Summed Area Ripmaps

Gernot Ziegler, NVIDIA
Problem Task

- Compute large number of **area sums** over input data (one-, two- or n-dimensional, integer or float).
- Example of area sum request from 1D input:

```
5 0 2 3 1 0 0 3 0 5 0 7 1 2 0 0
```

Start=3  Requested sum range  End=15
Classic Solution: Summed Area Table

- Precompute a Prefix Scan of input (sum of all predecessors).
- Subtract values at End and Start, outcome is area sum.

**ISSUE:** Complexity of Prefix Scan - Minor

**ISSUE:** Large float array: Precision of \( \text{diff(large numbers)} \)!

Large integer array: Wrap-around!
Summed Area Ripmap: Partial Sums

- An alternative approach to computing area sums from data
- Better Precision, no unexpected datatype overflow
Idea for Summation

- Use partial sums from ripmap!
1D SUMMED AREA RIPMAP
Ripmap Buildup

- Summed area ripmaps hold **partial sums** of elements
- A ripmap contains all power-of-two reduction sums of input
- Reduction operator: Sum of 2 input elements

Reduction repeated until a single element remains (non-power-of-2 input: padding with zeroes)
### Summed Area Ripmap: Partial Sums

#### 1D Ripmap (memory view)

|   | 5 | 0 | 2 | 3 | 1 | 0 | 0 | 3 | 0 | 5 | 0 | 7 | 1 | 2 | 0 | 0 | 5 | 5 | 1 | 3 | 5 | 7 | 3 | 0 | 10 | 4 | 12 | 3 | 14 | 15 | 29 |

#### 1D ripmap (coverage view)

<table>
<thead>
<tr>
<th></th>
<th>29</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>
Implementation: Observations

- Summation request might not be aligned, e.g. Start=3, End=15

```
<table>
<thead>
<tr>
<th></th>
<th>14</th>
<th></th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Starting row 3, ending row 15.
Implementation: Observations

- Partial sums only available for aligned address ranges:
  4-wide sums at 0, 4, 8, ... (0000b, 0100b, 1000b)
  16-wide sums at 0, 16, ... (0000b, 1000b)
Implementation: Address optimizations

- Note: Each ripmap level has its own “address space”
Implementation: Approach

- If we fetch from Start forwards and from End backwards. Alignment for increasingly (!) wider partial sums happens.

<table>
<thead>
<tr>
<th></th>
<th>14</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>7</td>
</tr>
</tbody>
</table>

5 0 2 3 1 0 0 3 0 5 0 7 1 2 0 0

Start/End-Fetches meet! Stop.
Implementation: Approach

```c
foreach (Start, End) do (in parallel)
    while (Start < End) // as long as Start hasn't surpassed End
    {
        if (Start & 1) // Start's lowest bit set?
            result += ripmap_fetch(level, Start); // Yes: fetch partial sum from ripmap (No: Wait until next level)
        if (Start < End && (End & 1)) // End's highest bit set and Start < End valid?
            result += ripmap_fetch(level, End - 1); // Yes: fetch partial sum from ripmap (No: Wait until next level)
        Start = (Start + 1) >> 1; // move Start forward and prepare for next ripmap level
        End = End >> 1; // prepare End for next ripmap level
        level = level + 1; // next ripmap level
    }
```
Implementation: Address optimizations

- Now look at bit pattern of start and end:
  Start=3=0011b, End=15=1111b.

- Insight: There will always be a level to fetch from (lowest level), until Start and End are equal.

- Every time we fetch an element (Start: right, End: left), Start increases and End decreases.

- As soon as Start and End have reached higher level alignment, no lower level fetches are necessary anymore.
Implementation: Address optimizations

- Due to the binary reduction pattern, we only need to fetch once from every level before we can move on to higher ripmap level. (if we would fetch twice from same level, we could have fetched from the higher ripmap level instead)

- Address conversion between ripmap levels happens through right-shift of Start and End (e.g. $\text{End} = \text{End} \gg 1$).

- The lowest bits of Start and End determine if a fetch happens at a certain ripmap level (if $(\text{Start} \& 1 == 1)$ ...).

- Stop when End and Start are equal (= all fetches done)
Implementation: Address optimizations

- Level 0: Start=3=0011b End=15=1111b

The lowest bits of Start and End determine if a fetch happens at a certain ripmap level.
Implementation: Address optimizations

- Level 1: Start = (3+1)>>1 = 2 = 0010b  End = (15-1)>>1 = 7 = 0111b

Address conversion between ripmap levels happens through right-shift of Start and End (e.g. End = End >> 1).
Implementation: Address optimizations

- Level 2: Start=2>>1=1=0001b  End=(7-1)>>1=3=0011b

Stop when End and Start are equal (= all fetches done)
Ripmap Buildup: 2D and N-D

- More dimensions are handled one-by-one
- 2D input: E.g. x-axis (horizontal reduction) first, then reduction of *complete* horizontal ripmap along y-axis.

![Diagram of ripmap x-axis reduction](attachment:image.png)
Ripmap Buildup: 2D and N-D

- 2D example: Vertical reduction:
  - Input is horizontal ripmap!

- General:
  Following reduction stages always take complete ripmap output from previous stage
# 2D Ripmap

## 2D Input

```
0 0 2 3 0 0 0 0 0 5 0 0 5 0 5
0 0 1 0 0 1 0 0 0 1 1 0 1 1 2
0 0 1 0 0 1 0 0 0 1 1 0 1 1 2
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 1 1 0 1 1 2
0 1 5 0 0 0 1 0 1 5 0 1 6 1 7
0 0 1 1 1 1 0 0 0 2 2 0 2 2 4
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 3 3 0 1 0 0 0 6 1 0 6 1 7
0 0 1 0 0 1 7 0 0 1 1 7 1 8 9
1 1 5 0 0 0 1 1 2 5 0 2 7 2 9
0 0 1 1 1 1 0 0 0 2 2 0 2 2 4
0 0 4 3 0 2 7 0 0 7 2 7 7 9 16
1 1 6 1 1 1 1 1 2 7 2 2 9 4 13
1 1 10 4 1 3 8 1 2 14 4 9 16 13 29
```
2D Ripmap: Meaning of Partial sums

2D Input

2D Ripmap

Precomputed sum coverages
2D Ripmap: How to compute summation?

2D Input

2D Ripmap

Precomputed sum coverages
2D Ripmap: How to compute summation?

Summation Requests are 2D: StartX, EndX, StartY, EndY

Two-Stage:

I) Compute 2D ripmap Y positions via StartY and EndY (similar to 1D case, but no actually ripmap fetch)

II) For every given 2D ripmap Y position, use StartX and EndX to compute all ripmap X positions to conduct the actual 2D ripmap fetches.

(Similar with 3D input and ripmaps in x-y-z stages, etc.)
2D Ripmap: How to compute summation?
2D Ripmap: How to compute summation?

Box filter request
4x4 @ [1,4]

Actual Ripmap lookups (two 4x1, one 4x2 request)
2D Summed Area Table (2D SAT)

- Also known as “integral images”
- Originally designed to replace mipmaps [Crow84]
- Used in spatially varying filters (e.g. [Hensley05])
- Buildup: Horizontal and vertical Prefix-Sum Scan operations
- Sum Area Requests: Add/subtract values from area corners
- Faster? Yes, but consider precision of input. Area sum is computed from data that has large numbers to obtain difference!

[Crow84] F.C.Crow “Summed-area tables for texture mapping”, Proc. of SIGGRAPH 84
Applications of Summed Area Ripmaps

Image Processing:
- Spatially varying filters of high contrast input (e.g. Face Detection on HDR video)
- Anisotropic data filtering (Algorithm is very hardware and cache friendly)

Non-imaging:
- Numerical computation of 2D and ND probabilities (area under the probability distribution) from cumulative distribution functions
RESULTS
GENERAL PERFORMANCE

- CUDA C 5.0, GTX 680
- Input: 1920x1080 float1 values
- Ripmap buildup: 1.37 ms

- Realtime!
  As expected, execution time nearly independent of window size:
  min 1 ripmap fetch, max. 2 log width * 2 log height ripmap fetches.

- (*): Little spatial variation in window sizes leads to better caching.

<table>
<thead>
<tr>
<th>Area Sum Requests’ Window Size</th>
<th>Kernel Timing</th>
</tr>
</thead>
<tbody>
<tr>
<td>90x90 to 110x110</td>
<td>10.12 ms</td>
</tr>
<tr>
<td>990x990 to 1010x1010</td>
<td>7.23 ms (*)</td>
</tr>
<tr>
<td>900x900 to 1100x1100</td>
<td>10.61 ms</td>
</tr>
</tbody>
</table>
2D SAT COMPARISON: SPEED

- CUDA C 4.2, Quadro 1000M
- SAT provided by CUDPP 2.0
- Input: 1024x1024 random float1 values [0.0, 1.0]
- 1000 area sum requests of 30x30 to 50x50 size
- Slower than SAT for larger # of requests

<table>
<thead>
<tr>
<th>Number of area sum requests</th>
<th>2D SAT</th>
<th>Summed Area Ripmap</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>2.15ms</td>
<td>2.22ms</td>
</tr>
<tr>
<td>10 K</td>
<td>2.16 ms</td>
<td>3.52ms</td>
</tr>
<tr>
<td>1 Million</td>
<td>17.94 ms</td>
<td>155ms</td>
</tr>
</tbody>
</table>
2D SAT COMPARISON: PRECISION

- CUDA C 4.2, Quadro 1000M
- SAT provided by CUDPP 2.0
- Input:
  2D array of random float1 values [0.0, 1.0]
- 1000 area sum requests of 30x30 to 50x50 size (CPU Reference: double)
- Up to 10x Better Precision
- No datatype overflow

<table>
<thead>
<tr>
<th>Input array resolution</th>
<th>Summed Area Ripmap</th>
<th>2D SAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>32x32</td>
<td>0.152</td>
<td>0.152</td>
</tr>
<tr>
<td>128x128</td>
<td>0.146</td>
<td>0.203</td>
</tr>
<tr>
<td>512x512</td>
<td>0.714</td>
<td>4.156</td>
</tr>
<tr>
<td>1024x1024</td>
<td>2.231</td>
<td>22.09</td>
</tr>
</tbody>
</table>
**Summary**

- An alternative approach to computing area sums from data
- Better Precision (10x at 1024x1024) in exchange for runtime
- No unexpected datatype overflow (e.g. 2048x2048, float32)

**Ideas**
- Use Ripmaps to precompute partial sums
- Exploit bit pattern of area bounds to fetch partial sums
- Algorithm independent of dimensionality

**Future work**
- Analyze the performance in greater details
- Explore non-rectangular and higher-order filtering deeper
- Investigate usefulness of other global operators (e.g. histograms)
QUESTIONS?
Non-rectangular summation tasks

Implementation: Break into less-dimensional stripes (left) or use rectangular tiles (not shown)

More ripmap fetches than rectangular summation, but still more efficient than only input fetches.
Stripes: only less-dimensional ripmap necessary (right).