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

SPIE Medical Imaging

Introductory Code Examples

Klaus Mueller
Stony Brook University
Computer Science
Stony Brook, NY

Fang Xu
Siemens USA Research
Princeton, NJ

Outline

- Parallelism → Sum of Many Numbers
  - Non-parallel Approach
  - Tree-based Approaches
- Memory Access → Sobel Filter
  - R/W Global Memory
  - Reduce Memory Read
  - Read Shared Memory

Parallelism

Sum of Many Numbers

Sum of Many Numbers

Adding up a large set of numbers is common:

- Normalization factor:

$$S = v_1 + v_2 + \cdots + v_n$$

- Mean square error:

$$MSE = \frac{(a_1 - b_1)^2 + \cdots + (a_n - b_n)^2}{n}$$

- L2 Norm:

$$\|\vec{x}\| = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$$

Sum of Many Numbers

Common operator:

$$\sum: \quad v_1 \oplus v_2 \oplus v_3 \oplus \cdots \oplus v_n$$

O(n) additions

Code in C++ running on CPU:

```
float sum = 0;
for (int i=0; i<n; i++)
{
    sum += v[i];
}
return sum;
```

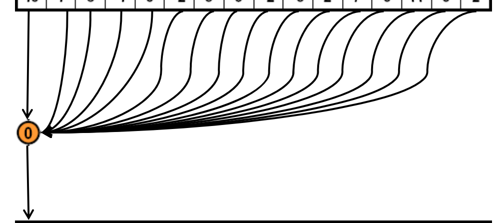

Non-Parallel Approach

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



Memory space

41	1	8	-1	0	-2	3	5	-2	-3	2	7	0	11	0	2

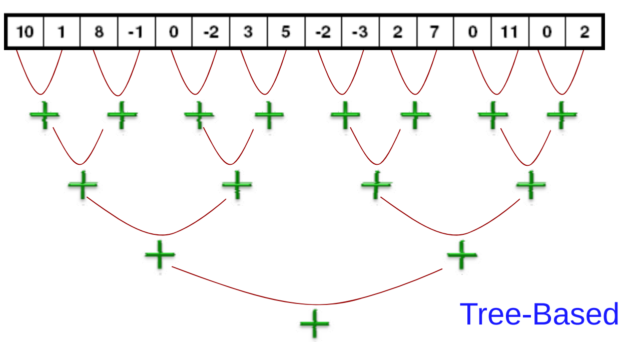
O(n) complexity → How to optimize?

Parallel Approach

Two tasks:

- read numbers to memory
- do the computation (addition) and write result

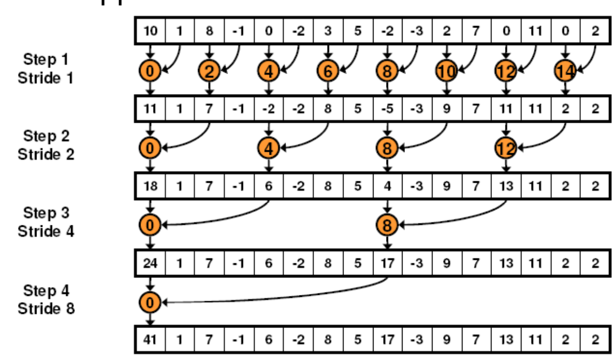
a + b



Tree-Based Approach

Tree-based Approach: Kernel 1

- Parallel Approach: Kernel 1



Generate 16 threads

Threads in same step execute in parallel

O(logn) complexity

Tree-based Approach: Kernel 1

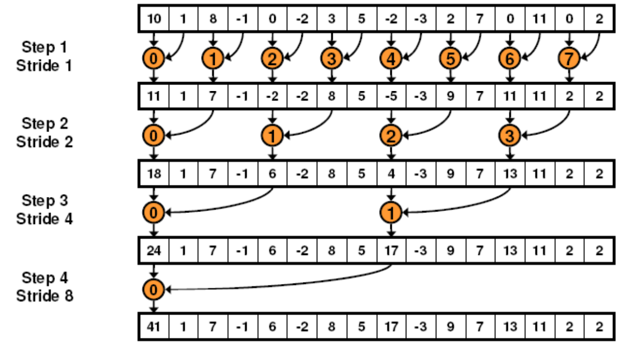
CUDA code:

```
__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];
}
```

very inefficient statement,
% operator is very slow

Tree-based Approach: Kernel 2

- Improved Parallel Approach: Kernel 2



Tree-based Approach: Kernel 2

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();
}
```

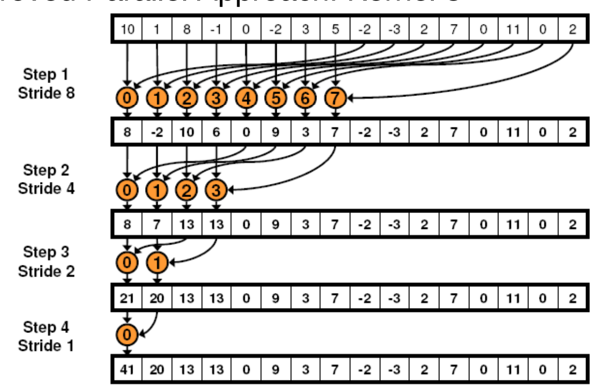
With strided index and non-divergent branch:

```
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();
}
```

Shared memory
bank conflict !

Tree-based Approach: Kernel 3

- Improved Parallel Approach: Kernel 3



Conflict-free

Tree-based 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();
}
```

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();
}
```


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

Final optimized kernel:

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

Parallel Reduction

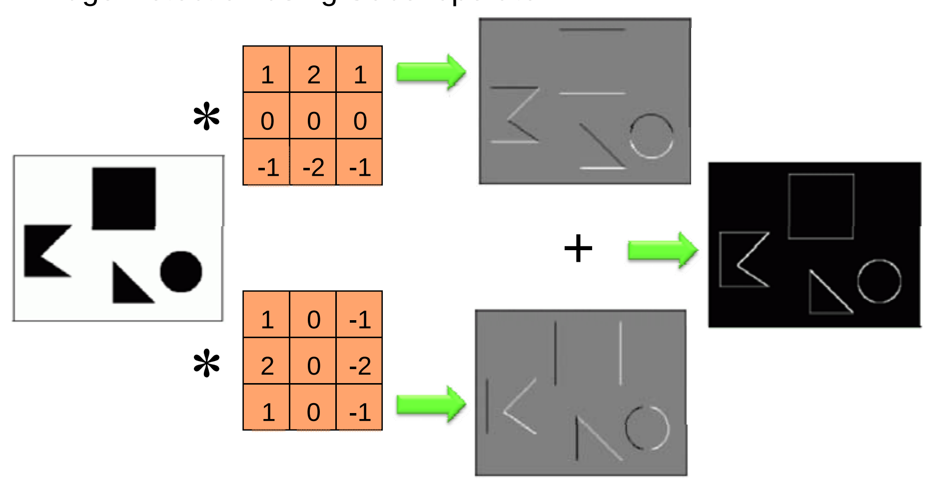
Memory Access

Sobel Filter

Sobel Filter

SPIE Medical Imaging

Edge Detection using Sobel operator



Sobel Filter Implementation

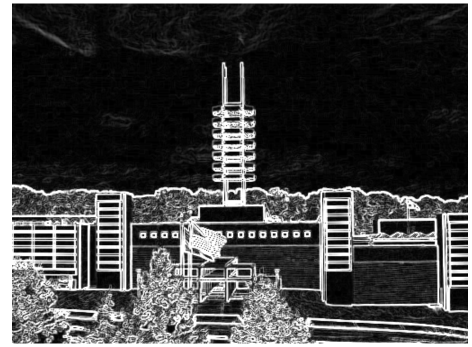
- A discrete differentiation operator: approximates the gradient of the intensity image
- On CPU in C++:

```
float dx[9] = {1, 0, -1, 2, 0, -2, 1, 0, -1};        float dy[9] = {1, 2, 1, 0, 0, 0, -1, -2, -1};
for (int j=0; j<pic_h; j++)
    for (int i=0; i<pic_w; i++) {
          float sumx = 0, sumy = 0;
          for(int n = -1; n <= 1; n++)
             for(int m = -1; m <= 1; m++) {
                   float temp = img[(pic_h - j - 1 + n)*pic_w + i + m];
                   sumx += temp*dx[(n+1)*3 + (m+1)];
                   sumy += temp*dy[(n+1)*3 + (m+1)];
             }
          edge[(pic_h - j - 1)*pic_w + i] = abs(sumx) + abs(sumy);
    }
```


Sobel Filter Effect

Before:

After:



Memory Access

Sobel filter with:

- Global memory
- Texture memory
- Shared memory

More details to follow….

1: R/W Global Memory

- Global memory only
- Up to 12 global memory reads per thread
- Each thread computes one pixel

ul	um	ur
ml	mm	mr
ll	lm	lr

```
__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);
}
```

2: Reduce Memory Read

SPIE Medical Imaging

- Read from texture memory
- Reduce 12 reads to 9 reads per thread
- Each thread computes one pixel

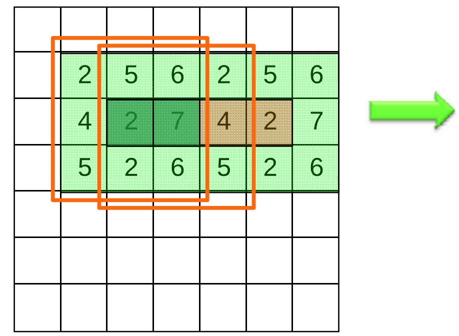
ul	um	ur
ml	mm	mr
ll	lm	lr

```
__device__ unsigned char ComputeSobel(
        unsigned char ul, unsigned char um,  unsigned char ur,  unsigned char ml,
        unsigned char mm, 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;
}
```


3: Read Shared Memory

SPIE Medical Imaging

- Consecutive rows of pixels share common pixels around

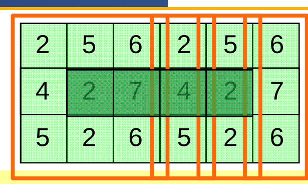


Compute 4 pixels by each thread

3: Read Shared Memory

SPIE Medical Imaging

CUDA code:




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

```
……// 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();
```


Memory Access

SPIE Medical Imaging

Sobel filter with:

- Global memory
- Texture memory
- Shared memory

Conclusion…

Kernel optimization

Kernel optimization

Kernel optimization

Kernel optimization

Course Schedule

1:30 – 1:45: Introduction (KM)

1:45 – 2:15: Introductory code examples (KM)

2:15 – 2:30: Parallel programming primer (KM)

2:30 – 3:00: Parallelism in CT reconstruction (FX)

Coffee Break

3:30 – 3:45: GPU hardware (KM)

3:45 – 4:30: CUDA API, threads, memory, performance optimization (KM)

4:30 – 4:45: CUDA programming environment (FX)

4:45 – 5:25: CT reconstruction examples (FX, KM)

5:25 – 5:30: Closing remarks (KM, FX)