Histograms & Sparse Matrix Multiplications

General-purpose Programming of Massively Parallel Graphics Processors
Shiraz University, Spring 2010
Instructor: Reza Azimi

The slides are primarily adapted from:
Andreas Moshovos’ Course at UofT

Histogram

- A Diagram showing the distribution of values
- Example: Grades Histogram

Grades

Frequency

GPU Programming Shiraz University, Winter 88/Spring 88, Reza Azimi
Histograms in Image Processing

Sequential Algorithm

\[
\text{for } (i = 0; i < \text{BIN\_COUNT}; i++) \\
\quad \text{bin}[i] = 0;
\]

\[
\text{for } (i = 0; i < N; i++) \\
\quad \text{bin[data[i]]++;
}\]

- Complexity: \(O(N)\), where \(N\) is the size of the input data
- Assuming: \(0 \leq \text{data[i]} < \text{BIN\_COUNT}\)
Parallel Strategy

- Distribute work across multiple blocks
  - Divide input data to blocks

- Each block process its own portion
  - One pixel per thread
  - Multiple pixels per thread

- Partial histograms
  - Multiple histograms, each for a part of the image (input data)

- Merge all partial histograms
  - Simply add all the histograms entry-by-entry

Data Structures and Access Patterns

- data[]:
  - Accessed sequentially
  - Each element accessed only once
  - Potentially huge in size (multiple millions of entries)

- result[]:
  - Access is data-dependent
  - Each element may be accessed multiple times
  - Fairly small (tens to hundreds)

Data Allocation Strategy:
- data[] in global memory (why?)
- result[] in shared memory (why?)
Sub-Histograms

- How many sub-histograms can we fit in shared memory?
  - Input value range: 0-255, 1 byte
  - Each histogram needs 256 entries
  - How many bytes per entry?
    - That’s data dependent
    - Let’s assume 32-bits or 4 bytes

- 16KB (shared memory) / (256 x 4) (histogram)
  - 16 sub-histograms at any given point of time

- Let’s try one histogram per block

---

Partial Histogram Data Structure

- s_Hist[]: An array in shared memory

- Each entry is 4 bytes (maximum number of pixels with that value)

- One partial histogram per thread block
Algorithm Overview

1. Initialize partial histogram
   - Each thread:
     - \( s_{\text{Hist}}[\text{index}] = 0 \)
     - index += threads per block

2. Update local histogram
   - Each thread:
     - \( s_{\text{Hist}}[\text{data[index]}]++ \)
     - index += Total number of threads

3. Update global histogram
   - Each thread
     - \( g_{\text{Hist}}[\text{index}] += s_{\text{Hist}}[\text{index}] \)
     - index += threads per block

Updates to the Histograms

- **Case 1:** Threads in a block: update \( s_{\text{Hist}}[] \)
  - \( s_{\text{Hist}}[\text{data[index]}]++ \)

- **Case 2:** All threads: update global histogram
  - \( g_{\text{Hist}}[\text{index}] += s_{\text{Hist}}[\text{index}] \)

These are \texttt{read-modify-write} sequences

```c
var++;
```
compiled to

1. \texttt{register = var;}
2. \texttt{register++;}
3. \texttt{var = register;}

Simultaneous Updates?

thread1

\[
\begin{align*}
\text{r1:10} & \quad \text{r1 = var;} \\
\text{r1:11} & \quad \text{r1++;} \\
\text{var:11} & \quad \text{var = r1;} \\
\end{align*}
\]

\[
\begin{align*}
\text{var:11} & \quad \text{var++;} \\
\end{align*}
\]

thread2

\[
\begin{align*}
\text{r2:10} & \quad \text{r2 = var;} \\
\text{r2:11} & \quad \text{r2++;} \\
\text{var:11} & \quad \text{var = r2;} \\
\end{align*}
\]

An update is lost! The correct value should be 12!

Atomic Operations

- Read-Modify-Write operations that are guaranteed to happen “atomically”
  - Produces the same result as if the sequence executed in isolation in time
  - Think of it as “serializing the execution” of all atomics
  - This is not what happens – This is how you should think about them
Serializing Read-Modify-Updates

Serializing updates in parallel threads can lead to inconsistent results. Consider two threads `thread1` and `thread2` updating a variable `var`.

```
thread1
var++;       thread2
var++;       var = var;
```

Here, `var` is updated in parallel by both threads. To ensure consistency, we can use serializing operations.

```
r1 = var;   r1:10
r1++;       r1:11
var = r1;   var:11
```

Thread `thread1` reads `var`, increments it, and stores the result back. Similarly, `thread2` reads `var`, increments it, and stores the result back. This ensures that the updates are serialized.

Atomic Operations

Supported both in Shared and Global memory

Example:

```c
atomicAdd (int *pointer, int value)
    *pointer += value (atomically)
```

- **Atomic Operations**
  - Add, Sub, Inc, Dec
  - Exch, Min, Max, CAS
  - Bitwise: And, Or, Xor

- Work with (unsigned) integers
- `atomicExch` works with single-precision Floating Point as well
- Take a look at *NVIDIA CUDA Programming Guide* for further details
Histogram: CUDA Implementation

```c
__global__ void histogram256Kernel (uint *d_Result, uint *d_Data,
   int dataN)
{
   int globalTid =
      blockIdx.x * blockDim.x + threadIdx.x;

   // Total number of threads in the compute grid
   int numThreads = blockDim.x * gridDim.x;

   __shared__ uint s_Hist[BIN_COUNT];
   ..
   ..
```

Clearing Partial Histogram

```c
// Clear shared memory buffer
// for current thread block before processing
for ( int pos = threadIdx.x;
   pos < BIN_COUNT;
   pos += blockDim.x) {
   s_Hist[pos] = 0;
}
__syncthreads ();
.. ..
Generate Partial Histograms

```c
for (int pos = globalTid;
    pos < dataN;
    pos += numThreads) {
    uint data4 = d_Data[pos];
    atomicAdd (s_Hist + (data4 >> 0) & 0xFFU, 1);
    atomicAdd (s_Hist + (data4 >> 8) & 0xFFU, 1);
    atomicAdd (s_Hist + (data4 >> 16) & 0xFFU, 1);
    atomicAdd (s_Hist + (data4 >> 24) & 0xFFU, 1);
}
__syncthreads();
```

Merging Partial Histograms

```c
.. for (int pos = threadIdx.x;
    pos < BIN_COUNT;
    pos += blockDim.x) {
    atomicAdd(d_Result + pos, s_Hist[pos]);
}..
..`
```
Histogram: CUDA Implementation

```c
__global__ void histogram256Kernel (uint *d_Result, uint *d_Data, int dataN)
{
    int globalTid = blockIdx.x * blockDim.x + threadIdx.x;

    // Total number of threads in the compute grid
    int numThreads = blockDim.x * gridDim.x;
    __shared__ uint s_Hist[BIN_COUNT];

    // Clear shared memory buffer
    // for current thread block before processing
    for (int pos = threadIdx.x; pos < BIN_COUNT; pos += blockDim.x) {
        s_Hist[pos] = 0;
    }
    __syncthreads();

    for (int pos = threadIdx.x; pos < BIN_COUNT; pos += blockDim.x)
        atomicAdd(d_Result + pos, s_Hist[pos]);
}
```

Discussion

- **s_Hist updates**
  - Conflicts in shared memory
  - Data dependent
  - 16-way conflicts possible
    - Why is it likely to happen?

- Any alternatives?
  - One histogram per thread?
  - Load data in shared memory
    - Each thread produces a portion of the s_Hist that maps onto the same bank?
  - Take a look at the histogram64 and histogram256 examples in the NVIDIA CUDA SDK.
Warp Vote Functions

```c
int __all (int predicate);
```
Evaluates `predicate` for all threads of the warp and returns non-zero if and only if `predicate` evaluates to non-zero for all of them.

```c
int __any (int predicate);
```
Evaluates `predicate` for all threads of the warp and returns non-zero if and only if `predicate` evaluates to non-zero for any of them.

---

Warp Vote Functions: Example

- **Original code:**
  ```c
  for (int pos = threadIdx.x; pos < BIN_COUNT; pos += blockDim.x){
    atomicAdd(d_Result + pos, s_Hist[pos]);
  }
  ```

- **Modified with `__any()`:**
  ```c
  for (int pos = threadIdx.x; pos < BIN_COUNT; pos += blockDim.x){
    if (__any (s_Hist[pos] != 0) )
      atomicAdd(d_Result + pos, s_Hist[pos]);
  }
  ```

- **Modified with `__all()`:**
  ```c
  for (int pos = threadIdx.x; pos < BIN_COUNT; pos += blockDim.x){
    if (!__all (s_Hist[pos] == 0) )
      atomicAdd(d_Result + pos, s_Hist[pos]);
  }
  ```
Sparse Matrix Multiplication

- **Sparse Matrix** $N \times N$:
  - number of non-zero entries $m$ is only a small fraction of the total

- Representation goal:
  - store only non-zero entries

- Typically:
  - $m = O(N)$

---

Compressed Sparse Row (CSR) Representation

- $A$:
  - Sample matrix

- $Av[]$:
  - Array values in row-major order

- $Aj[]$:
  - Column for corresponding $Av[]$ entry

- $Ap[]$:
  - row $i$ extends from indexes $Ap[i]$ to $Ap[i+1]-1$ in $Av[]$ and $Aj[]$

---

$GPU$ Programming Shiraz University, Winter 88/Spring 89, Reza Azimi
Single Row Multiply: \( y = Ax \)

\[
\begin{array}{cccc}
\text{Av} & 3 & 1 & 2 \\
\text{Aj} & 0 & 2 & 1 \\
\text{Ap} & 0 & 2 & 2 \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{x} & 10 & 20 & 30 & 40 \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{Ax} & 60 & 0 & 210 & 50 \\
\end{array}
\]

\[
2 \times 20 + 4 \times 30 + 1 \times 40 = 210
\]

---

**Single Row Multiply**

```c
float multiply_row (uint rowsize,
        uint *Aj, // column indices for row
        float *Av, // nonzero entries for row
        float *xl // the RHS vector
    {
        float sum = 0;
        for (uint column=0; column<rowsize; column++)
            sum = Av[column] * x[ Aj[column] ] ;
        return sum;
    }
```

---

13
Sparse Matrix Multiplication: Serial Code

```c
void csrmul_serial (uint *Ap, uint *Aj, float *Av, uint num_rows, float *x, float *y)
{
    for (uint row=0; row<num_rows; row++)
    {
        uint row_begin = Ap[row];
        uint row_end = Ap[row + 1];

        y[row] = multiply_row (
            row_end - row_begin,
            Aj+row_begin,
            Av+row_begin,
            x);
    }
}
```

CUDA Strategy

- Assume that there are many rows
- One thread per row
CUDA Kernel

```c
__global__
void csrMul_kernel (uint *Ap,
                   uint *Aj,
                   float *Av,
                   uint num_rows,
                   float *x,
                   float *y)
{
    uint row = blockIdx.x * blockDim.x + threadIdx.x;
    uint row_begin = Ap[row];
    uint row_end = Ap[row+1];

    y[row] = multiply_row (row_end - row_begin,
                           Aj + row_begin,
                           Av + row_begin,
                           x);
}
```

Other Alternatives

- Assign each row to a warp, instead of a thread

- Each thread in the warp calculates a portion of the row

- Use parallel reduction for accumulating the results
  - Easy since it’s done within a warp
  - Complete unrolling can be done
CSR Vector Multiplication

```c
__global__ void spmv_csr_vector_kernel (const int num_rows , const int * ptr , const int * indices , const float * data , const float * x, float * y)
{
    __shared__ float vals [];
    int thread_id = blockDim.x * blockIdx.x + threadIdx.x; // global thread index
    int warp_id = thread_id / 32; // global warp index
    int lane = thread_id & (32 - 1); // thread index within the warp
    int row = warp_id;

    if (row < num_rows) {
        int row_start = ptr[row];
        int row_end = ptr[row + 1];
        vals[threadIdx.x] = 0;
        for (int jj = row_start + lane; jj < row_end; jj += 32)
            vals[threadIdx.x] += data[jj] * x[indices[jj]];

        // parallel reduction in shared memory
        if (lane < 16) vals[threadIdx.x] += vals[threadIdx.x + 16];
        if (lane < 8) vals[threadIdx.x] += vals[threadIdx.x + 8];
        if (lane < 4) vals[threadIdx.x] += vals[threadIdx.x + 4];
        if (lane < 2) vals[threadIdx.x] += vals[threadIdx.x + 2];
        if (lane < 1) vals[threadIdx.x] += vals[threadIdx.x + 1];

        // first thread writes the result
        if (lane == 0)
            y[row] += vals[threadIdx.x];
    }
}
```

source: Efficient Sparse Matrix-Vector Multiplication on CUDA, by Nathan Bell and Michael Garland