CS 677: Parallel Programming for Many-core Processors
Lecture 7

Instructor: Philipppos Mordohai
Webpage: www.cs.stevens.edu/~mordohai
E-mail: Philipppos.Mordohai@stevens.edu
Logistics

• Midterm: March 22 (after spring break)
  – Closed book
  – All notes from weeks 2 to 7, except prefix sum
  – No version-specific details and parameters
  – Device parameters will be provided if necessary
Overview

• Homework 4
• Case Study – Electrostatic Potential Calculation
  – A class project at UIUC also resulting in publications
  – Chapter 12 in K&H
• Input Binning
  – From NVIDIA and University of Houston
• Sparse vector matrix multiplication
• Summed area tables
Homework Assignment 4

• Apply Sobel filter on (grayscale) images

\[
G_x = \begin{bmatrix}
-1 & 0 & 1 \\
-2 & 0 & 2 \\
-1 & 0 & 1
\end{bmatrix}
\quad
G_y = \begin{bmatrix}
-1 & -2 & -1 \\
0 & 0 & 0 \\
1 & 2 & 1
\end{bmatrix}
\]
for (i = 1; i < ImageNRows - 1; i++)
    for (j = 1; j < ImageNCols - 1; j++)
    {
        sum1 = u[i-1][j+1] - u[i-1][j-1]
            + 2 * u[i][j+1] - 2 * u[i][j-1]
            + u[i+1][j+1] - u[i+1][j-1];
        sum2 = u[i-1][j-1] + 2 * u[i-1][j]
            + u[i-1][j+1] - u[i+1][j-1]
            - 2 * u[i+1][j] - u[i+1][j+1];
        magnitude = sum1*sum1 + sum2*sum2;
        if (magnitude > THRESHOLD)
            e[i][j] = 255;
        else
            e[i][j] = 0;
    }
• Compute magnitude of filter response $G_x^2 + G_y^2$ and output:
  – 0 if magnitude below threshold
  – 255 if magnitude above threshold
  – 0 pixel is within 1 pixel of image border
Example Output
Open Questions

• Memory bandwidth
• 1D vs. 2D block structure
  – Fetching of pixels at block boundaries
• I prefer solutions without padding, but you can pad for a 10% penalty

• Solutions using global memory only will receive little credit
The PPM Image Format

- PPM is a very simple format
- Each image file consists of a header followed by all the pixel data
- Header

  P6
  # comment 1
  # comment 2

  #comment n
  rows columns maxvalue pixels

  P3 means ASCII file
  P6 means binary (most practical)

  See filereading code in homework zip file

Use Gimp or IrfanView to manipulate images and convert between formats
fp = fopen(filename, "rb");
...
int num = fread(chars, sizeof(char), 1000, fp);
if (chars[0] != 'P' || chars[1] != '6')
{
    fprintf(stderr, "ERROR file '%s' does not
    start with "P6" I am expecting a binary
    PPM file\n", filename);
    return NULL;
}

check for “P6” in first line
Reading the Header (cont)

unsigned int width, height, maxvalue;
char *ptr = chars+3; // P 6 newline
if (*ptr == '#') // comment line!
{
    ptr = 1 + strstr(ptr, "\n");
}
num = sscanf(ptr, "%d\n%d\n%d",
    &width, &height, &maxvalue);
fprintf(stderr, "read %d things   width %d  height %d
    maxval %d\n", num, width, height, maxvalue);
*xsize = width;
*ysize = height;
*maxval = maxvalue;

Mary Hall
CS6963 University of Utah
Reading the Data

// allocate buffer to read the rest of the file into
int bufsize = 3 * width * height * sizeof(unsigned char);
if ((*maxval) > 255) bufsize *= 2;
unsigned char *buf = (unsigned char *)malloc( bufsize );

... long numread = fread(buf, sizeof(char), bufsize, fp);
...

int pixels = (*xsize) * (*ysize);
for (int i=0; i<pixels; i++)
    pic[i] = (int) buf[3*i]; // red channel
return pic; // success
Electrostatic potential map is used in building stable structures for molecular dynamics simulation
The contribution of atom[i] to the electrostatic potential at lattice point j is \( \text{atom[i].charge} / r_{ij} \)

The total potential at lattice point j is the sum of contributions from all atoms in the system.
Sequential CPU Code

```c
void cenergy(float *energygrid, dim3 grid, float gridspacing, float z, const float *atoms,
        int numatoms) {
    int i,j,n;
    int atomarrdim = numatoms * 4:
    for (j=0; j<grid.y; j++) {
        float y = gridspacing * (float) j;
        for (i=0; i<grid.x; i++) {
            float x = gridspacing * (float) i;
            float energy = 0.0f;
            for (n=0; n<atomarrdim; n+=4) { // calculate potential contribution of each atom
                float dx = x - atoms[n    ];
                float dy = y - atoms[n+1];
                float dz = z - atoms[n+2];
                energy += atoms[n+3] / sqrtf(dx*dx + dy*dy + dz*dz);
            }
            energygrid[grid.x*grid.y*k + grid.x*j + i] = energy;
        }
    }
}
```
GPU Implementation

• Option 1: each thread calculates the contribution of one atom to all grid points
  – “Scatter”
• Option 2: each thread calculates the accumulated contributions of all atoms to one grid point
  – “Gather”
• Pros/cons?
Loop Transformation

• Need perfectly nested loops
  – as in MRI example

  – Move calculation of y into inner loop

  – Pros/cons?

```c
for (j=0; j<grid.y; j++) {
    float y = gridspacing * (float) j;
    for (i=0; i<grid.x; i++) {
        float x = gridspacing * (float) i;
        float energy = 0.0f;
        for (n=0; n<atomarrdim; n+=4) {
            float dx = x - atoms[n];
            float dy = y - atoms[n+1];
            float dz = z - atoms[n+2];
            energy += atoms[n+3] / sqrtf(dx*dx + dy*dy + dz*dz);
        }
        energygrid[grid.x*grid.y*k + grid.x*j + i] = energy;
    }
}
```
DCS Kernel Design Overview

- Grid of thread blocks
- Lattice padding
- Thread blocks: 64-256 threads
- Threads compute up to 8 potentials, skipping by half-warps
- Host
- Atomic Coordinates Charges
- Constant Memory
- GPUs
- Parallel Data Cache
- Texture
- Global Memory
float curenergy = energygrid[outaddr];
float coorx = gridspacing * xindex;
float coory = gridspacing * yindex;
int atomid;
float energyval = 0.0f;
for (atomid = 0; atomid < numatoms; atomid++) {
    float dx = coorx - atominfo[atomid].x;
    float dy = coory - atominfo[atomid].y;
    energyval += atominfo[atomid].w *
        rqsqrtf(dx*dx + dy*dy + atominfo[atomid].z);
}
energygrid[outaddr] = curenergy + energyval;

qsqrtf(): reciprocal square root
DCS Kernel Version 1

... 
float curenergy = energygrid[outaddr];
float coorx = gridspacing * xindex;
float coory = gridspacing * yindex;
int atomid;
float energyval = 0.0f;
for (atomid=0; atomid<numatoms; atomid++) {
  float dx = coorx - atominfo[atomid].x;
  float dy = coory - atominfo[atomid].y;
  energyval += atominfo[atomid].w *
      rsqrtf(dx*dx + dy*dy + atominfo[atomid].z);
}
energygrid[outaddr] = curenergy + energyval;

Start global memory reads early. Kernel hides some of its own latency.

atominfo[].z is already squared

Only dependency on global memory read is at the end of the kernel...

qsqrtf(): reciprocal square root
Information Reuse
...for (atomid=0; atomid<numatoms; atomid++) {
    float dy = coory - atominfo[atomid].y;
    float dysqpdzsq = (dy * dy) + atominfo[atomid].z;
    float x = atominfo[atomid].x;
    float dx1 = coorx1 - x;
    float dx2 = coorx2 - x;
    float dx3 = coorx3 - x;
    float dx4 = coorx4 - x;
    float charge = atominfo[atomid].w;
    energyvalx1 += charge * rsqrtf(dx1*dx1 + dysqpdzsq);
    energyvalx2 += charge * rsqrtf(dx2*dx2 + dysqpdzsq);
    energyvalx3 += charge * rsqrtf(dx3*dx3 + dysqpdzsq);
    energyvalx4 += charge * rsqrtf(dx4*dx4 + dysqpdzsq);
}
Memory Coalescing

• Two issues:
  – Each thread calculates potentials of four adjacent grid points
  
  – If grid width is not multiple of tile width, boundary management becomes complicated
Memory Layout for Coalescing

Unrolling increases computational tile size

Thread blocks: 64-256 threads

Grid of thread blocks:

0,0  0,1  ...

1,0  1,1  ...

… … …

Threads compute up to 8 potentials, skipping by half-warps

Padding waste

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2010
ECE408, University of Illinois, Urbana-Champaign
...float coory = gridspacing * yindex;
float coorx = gridspacing * xindex;
float gridspacing_coalesce = gridspacing * BLOCKSIZEX;
int atomid;
for (atomid=0; atomid<numatoms; atomid++) {
    float dy = coory - atominfo[atomid].y;
    float dyz2 = (dy * dy) + atominfo[atomid].z;
    float dx1 = coorx - atominfo[atomid].x;
    [...]
    float dx8 = dx7 + gridspacing_coalesce;
    energyvalx1 += atominfo[atomid].w * rsqrtf(dx1*dx1 + dyz2);
} [...]
energyvalx8 += atominfo[atomid].w * rsqrtf(dx8*dx8 + dyz2);
energygrid[outaddr ] += energyvalx1;
[...]
energygrid[outaddr+7*BLOCKSIZEX] += energyvalx7;

Points spaced for memory coalescing
Reuse partial distance components dy^2 + dz^2
Global memory ops occur only at the end of the kernel, decreases register use

ILP vs. TLP
Performance Comparison

Number of thread blocks modulo number of SMs results in significant performance variation for small workloads.

CUDA-Unroll8clx: fastest GPU kernel, 44x faster than CPU, 291 GFLOPS on GeForce 8800GTX

CUDA-Simple: 14.8x faster, 33% of fastest GPU kernel

CPU vs. CPU-GPU Comparison

Accelerating molecular modeling applications with graphics processors.


© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2010
ECE408, University of Illinois, Urbana-Champaign
Input Binning
Objective

• To understand how data scalability problems in gather parallel execution motivate input binning
• To learn basic input binning techniques
• To understand common tradeoffs in input binning
Scatter to Gather Transformation
However

• Input tends to be much less regular than output
  – It may be difficult for each thread to efficiently locate all inputs relevant to its output
  – Or, to efficiently exclude all inputs irrelevant to its output

• In a naïve arrangement, all threads may have to process all inputs to decide if each input is relevant to its output
  – This makes execution time scale poorly with data set size
  – Important problem when processing large data sets
DCS Algorithm for Electrostatic Potentials Revisited

• At each grid point, sum the electrostatic potential from all atoms
  – All threads read all inputs

• Highly data-parallel

• But has quadratic complexity
  – Number of grid points × number of atoms
  – Both proportional to volume
  – Poor data scalability
Algorithm for Electrostatic Potentials With a Cutoff

- Ignore atoms beyond a cutoff distance, $r_c$
  - Typically 8Å-12Å
  - Long-range potential may be computed separately
- Number of atoms within cutoff distance is roughly constant (uniform atom density)
  - 200 to 700 atoms within 8Å-12Å cutoff sphere for typical biomolecular structures
Implementation Challenge

• For each tile of grid points, we need to identify the set of atoms that need to be examined
  – One could naively examine all atoms and only use the ones whose distance is within the given range
  – But this examination still takes time, and brings the time complexity right back to
    • number of atoms × number of grid points
  – Each thread needs to avoid examining the atoms outside the range of its grid point(s)
Binning

- A process that groups data to form a chunk called *bin*
- Helps problem solving due to data coarsening
- Uniform bin arrays, Variable bins, KD Trees, ...
Binning for Cut-Off Potential

• Divide the simulation volume with non-overlapping uniform cubes

• Every atom in the simulation volume falls into a cube based on its spatial location
  – Bins represent location property of atoms

• After binning, each cube has a unique index in the simulation space for easy parallel access
Spatial Sorting Using Binning

- Presort atoms into *bins* by location in space
- Each bin holds several atoms
- Cutoff potential only uses bins within $r_c$
  - Yields a linear complexity cutoff potential algorithm
**Bin Size Considerations**

- Capacity of atom bins needs to be balanced
  - Too large - many dummy atoms in bins
  - Too small - some atoms will not fit into bins
  - Target bin capacity to cover more than 95% or atoms

- CPU places all atoms that do not fit into bins into an overflow bin
  - Use a CPU sequential algorithm to calculate their contributions to the energy grid lattice points.
  - CPU and GPU can do potential calculations in parallel
Bin Design

- Uniform sized/capacity bins allow array implementation
  - And the relative offset list approach
- Bin capacity should be big enough to contain all the atoms that fall into a bin
  - Cut-off will screen away atoms that weren’t processed
  - Performance penalty if too many are screened away
Going from DCS Kernel to Large Bin Cut-off Kernel

- Adaptation of techniques from the direct Coulomb summation kernel for a cutoff kernel
- Atoms are stored in constant memory as with DCS kernel
- CPU loops over potential map regions that are $(24\text{Å})^3$ in volume (cube containing cutoff sphere)
- Large bins of atoms are appended to the constant memory atom buffer until it is full, then GPU kernel is launched
- Host loops over map regions reloading constant memory and launching GPU kernels until completion
Large Bin Design Concept

• Map regions are $(24\text{Å})^3$ in volume

• Regions are sized large enough to provide the GPU enough work in a single kernel launch
  – $(48$ lattice points$)^3$ for lattice with $0.5\text{Å}$ spacing
  – Small bins don’t provide the GPU enough work to utilize all SMs, to amortize constant memory update time, or kernel launch overhead
Large-bin Cutoff Kernel Evaluation

• 6× speedup relative to fast CPU version
• Work-inefficient
  – Coarse spatial hashing into \((24\text{Å})^3\) bins
  – Only 6.5% of the atoms a thread tests are within the cutoff distance
• Better adaptation of the algorithm to the GPU will gain another 2.5×
Improving Work Efficiency

• Thread block examines atom bins up to the cutoff distance
  – Use a sphere of bins
  – All threads in a block scan the same bins and atoms
    • No hardware penalty for multiple simultaneous reads of the same address
    • Simplifies fetching of data
  – The sphere has to be big enough to cover all grid point at corners
  – There will be a small level of divergence
    • Not all grid points processed by a thread block relate to all atoms in a bin the same way
    • (A within cut-off distance of N but outside cut-off of M)
The Neighborhood is a volume

- Calculating and specifying all bin indexes of the sphere can be quite complex
  - Rough approximations reduce efficiency
Neighborhood Offset List (Pre-calculated)

- A list of relative offsets enumerating the bins that are located within the cutoff distance for a given location in the simulation volume.

- Detection of surrounding atoms becomes realistic for output grid points
  - By visiting bins in the neighborhood offset list and iterating over the atoms they contain.
Performance

- $O(MN')$ where $M$ and $N'$ are the number of output grid points and atoms in the neighborhood offset list, respectively
  - In general, $N'$ is small compared to the number of all atoms
- Works well if the distribution of atoms is uniform
Details on Small Bin Design

- For 0.5Å lattice spacing, a $(4\text{Å})^3$ cube of the potential map is computed by each thread block
  - 8×8×8 potential map points
  - 128 threads per block
    (4 points/thread)
  - 34% of examined atoms are within cutoff distance
More Design Considerations for the Cutoff Kernel

• High memory throughput to atom data essential
  – Group threads together for locality
  – Fetch bins of data into shared memory
  – Structure atom data to allow fetching

• After taking care of memory demand, optimize to reduce instruction count
  – Loop and instruction-level optimization
Tiling Atom Data

- Shared memory used to reduce Global Memory bandwidth consumption
  - Threads in a thread block collectively load one bin at a time into shared memory
  - Once loaded, threads scan atoms in shared memory
  - Reuse: Loaded bins used 128 times

Execution cycle of a thread block

<table>
<thead>
<tr>
<th>Time</th>
<th>Threads individually compute potentials using bin in shared mem</th>
<th>Collectively load next bin</th>
<th>Suspend</th>
<th>Data returned from global memory</th>
<th>Ready</th>
<th>Write bin to shared memory</th>
</tr>
</thead>
</table>

Another thread block runs while this one waits
Handling Overfull Bins

• In typical use, 2.6% of atoms exceed bin capacity
• Spatial sorting puts these into a list of extra atoms
• Extra atoms processed by the CPU
  – Computed with CPU-optimized algorithm
  – Takes about 66% as long as GPU computation
  – Overlapping GPU and CPU computation yields additional speedup
  – CPU performs final integration of grid data
CPU Grid Data Integration

- Effect of overflow atoms are added to the CPU master energy grid array.
- Slice of grid point values calculated by GPU are added into the master energy grid array while removing the padded elements.
GPU Thread Coarsening

• Each thread computes potentials at four potential map points
  – Reuse x and z components of distance calculation
  – Check x and z components against cutoff distance (cylinder test)
• Exit inner loop early upon encountering the first empty slot in a bin
GPU Thread Inner Loop

for (i = 0; i < BIN_DEPTH; i++) {
    aq = AtomBinCache[i].w;
    if (aq == 0) break;

dx = AtomBinCache[i].x - x;
dz = AtomBinCache[i].z - z;
dxdz2 = dx*dx + dz*dz;
    if (dxdz2 > cutoff2) continue;

dy = AtomBinCache[i].y - y;
r2 = dy*dy + dxdz2;
    if (r2 < cutoff2)
        poten0 += aq * rsqrtf(r2);
        // Simplified example

dy = dy - 2 * grid_spacing;
/* Repeat three more times */
}
Cutoff Summation Runtime

GPU cutoff with CPU overlap: 12x-21x faster than CPU core

50k–1M atom structure size

©Wen-mei W. Hwu and David Kirk/NVIDIA 2010
Summary

• Large bins allow re-use of all-input kernels with little code change
  – But work efficiency can be very low
• Use of small-sized bins require more sophisticated kernel code to traverse list of small bins
  – Much higher work efficiency
  – Small bins also serve as tiles for locality
• CPU processes overflow atoms from fixed capacity bins
Sparse Matrix-Vector Multiplication

slides by Jared Hoberock and David Tarjan
(Stanford CS 193G)
Overview

• GPUs deliver high Sparse Matrix Vector (SpMV) performance

• No one-size-fits-all approach
  – Match method to matrix structure

• Exploit structure when possible
  – Fast methods for regular portion
  – Robust methods for irregular portion
Characteristics of SpMV

• Memory bound
  – FLOP : MemOp ratio is very low

• Generally irregular & unstructured
  – Unlike dense matrix operations
Finite-Element Methods

- Discretized on structured or unstructured meshes
  - Determines matrix sparsity structure
Objectives

• Expose sufficient parallelism
  – Develop 1000s of independent threads

• Minimize execution path divergence
  – SIMD utilization

• Minimize memory access divergence
  – Memory coalescing
Sparse Matrix Formats

(DIA) Diagonal
(ELL) ELLPACK
(CSR) Compressed Row
(HYB) Hybrid
(COO) Coordinate

Structured → Unstructured
Compressed Sparse Row (CSR)

- Rows laid out in sequence
- Inconvenient for fine-grained parallelism
CSR (scalar) kernel

• One thread per row
  – Poor memory coalescing
  – Unaligned memory access
CSR (vector) kernel

- One SIMD vector or *warp* per row
  - Partial memory coalescing
  - Unaligned memory access
ELLPACK (ELL)

• Storage for $K$ nonzeros per row
  – Pad rows with fewer than $K$ nonzeros
  – Inefficient when row length varies
Hybrid Format

- ELL handles *typical* entries
- COO handles *exceptional* entries
  - Implemented with segmented reduction
Exposing Parallelism

- DIA, ELL & CSR (scalar)
  - One thread per row
- CSR (vector)
  - One warp per row
- COO
  - One thread per nonzero
Exposing Parallelism

- COO
- CSR (scalar)
- CSR (vector)
- ELL

Matrix Rows

GFLOP/s

1 4 16 64 256 1K 4K 16K 64K 256K 1M 4M
Execution Divergence

• Variable row lengths can be problematic
  – Idle threads in CSR (scalar)
  – Idle processors in CSR (vector)

• Robust strategies exist
  – COO is insensitive to row length
Memory Access Divergence

- Uncoalesced memory access is costly
  - Sometimes mitigated by cache

- Misaligned access is suboptimal
  - Align matrix format to coalescing boundary

- Access to matrix representation
  - DIA, ELL and COO are fully coalesced
  - CSR (vector) is partially coalesced
  - CSR (scalar) is seldom coalesced
## Performance Comparison

<table>
<thead>
<tr>
<th>System</th>
<th>Cores</th>
<th>Clock (GHz)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTX 285</td>
<td>240</td>
<td>1.5</td>
<td>NVIDIA GeForce GTX 285</td>
</tr>
<tr>
<td>Cell</td>
<td>8 (SPEs)</td>
<td>3.2</td>
<td>IBM QS20 Blade (half)</td>
</tr>
<tr>
<td>Core i7</td>
<td>4</td>
<td>3.0</td>
<td>Intel Core i7 (Nehalem)</td>
</tr>
</tbody>
</table>

### Sources:

- Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors
  N. Bell and M. Garland, Proc. Supercomputing '09, November 2009

- Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms
Performance Comparison

![Graph comparing performance of GTX 285, Cell, and Core i7 processors.](image-url)
__global__ void ell_spmv(const int num_rows, const int num_cols,
const int num_cols_per_row, const int stride,
const double * Aj, const double * Ax,
const double * x, double * y)
{
    const int thread_id = blockDim.x * blockIdx.x + threadIdx.x;
    const int grid_size = gridDim.x * blockDim.x;

    for (int row = thread_id; row < num_rows; row += grid_size) {
        double sum = y[row];

        int offset = row;

        for (int n = 0; n < num_cols_per_row; n++) {
            const int col = Aj[offset];

            if (col != -1)
                sum += Ax[offset] * x[col];

            offset += stride;
        }

        y[row] = sum;
    }
}
```c
#include <cusp/hyb_matrix.h>
#include <cusp/io/matrix_market.h>
#include <cusp/krylov/cg.h>

int main(void)
{
    // create an empty sparse matrix structure (HYB format)
    cusp::hyb_matrix<int, double, cusp::device_memory> A;

    // load a matrix stored in MatrixMarket format
    cusp::io::read_matrix_market_file(A, "5pt_10x10.mtx");

    // allocate storage for solution (x) and right hand side (b)
    cusp::array1d<double, cusp::device_memory> x(A.num_rows, 0);
    cusp::array1d<double, cusp::device_memory> b(A.num_rows, 1);

    // solve linear system with the Conjugate Gradient method
    cusp::krylov::cg(A, x, b);

    return 0;
}
```

A library for **sparse linear algebra** and **graph** computations on CUDA
Summed Area Tables

Patrick Cozzi
University of Pennsylvania
CIS 565 - Spring 2011
Summed Area Table

- **Summed Area Table (SAT):** 2D table where each element stores the sum of all elements in an input image between the lower left corner and the entry location.
Summed Area Table

Example:

<table>
<thead>
<tr>
<th>Input image</th>
<th>SAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 1 0 0</td>
<td>4 9 12 14</td>
</tr>
<tr>
<td>0 1 2 0</td>
<td>2 6 9 11</td>
</tr>
<tr>
<td>1 2 1 0</td>
<td>2 5 6 8</td>
</tr>
<tr>
<td>1 1 0 2</td>
<td>1 2 2 4</td>
</tr>
</tbody>
</table>

\[(1 + 1 + 0) + (1 + 2 + 1) + (0 + 1 + 2) = 9\]
Summed Area Table

• Benefit
  – Used to compute different width filters at every pixel in the image in constant time per pixel
  – Just sample four pixels in SAT:

\[
\sum_{\text{filter}} = \frac{s_{ur} - s_{ul} - s_{lr} + s_{ll}}{w \times h},
\]
Summed Area Table

• Uses
  – Glossy environment reflections and refractions
  – Approximate depth of field

## Summed Area Table

<table>
<thead>
<tr>
<th>Input image</th>
<th>SAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 1 0 0</td>
<td></td>
</tr>
<tr>
<td>0 1 2 0</td>
<td></td>
</tr>
<tr>
<td>1 2 1 0</td>
<td></td>
</tr>
<tr>
<td>1 1 0 2</td>
<td></td>
</tr>
</tbody>
</table>
Summed Area Table

<table>
<thead>
<tr>
<th>Input image</th>
<th>SAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 1 0 0</td>
<td></td>
</tr>
<tr>
<td>0 1 2 0</td>
<td></td>
</tr>
<tr>
<td>1 2 1 0</td>
<td></td>
</tr>
<tr>
<td>1 1 0 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
Summed Area Table

Input image

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

SAT

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summed Area Table

Input image

SAT

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
Summed Area Table

Input image

```
2  1  0  0
0  1  2  0
1  2  1  0
1  1  0  2
```

SAT

```
  1  2  2  4
```
Summed Area Table

**Input image**

```
  2  1  0  0
  0  1  2  0
  1  2  1  0
  1  1  0  2
```

**SAT**

```
[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ]
```

- The SAT is constructed by summing the values of the input image in 4x4 blocks and storing them in the corresponding SAT cells.
- The highlighted value in the SAT indicates the cumulative sum for the highlighted block in the input image.
Summed Area Table

Input image

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

SAT

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>
### Summed Area Table

<table>
<thead>
<tr>
<th>Input image</th>
<th>SAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 1 0 0</td>
<td></td>
</tr>
<tr>
<td>0 1 2 0</td>
<td>4 9</td>
</tr>
<tr>
<td>1 2 1 0</td>
<td>2 6</td>
</tr>
<tr>
<td>1 1 0 2</td>
<td>2 5</td>
</tr>
<tr>
<td></td>
<td>2 6</td>
</tr>
<tr>
<td></td>
<td>1 2</td>
</tr>
<tr>
<td></td>
<td>2 4</td>
</tr>
</tbody>
</table>
Summed Area Table

<table>
<thead>
<tr>
<th>Input image</th>
<th>SAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 1 0 0</td>
<td>4 9 12</td>
</tr>
<tr>
<td>0 1 2 0</td>
<td>2 6 9 11</td>
</tr>
<tr>
<td>1 2 1 0</td>
<td>2 5 6 8</td>
</tr>
<tr>
<td>1 1 0 2</td>
<td>1 2 2 4</td>
</tr>
</tbody>
</table>
## Summed Area Table

<table>
<thead>
<tr>
<th>Input image</th>
<th>SAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 1 0 0</td>
<td>4 9 12 14</td>
</tr>
<tr>
<td>0 1 2 0</td>
<td>2 6 9 11</td>
</tr>
<tr>
<td>1 2 1 0</td>
<td>2 5 6 8</td>
</tr>
<tr>
<td>1 1 0 2</td>
<td>1 2 2 4</td>
</tr>
</tbody>
</table>
Summed Area Table

How would you implement this on the GPU?
Summed Area Table

- Recall **Inclusive Scan**:

```
0 1 2 3 4 5 6 7
0 1 3 6 10 15 21 28
```
# Summed Area Table

## Step 1 of 2:

**Input image**

```
  2  1  0  0
  0  1  2  0
  1  2  1  0
  1  1  0  2
```

**Partial SAT**

```
  2  3  3  3
  0  1  3  3
  1  3  4  4
  1  2  2  4
```

One inclusive scan for each row
## Summed Area Table

### Step 2 of 2:

<table>
<thead>
<tr>
<th>Partial SAT</th>
<th>Final SAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 3 3 3</td>
<td>4 9 12 14</td>
</tr>
<tr>
<td>0 1 3 3</td>
<td>2 6 9 11</td>
</tr>
<tr>
<td>1 3 4 4</td>
<td>2 5 6 8</td>
</tr>
<tr>
<td>1 2 2 4</td>
<td>1 2 2 4</td>
</tr>
</tbody>
</table>

One inclusive scan for each Column, bottom to top