Parallelism

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:
  \[ \|\mathbf{x}\| = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2} \]
**Sum of Many Numbers**

Common operator:

\[ \sum: v_1 + v_2 + v_3 + \cdots + v_n \]

\(O(n)\) additions

Code in C++ running on CPU:

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

**Non-Parallel Approach**

Input numbers:

\[ \begin{array}{cccccccccccccc}
10 & 1 & 8 & -1 & 0 & -2 & 3 & 5 & -2 & -3 & 2 & 7 & 0 & 11 & 0 & 2
\end{array} \]

Non-parallel approach:

- Generate only one thread

How to optimize?

**Parallel Approach**

Two tasks:

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

\[ a + b \]

**Tree-based Approach: Kernel 1**

Generate 16 threads

Threads in same step execute in parallel

\(O(\log n)\) complexity
Tree-based Approach: Kernel 1

CUDA code:

```c
__global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[16];

    // 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];
}
```

- This code snippet is very inefficient due to the slow `%` operator. The `%` operator is very slow because it performs a division and modulo operation, which is computationally expensive on CPUs and GPUs.

- The code optimizes the reduction operation by using shared memory to store the intermediate results. This significantly reduces the number of memory accesses and improves performance.

Tree-based 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();
}
```

- Shared memory bank conflict!

- Conflict-free
Tree-based Approach: Kernel 3

CUDA code:

Just replace strided Indexing in inner loop:

```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();
}
```

With reversed loop and threadIdx-based indexing:

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

Toward Final Optimized Kernel

Performance for 4M numbers:

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Time (2^20 int)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel 1</td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 2</td>
<td>3.456 ms</td>
<td>4.654 GB/s</td>
<td>2.33x</td>
<td>2.33x</td>
</tr>
<tr>
<td>Kernel 3</td>
<td>1.722 ms</td>
<td>9.741 GB/s</td>
<td>2.01x</td>
<td>4.68x</td>
</tr>
</tbody>
</table>

Final optimized kernel:

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

Sobel Filter

Edge Detection using Sobel operator

```
1 2 1
0 0 0
-1 -2 -1
```
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_w; j++)
    for (int i=0; i<pic_h; i++)
      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…. 

Memory Access

1: R/W Global Memory

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

```
__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.
  output[resultindex] = abs(Horz)+abs(Vert);
}
```
2: Reduce Memory Read

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

```c
__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

- Consecutive rows of pixels share common pixels around

```
...... // 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 );
    out.y = ComputeSobel( pix01, pix02, pix00, pix11, pix12, pix10, pix21, pix22, pix20, fScale );
    out.z = ComputeSobel( pix02, pix00, pix01, pix12, pix10, pix11, pix22, pix20, pix21, fScale );
    out.w = ComputeSobel( pix00, pix01, pix02, pix10, pix11, pix12, pix20, pix21, pix22, fScale );
    __syncthreads();
```

Memory Access

- Sobel filter with:
  - Global memory
  - Texture memory
  - Shared memory
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)