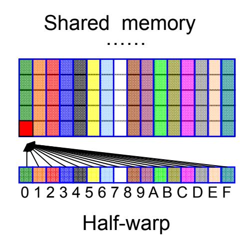
Compute Capability 2.x

- Bank conflicts occur when multiple threads access different words in the same bank

Shared Memory Broadcasting

Shared memory read a 32-bit word and broadcast to several threads simultaneously

- Read
- Reduce or resolve bank conflicts if set to broadcasting
- Which word is selected as the broadcast word and which address is picked up for each bank at each cycle is unspecified



Returning Example : Sum of Numbers

SPIE Medical Imaging

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:

$$\underbrace{v[0] + v[1] + v[2] + \cdots + v[n-1]}_{\text{n-1 additions}}$$

n-1 additions → How to optimize?

Non-parallel approach

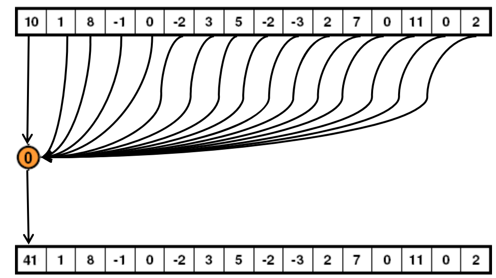
SPIE Medical Imaging

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



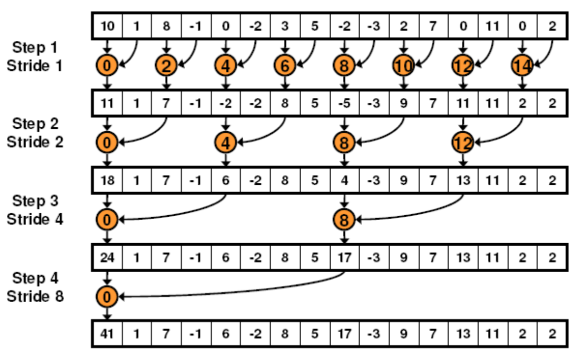
O(n) additions

Rule 1 : Maximized parallel execution

Parallel Approach: Kernel 1

SPIE Medical Imaging

- Interleaved addressing: Kernel 1



16 threads in a half wrap. Only 8 of them are active in the first loop

Uncoalesced global memory reading and writing pattern

O(logn) additions

Rule 2 : Optimize memory usage

Parallel Approach: Kernel 1

SPIE Medical Imaging

CUDA code:

```
__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

SPIE Medical Imaging

Refinement strategy:

Just replace divergent branch in inner loop:

```
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:

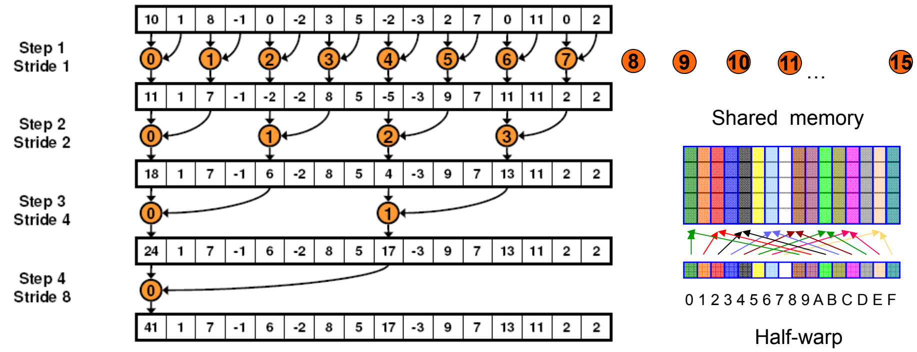
```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
   int index = 2 * s * tid;

   if (index < blockDim.x) {
      sdata[index] += sdata[index + s];
   }
   __syncthreads();
}
```


Parallel Approach: Kernel 2

SPIE Medical Imaging

Interleaved addressing: Kernel 2



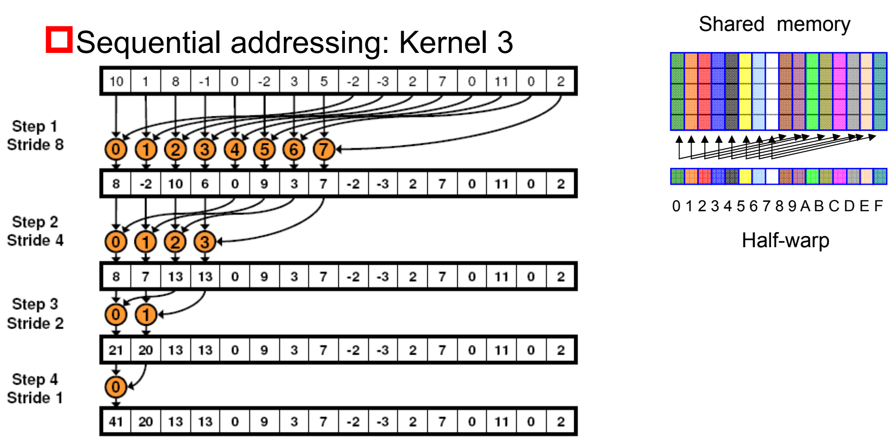
Shared memory conflict !

Law 2 : Optimize memory usage

Parallel Approach: Kernel 3

SPIE Medical Imaging

Sequential addressing: Kernel 3



Conflict-free

Parallel Approach: Kernel 3

SPIE Medical Imaging

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();
}
```

Every thread is active in first loop interation

With reversed loop and threadID-based indexing:

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

Half of the threads are idle in first loop iteration! Wasteful!

Law 3: Maximize instruction throughput

Toward Final Optimized Kernel

SPIE Medical Imaging

Performance for 4M numbers:

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x

Further Optimizations

SPIE Medical Imaging

Kernel 4 — Rule 3: Maximize instruction throughput

- Halve the number of blocks, with two loads

Kernel 5

- Unrolling last loop

Kernel 6

- Completely unrolling loops

Kernel 7

- Multiple element per thread
- See details changes in

  M. Harris, Optimizing parallel reduction with CUDA

Towards Final Optimized Kernel

SPIE Medical Imaging

Performance for 4M numbers:

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x
Kernel 6: completely unrolled	0.381 ms	43.996 GB/s	1.41x	21.16x

Final optimized kernel:

Kernel 7: multiple elements per thread	0.268 ms	62.671 GB/s	1.42x	30.04x

Best Programming Practices

SPIE Medical Imaging

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

Best Programming Practices

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

Best Programming Practices

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

Course Schedule

1:30 – 1:45:	Introduction
1:45 – 2:00:	Parallel programming primer
2:00 – 2:15:	GPU hardware
2:15 – 3:00:	CUDA API, threads level optimization
	Coffee Break
3:30 – 4:00:	CUDA memory optimization
4:00 – 4:15:	CUDA programming environment (Ziyi)
4:15 – 4:45:	Parallelism in medical image (Klaus)
4:45 – 5:25:	CT reconstruction examples (Eric + Ziyi)
5:25 – 5:30:	Closing remarks (Klaus)