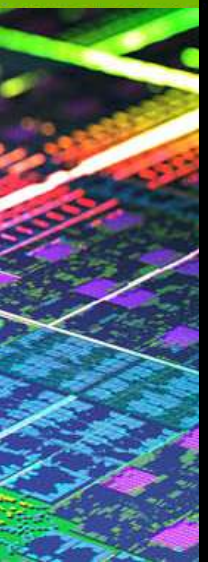

GPU TECHNOLOGY CONFERENCE
Connected Components
Revisited on Kepler (Session 3193)

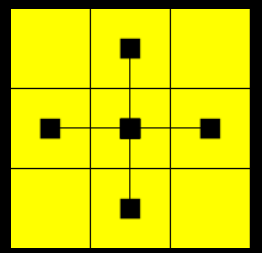


# Agenda

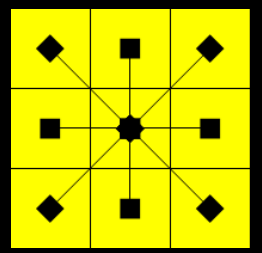
- Introduction / Problem Task
  - Input and Expected Output
- Algorithm
  - Cell Connectivity
  - Label Setup and Label Propagation
  - Acceleration concepts (Links/Max-Gather, LabelRoot)
- Results (Shared memory vs. SHFL, GTC2010 Comparison)
- Conclusion / Future Work

# Introduction / Problem Task

- Input
  - 2D array / 3D array of “data cells” (typical image/ volume data)
  - Connectivity Criterion (possible connections between cells )

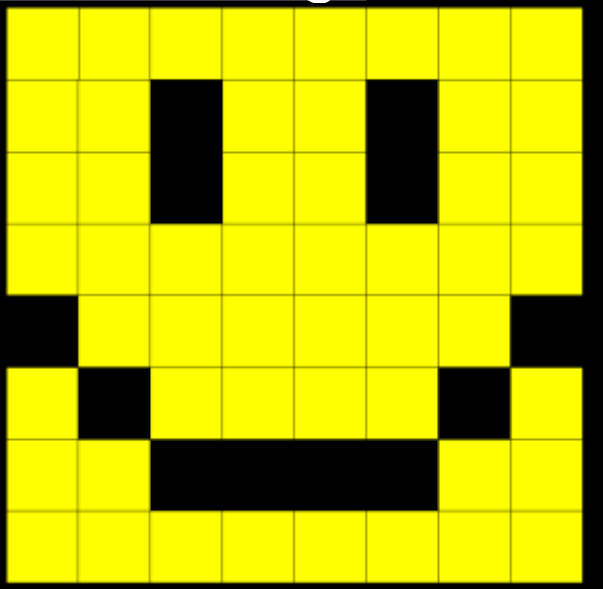


4-connectivity



8-connectivity

- Example Input
  - 2D RGB image

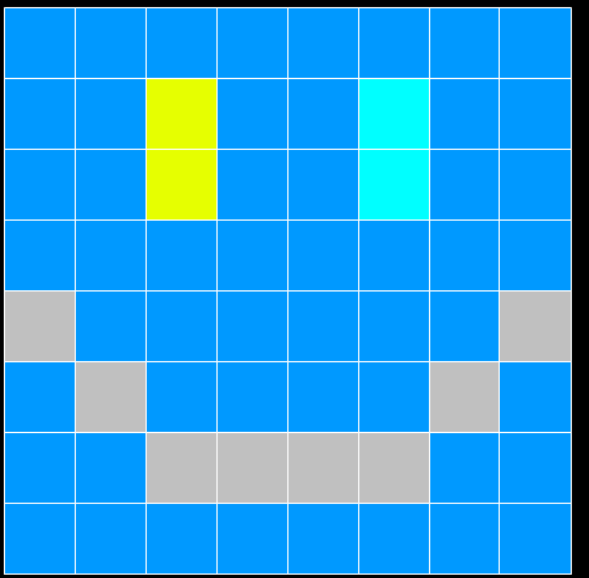


  - Connectivity Criterion: Equal colors, 8-connectivity

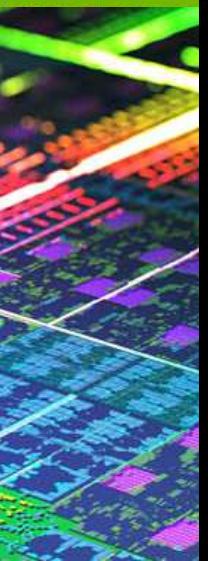
# Introduction / Problem Task

- Output
  - Uniquely labelled regions:
    2D Array / 3D Array
    with all connected regions
    having the same "label" (usually a
    32bit integer value)
- Applications
  - Segmentation (e.g. object separation via depth maps)
  - “Flood Fill”, Magic Wand tool
  - Volume analysis (e.g. MRI results)

- Example
  - 2D array of labels

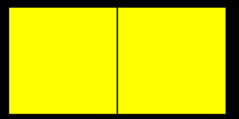
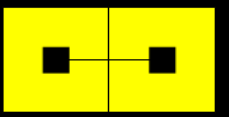


LEGEND
Labels: White outline



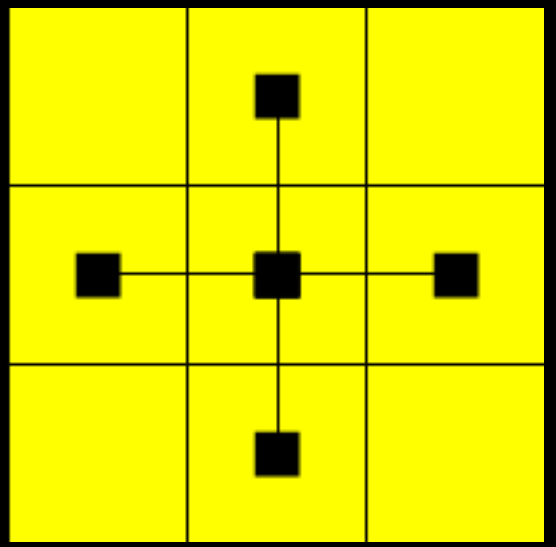
# Cell Connectivity

# Connectivity Criterion

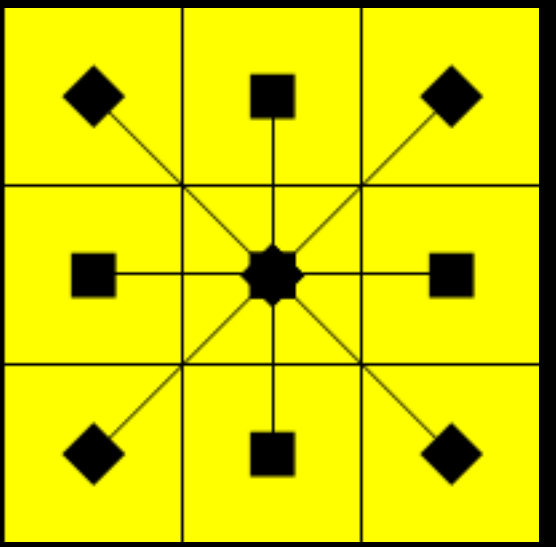
- When are neighboring cells "connected", become a region?
- Example criterion: Equal RGB values
  - Linked (= 1):  , symbolized as: 
  - Not linked (= 0): 
- More useful criterions for noisy input:
  Color gradient thresholding
  e.g. Sum(abs(p0.rgb – p1.rgb)) < 0.1
- Others: Motion vectors, depth maps,…

# 2D: 4- and 8-connectivity

- Are diagonal neighbors regarded as "connected" ?



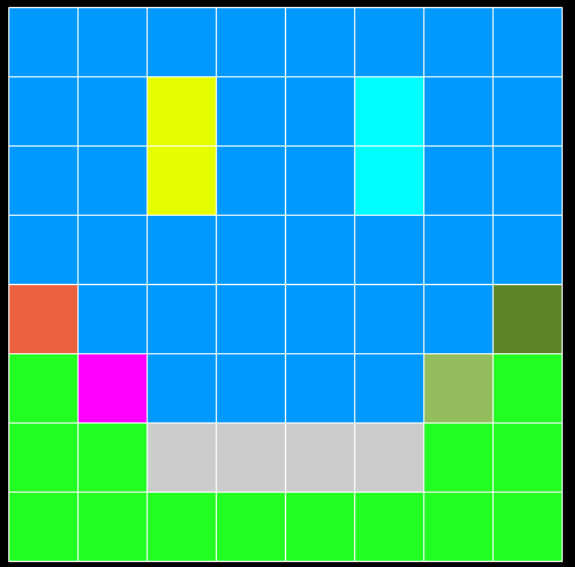
4-connectivity:
Look at vertical and
horizontal neighbors

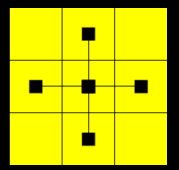


8-connectivity:
Also look at
Diagonal Neighbors

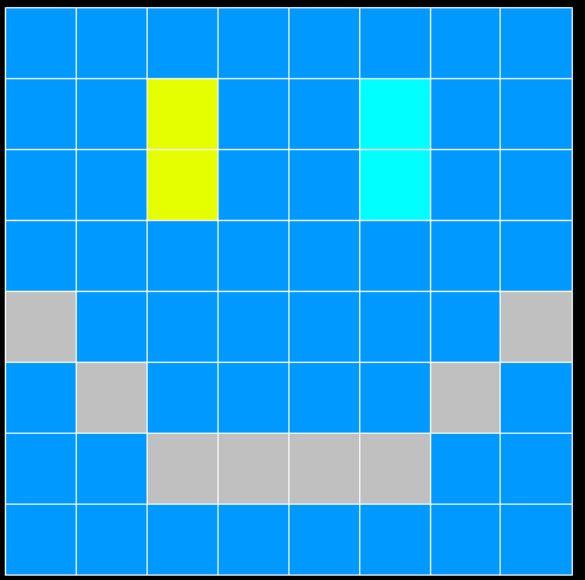
# 4- and 8-connectivity

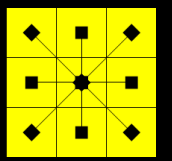
- Affects label propagation!
- Labelling results can differ substantially:





4-connectivity labelling:
Upper and lower part
separate

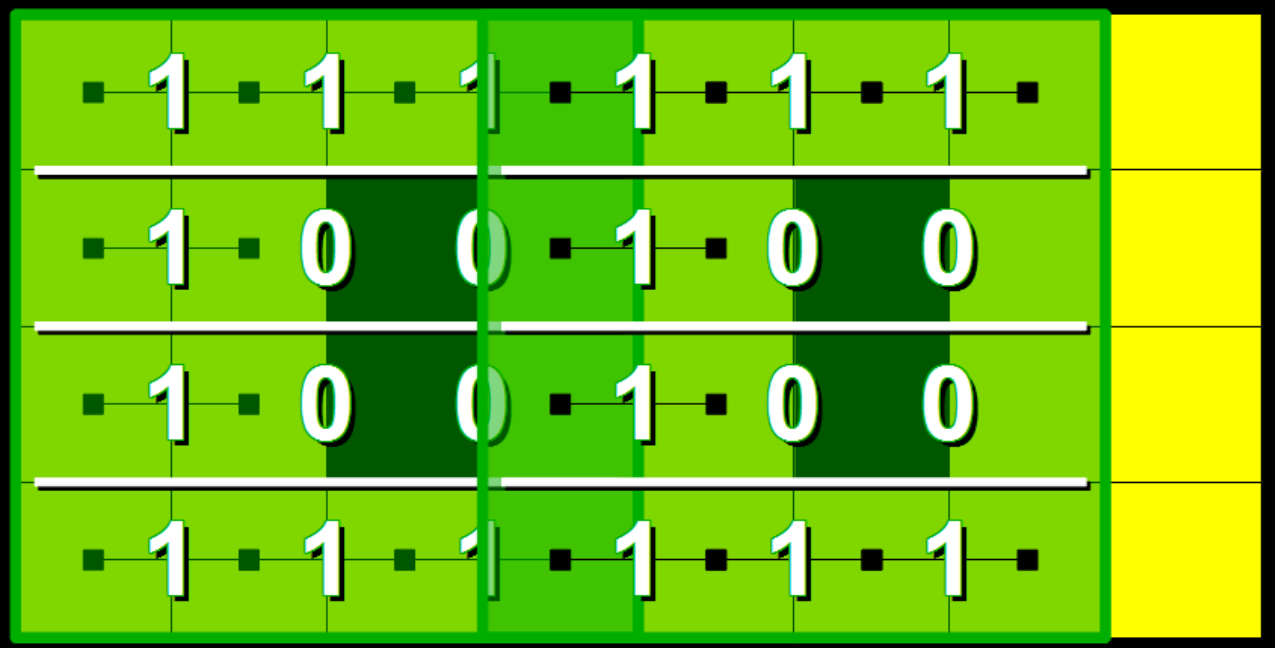


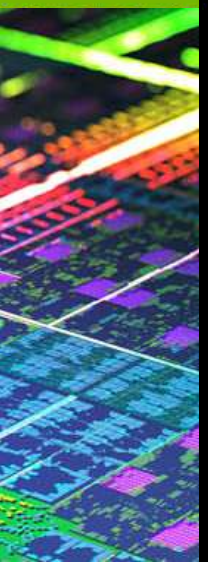


8-connectivity labelling:
Upper and lower part
connected

# Connectivity bits: Implementation

- Threadblocks load overlapping pixel regions into shared memory.
- Threads test connectivity to right and lower pixel neighbour.
- 4x4 threaded example for rightwards connectivity:



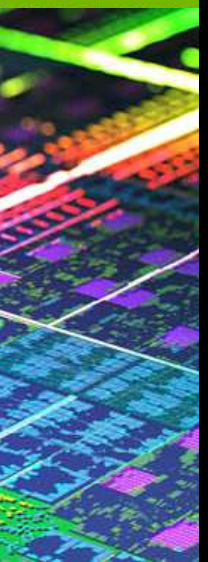


# Connectivity bits: Code snippet

- Bit connectivity to right neighbour
  (cell values pre-loaded in shmem, 32x32 threads):

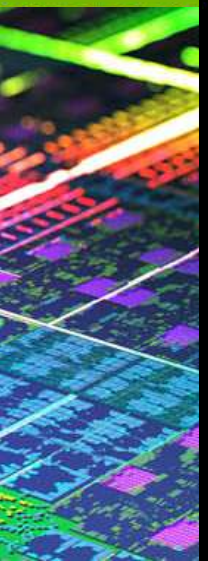
```
con.x =
cellConnected(
sh_pixels[threadIdx.y][threadIdx.x],
sh_pixels[threadIdx.y][threadIdx.x + 1], threshold);
uint32 connright_bits = __ballot(con.x);
if (threadIdx.x == 0) d_connright_bits[pos] = connright_bits;
```

- Label propagation paths (cell connectivity bits) now prepared.



# Connectivity bits: Implementation

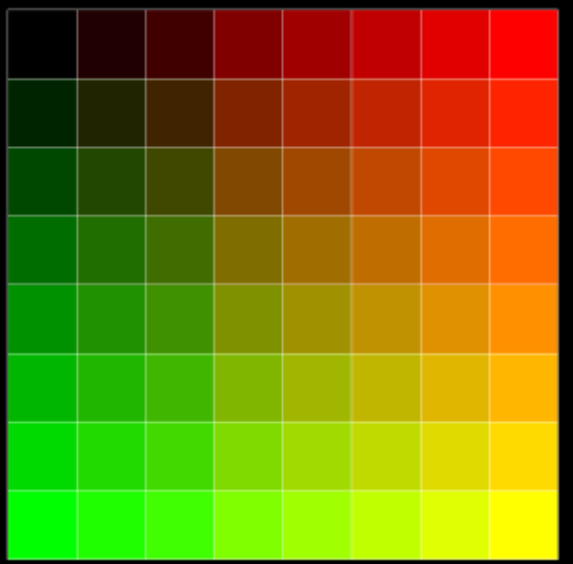
- Each threadblock processes 32x32 pixels, loads into shmem.
- 31x32 threads test connectivity to right pixel neighbour.
- 32x31 threads test connectivity to lower pixel neighbour.
- Output: 32Bit patterns (uint32) describing pixel connectivity rightwards (32 x 31 bits), downwards (32 x 31 bits)
- Warp vote `__ballot()` assists in generating output bit patterns.
- Output describes only region's internal pixel connectivity – but processed image regions overlap.
- Label propagation paths now prepared.



# Algorithm:
# Label Setup and Propagation

# Label setup

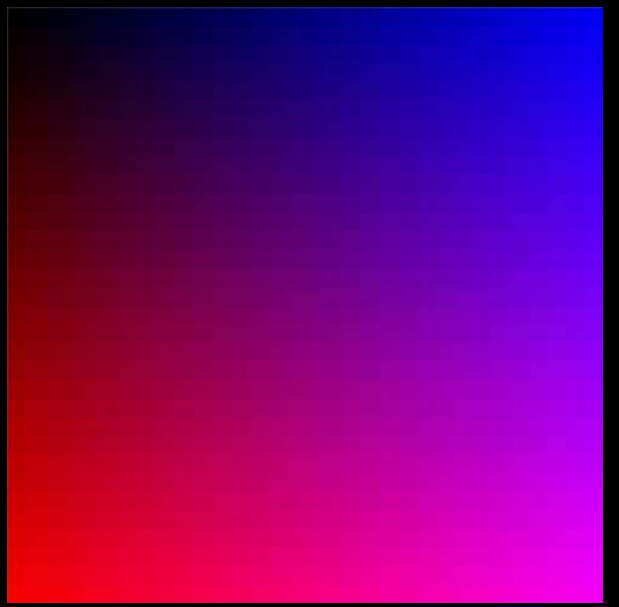
- Initially, each cell receives its own label:
  Label = f(P.x, P.y)
- We want labels comparable in a strict linear order
- e.g. for width < 256:
  Label = P.y * 256 + P.x
- Re-interpreted as colors:
  (X= Red, Y=Green)



# Label propagation: Overview

- Each threadblock loads 32x32 labels and their cell connectivity.
- Labels propagate now amongst threads, using shmem/shuffle.
- Writes out the new label situation if no more label updates.

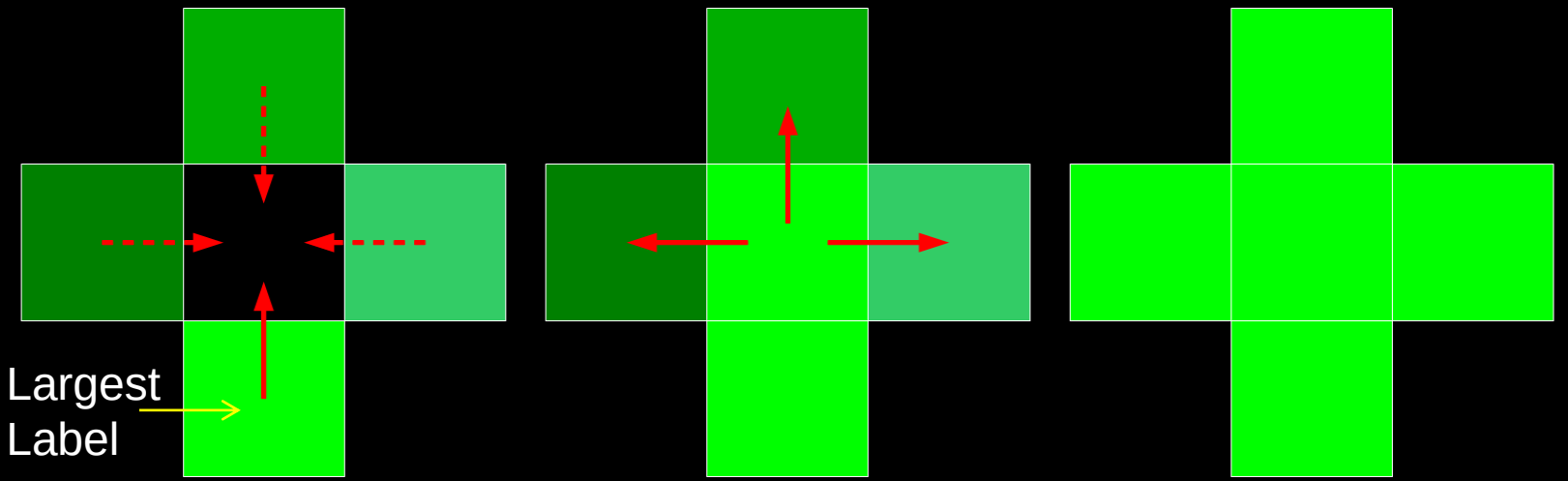
Input



Label Propagation (simple)

# Simple Label Propagation: 1-gather

- Cells gather labels from their closest neighbours: **1-gather**
- Larger value labels propagate to cells with smaller ones
- Gather (no atomics needed)




- *Finish*: When no more updates occur!

# Label update: SHMEM Algorithm

- Get cell connectivity (once):

```
conn_right = d_connright_bits[blockId] & 1 << threadIdx.x;
```

- Do
  - Get label from shmem:

```
labelA = sh_labels[threadIdx.y][threadIdx.x];
```

  - If connected to right: Compare and update label if larger:

```
if (conn_right)
    labelA = max(labelA, sh_data[threadIdx.y][threadIdx.x+1]);
```

  - Write label:

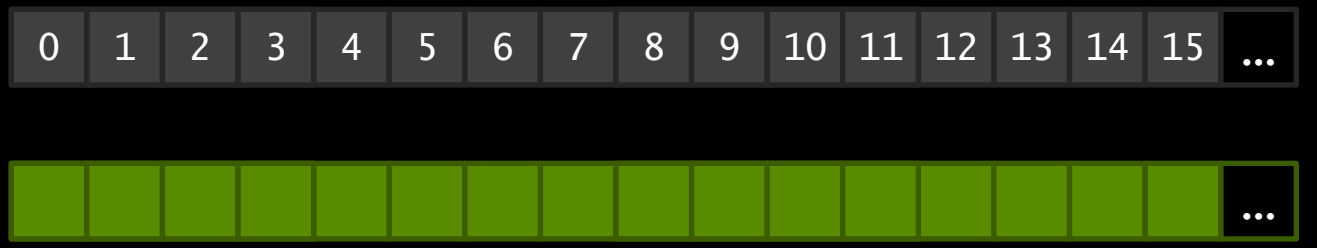
```
sh_labels[threadIdx.y][threadIdx.x] = labelA;
__syncthreads();
```

  - (Same for vertical connections)

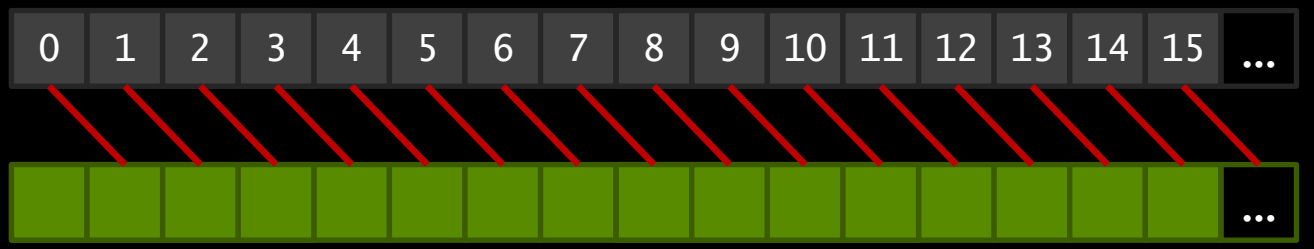
Until (no more label updates)

# Introducing Shuffle (SHFL)

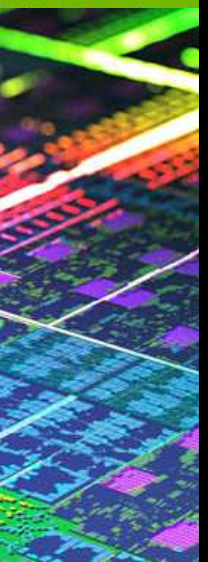
- Introduced in Kepler architecture
- Data communication within a warp (currently: 32 threads), for us: thread rows!
- Initially:




- x_right = __shfl_down(x, 1)




- Thus: Each thread receives its right neighbour's value



# Introducing Shuffle (SHFL)

- Introduced in Kepler architecture
- Data communication within a warp (currently: 32 threads)
- No __syncthreads() required (!)

- In our case: 32x32 threads
- Threads can thus communicate within rows (i.e. same threadIdx.y)

# Label update: SHUFFLE Algorithm

- Get pixel connectivity (once):

```
conn_right = d_connright_bits[blockId] & 1 << threadIdx.x;
```

- Do
  - Get label from shmem:

```
labelA = sh_labels[threadIdx.y][threadIdx.x];
```

  - If connected to right: Compare and update label if larger:

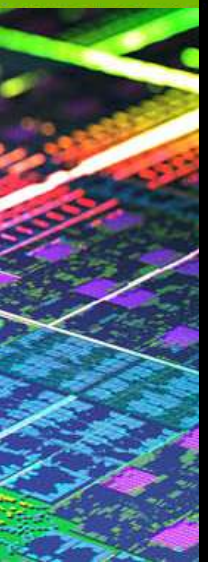
```
right_labelA = __shfl_down((int)labelA, 1);

if (conn_right) labelA = max(labelA, right_labelA);
```

  - Write label in 2D-transposed fashion:

```
sh_labels[threadIdx.x][threadIdx.y] = labelA;
__syncthreads();
```

  - (Same for vertical connections, and read/write transposed again!)

Until (no more label updates)



# Insights on Shuffle

- All threads must participate when using shuffle.
- However, a thread receives its own data when requesting at offset = 0!

  In label comparison, a statement like this:

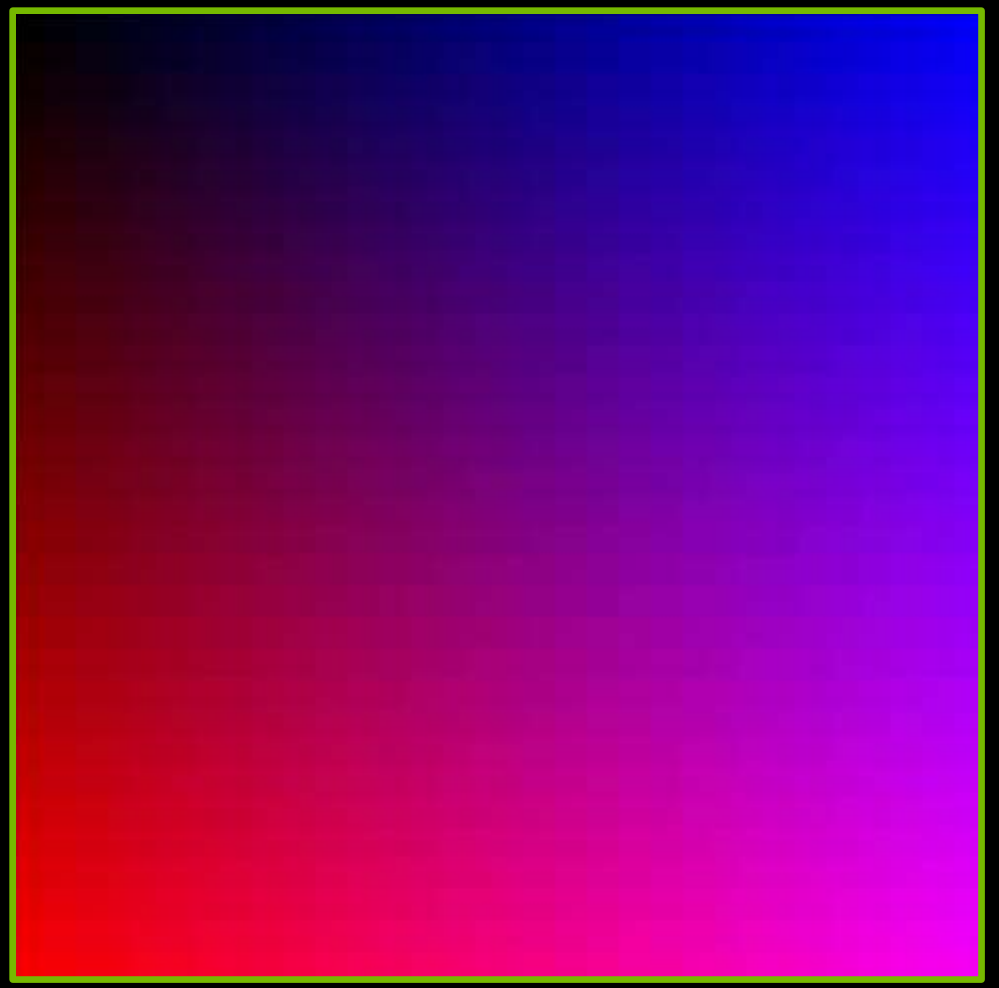
```
left_labelA = __shfl_up((int)labelA, 1);

    if (conn_right) labelA = max(labelA, left_labelA);
```

thus can become:

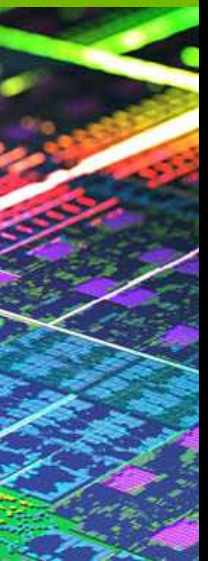
```
labelA = max(labelA, __shfl_up((int)labelA, conn_right ? 1 : 0));
```

i.e. an extra max() for unconnected pixels,
but also avoiding an if().

# Movie: Simple 1-gather (one block)



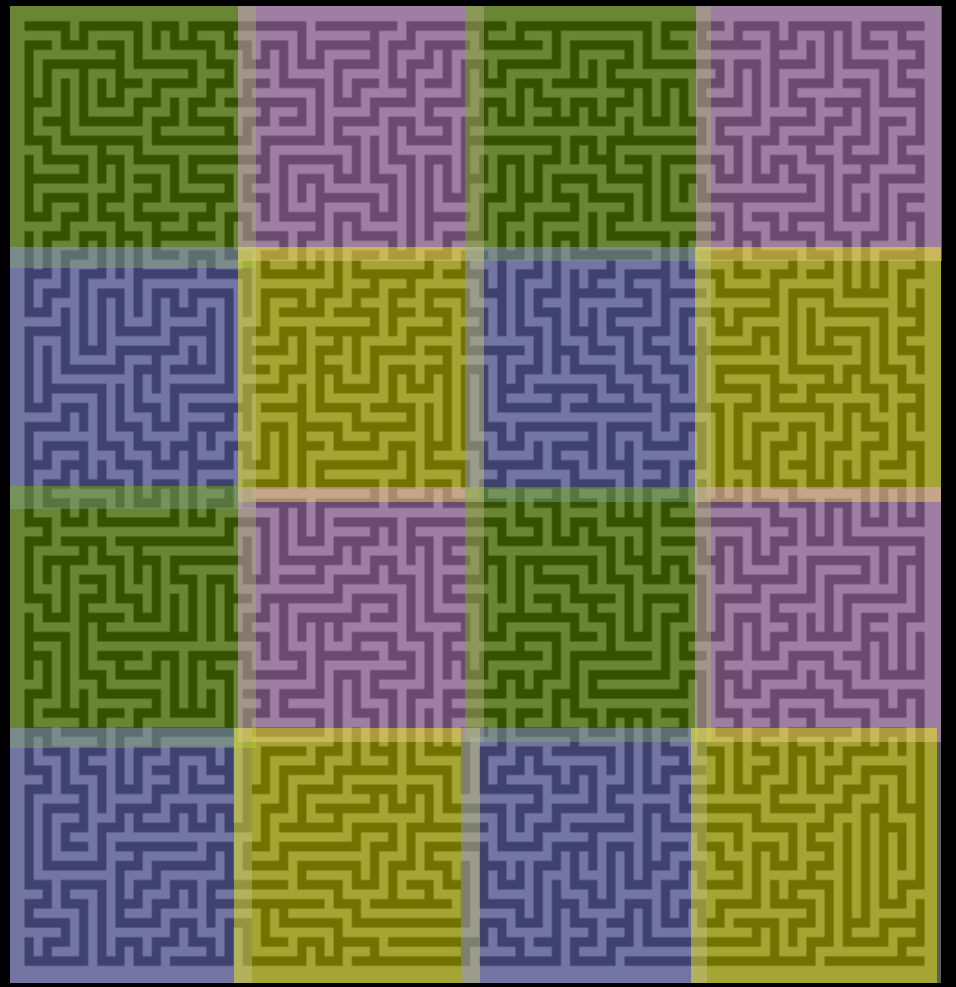
- Input:
- Only 1-gather (single block)
- Works (even though slow label propagation)
- Interesting:
  "Tug-of-war" in upper left, until a much larger label from right (large y component) comes along..



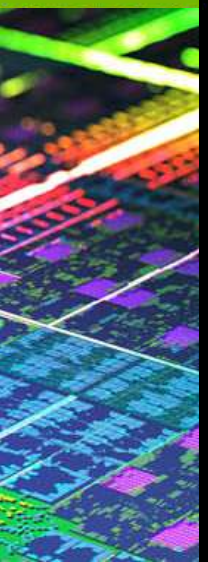
# Label propagation: Whole input

- How do labels pass from one threadblock to another?
- Positioning 32x32 threadblocks at offsets (31,31) from each other
- Creates a “label exchange zone” between the threadblocks (“halo”)
- Several threadblocks process these overlapping label regions!
- But: R/W hazards of label exchange!
  - 4 color approach (Northwest, Southwest, Northeast, Southeast)
  - Four threadblock launches that never overlap in label processing.
  - (Avoids atomic operations)

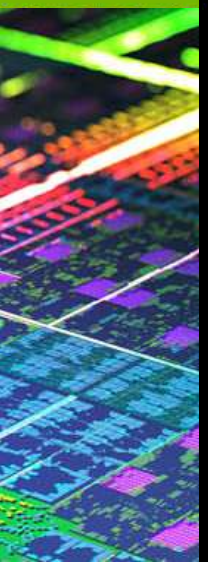
# Label propagation: 4-color grid batching



- Background: Input
- Starting threadblocks of 32x32 threads
- Distanced at offsets of 31 ! Thus overlap, allows label exchange
- Four sequential threadblock launches (four colors) enable “Label exchange zones” without atomic operations



# Algorithm Optimization: max-gather

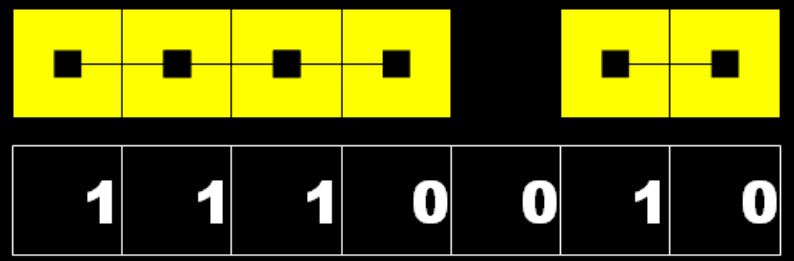


# Max-Gather Offsets: Motivation

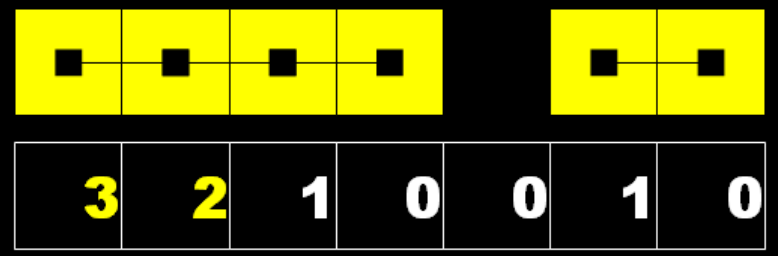
- Problem of 1-gather algorithm: SLOW
  (Each pass, labels propagate only one cell further)
- Can we make labels propagate faster?
- Observation: Connectivity between cells is static!
- Precompute the furthest connected cells along each connectivity direction (e.g. x and y)
- Algorithm: Log2(width|height|depth) steps
- (Similarities with Horn's data-parallel algorithm for prefix sum, GPU Gems 1)

# Max-Gather Offsets: Example

- Input:
  Connectivity to the right:



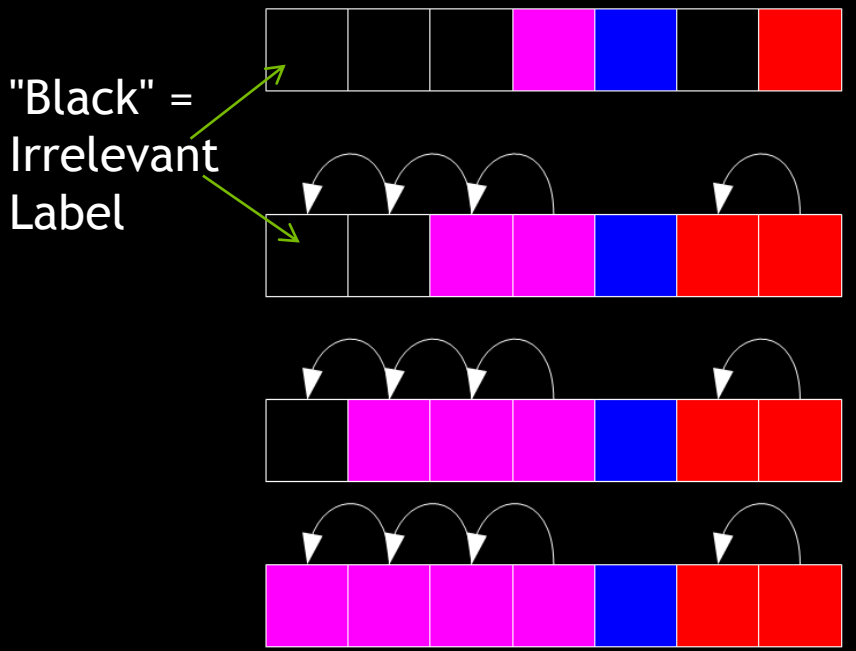

- Desired Output:
  Farthest connected cell
  to the right:



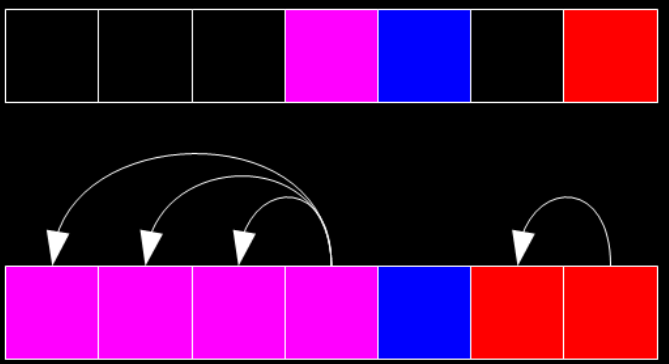
# Label Updates: Faster Gathering

- Precomputation permits **far** label gathering: **max-gather**

- 1-gather



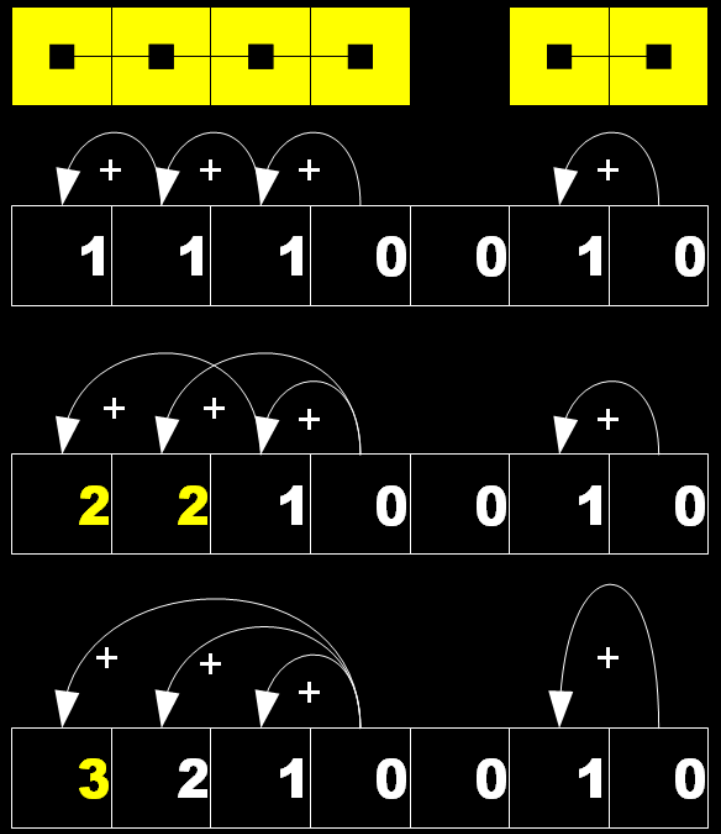

- Max-gather (via Links)



Faster label propagation

# Links: Precomputation Algorithm

- Initialize with local connectivity.
- Repeatedly add link value that link *points to*.
- Example shown: Computing furthest connected pixel to the right.
- (Inverse segmented scan!)



# Links: SHMEM Precomputation Algorithm

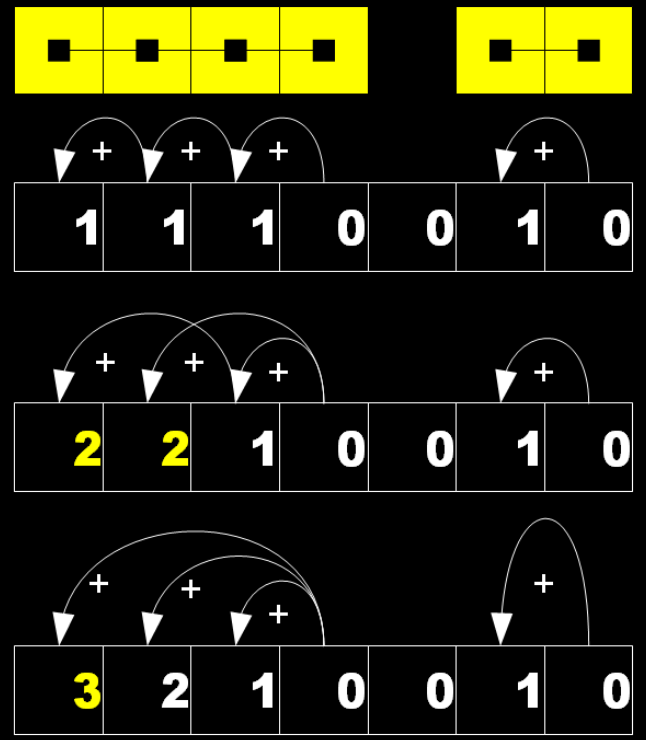
- Initialize with local connectivity:

```
sh_links[threadIdx.y][threadIdx.x] = con.x;
__syncthreads();
```

- Add value from cell that is *already known as connected*:

```
con.x +=
 sh_links[threadIdx.y][threadIdx.x+con.x];
```

- *Do this 5 times (2^5=32 threads)*
- (Same for con.y and reverse-directed links)
- Result: Gather offset to furthest connected cells! max-gather

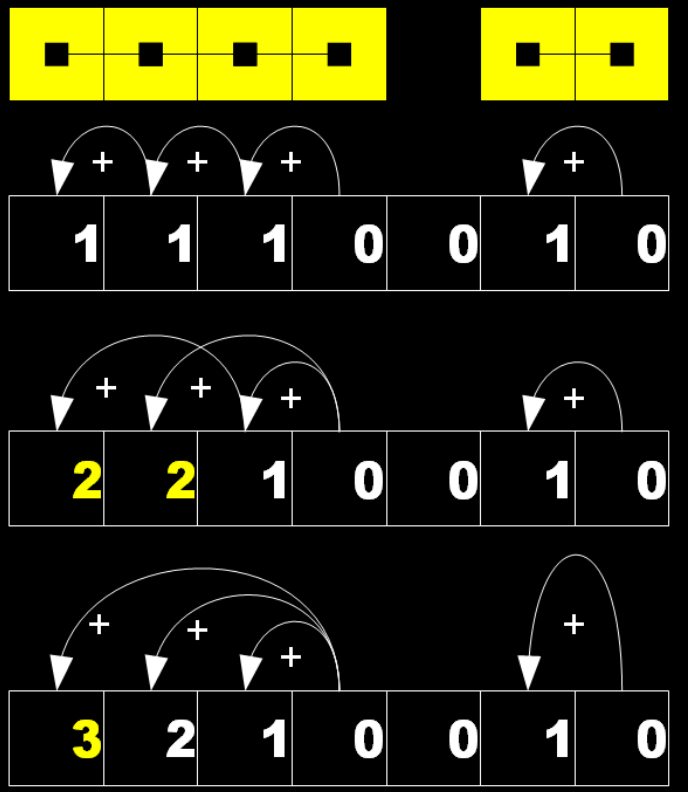


# Links: SHFL Precomputation Algorithm

- Initialize with 1/0 connectivity
- Add value from cell that is *already known as connected*:

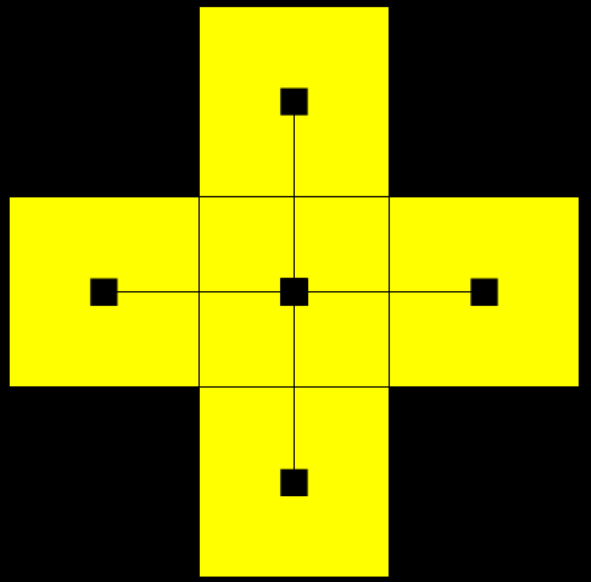
```
conn_right +=
  __shfl_down(conn_right, conn_right);
```

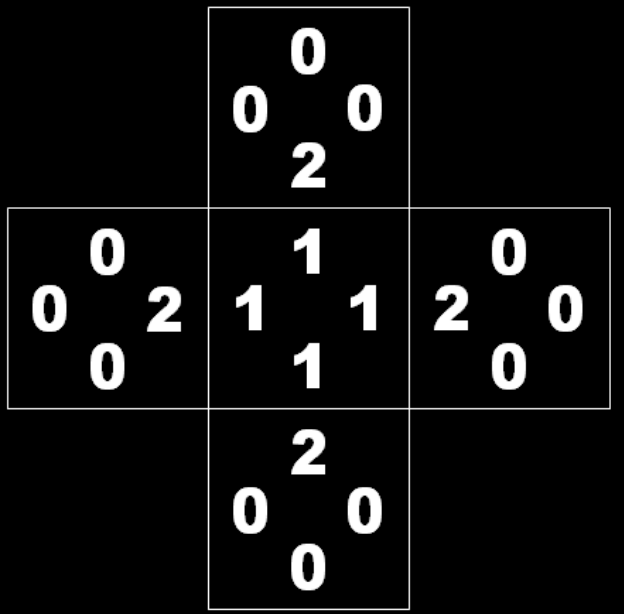
- *Do this 5 times (2^5=32 threads)*
- Result:
  Gather offset to
  furthest connected cells!
- -> max-gather
- (Same for con.y and reverse-directed links)



# Links: Directions

- One entry for each cell and each direction
- Example: 4-connectivity links for a cross of connected cells:



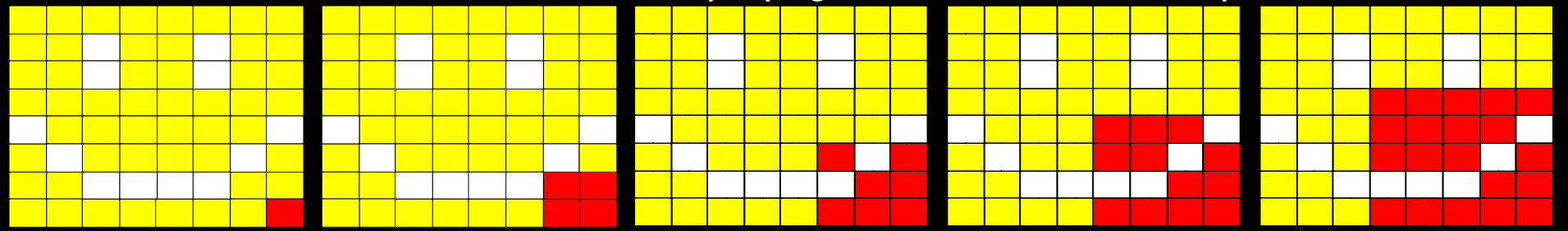



conn_up, conn_down, conn_left, conn_right

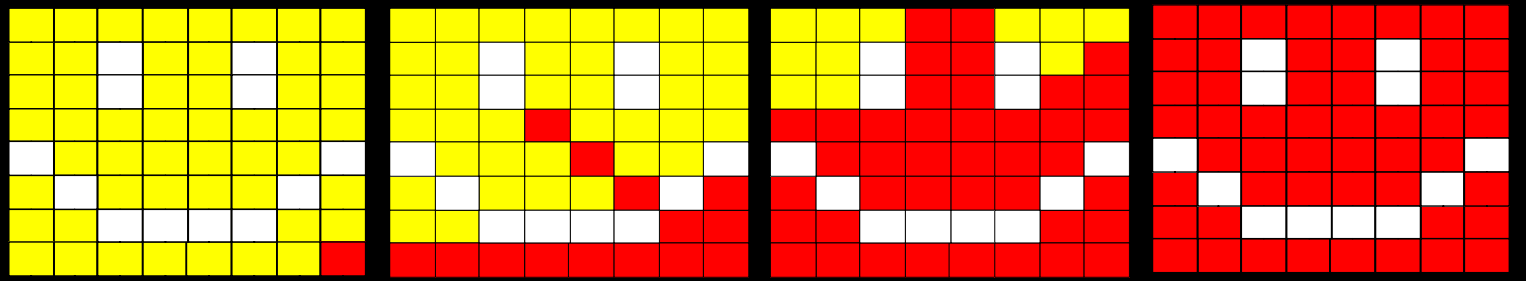
# Links: Faster Gathering

- Allows for far-away gathering: **max-gather**.

Without Links: label propagation @ one cell each pass



With Links: label propagation @ full distance in each direction

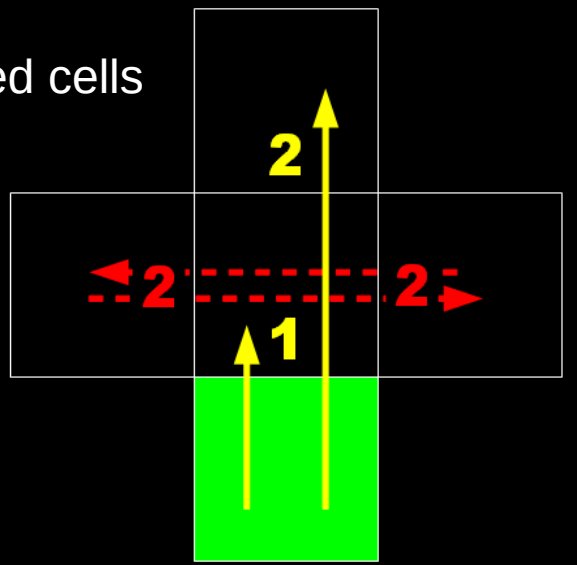


Example based on 8-connectivity, red = largest label

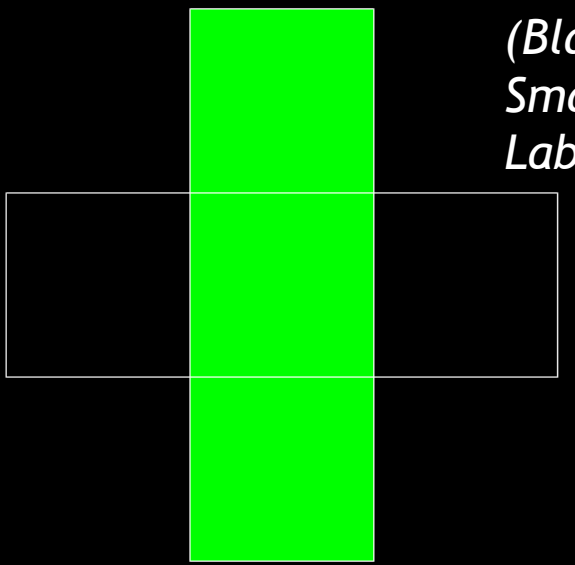
# Max-gather doesn't suffice

- Assumption: 1-gather is not necessary anymore.
- BUT: Cases where max-gather doesn't fill all cells!

Example:
Cross of connected cells




Green Label is largest -
Attempted max-gathering

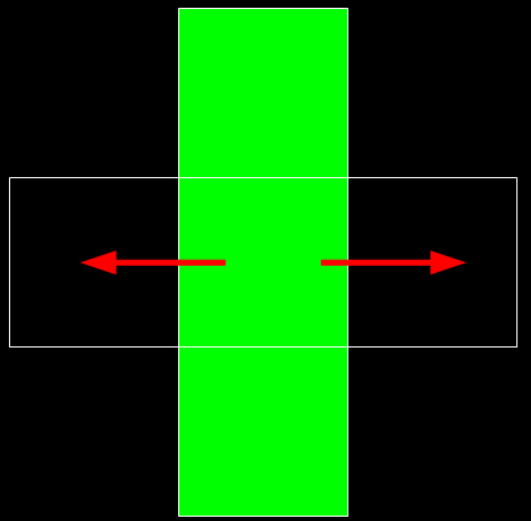


Label result **incomplete**!

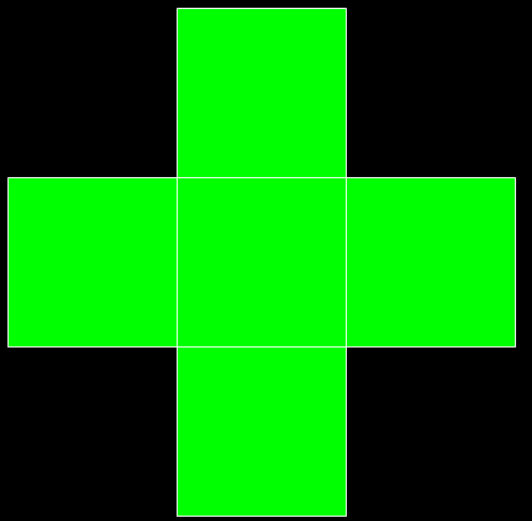
*(Black label color = Smaller/Irrelevant Label)*

# Max-gather doesn't suffice

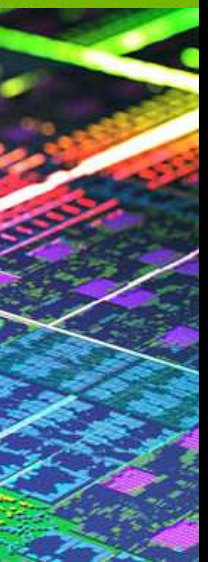
- 1-gather is still necessary to propagate labels at T-crossings!



Green Label is largest - Attempted 1-gathering



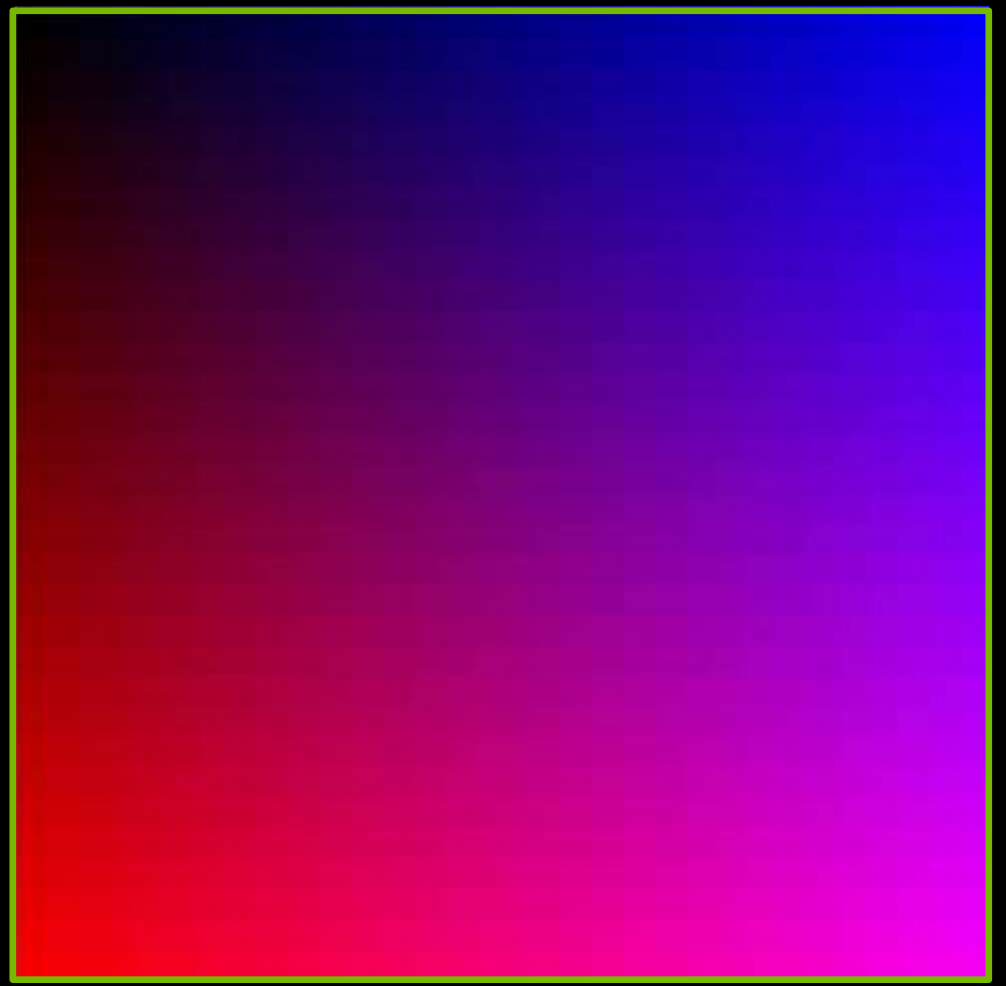
Label result (complete)



# Q/A: Shared memory versus Shuffle

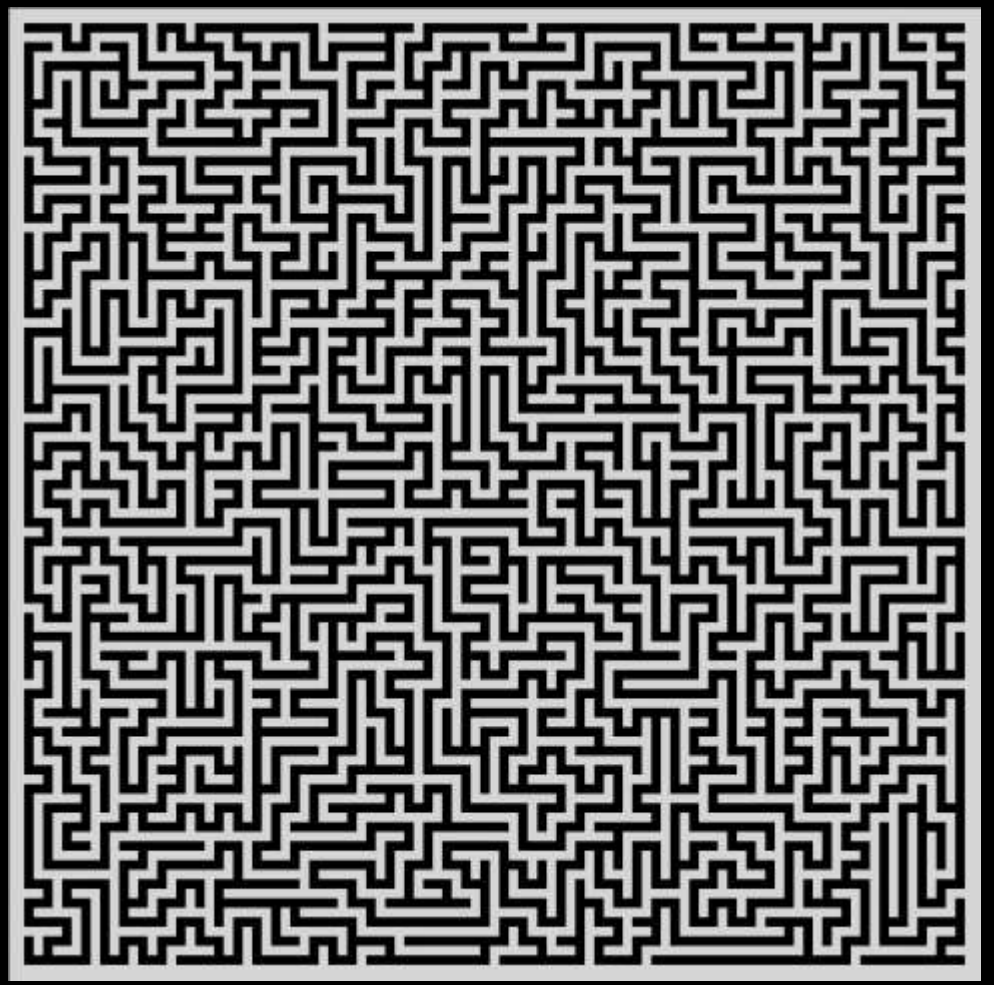
- Even Shuffle implementation requires shmem …
- Why faster?
  - Shared memory requires __syncthreads(), and address computations
  - max-gather's unpredictable label access patterns create **shared memory bank** conflicts (stalling the threads)
  - Shared memory only used for 2D transpose
    - Transpose avoids shmem bank conflicts of labels via [33][33] allocation!

# Movie: 1- and Max-Gather

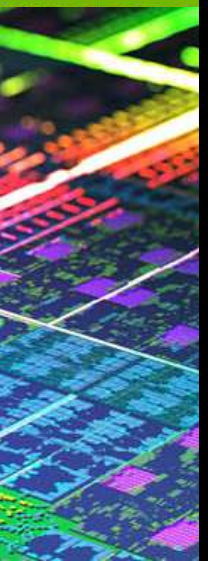


- Movie: A look “inside” a threadblock.
- Max-gather makes label propagate much faster
- Notice how left region would not be correct without vertical 1-gather!

# Movie: Whole input (1- and max-gather)



- Threadblocks converge
- 1-gather only:
  SHMEM/SHFL: 39/32 ms
- max-gather:
  No visible difference outside threadblocks, only *faster*:
  SHMEM/SHFL: 26/23 ms



# Movie: Whole input (1- and max-gather & RootLabel)

- Presented at GTC2010
- Uses every label's "root" (origin position) to gather additional label updates!
- (global optimization, no shuffle/shmem)
- Fastest combination! SHMEM/SHFL: 12/12 ms

# Results: Typical execution times

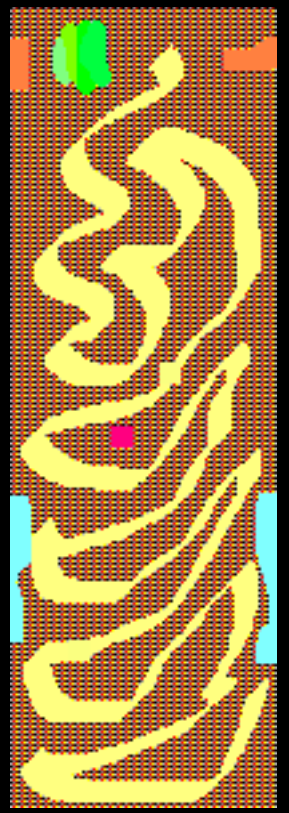
| Image | Kernel | Label updates | Time (ms) |
|---|---|---|---|
| Maze, 512x512 | 1-gather | SHMEM | 44 ms |
| Maze, 512x512 | 1-gather | SHUFFLE | 39 ms |
| Maze, 512x512 | 1- and max-gather | SHMEM | 26 ms |
| Maze, 512x512 | 1- and max-gather | SHUFFLE | 23 ms |
| Maze, 512x512 | 1- and max-gather, LabelRoot(*) | SHMEM | 12 ms |
| Maze, 512x512 | 1- and max-gather, LabelRoot(*) | SHUFFLE | 12 ms |

- Fast enough for video processing!
- Fast enough for interactive segmentation (change of threshold, etc.)

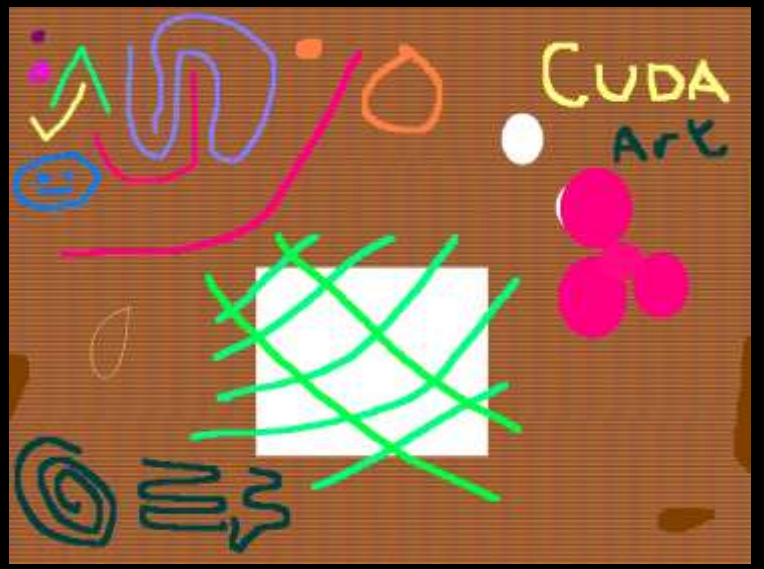
Run on GeForce GTX680, cumulated kernel timings

# Results: Input Images

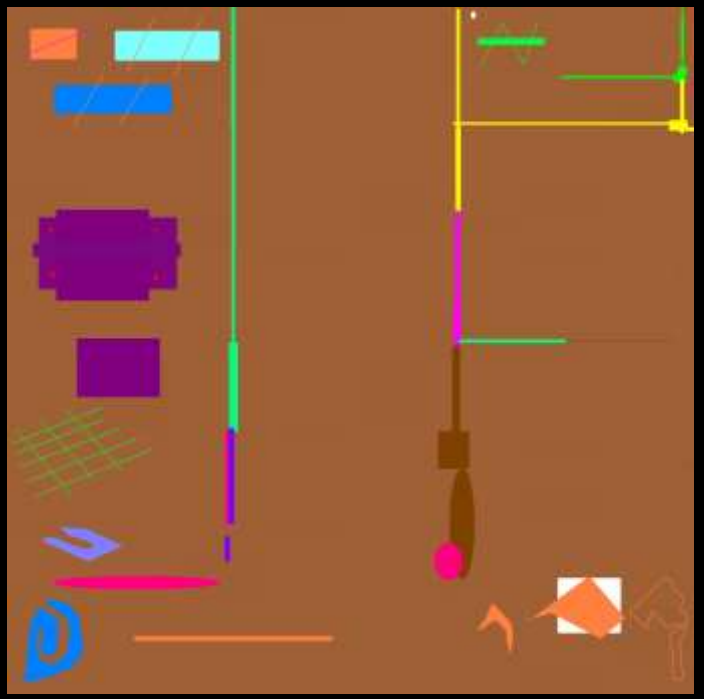
- Used in CUDA TopCoder challenge



100by300



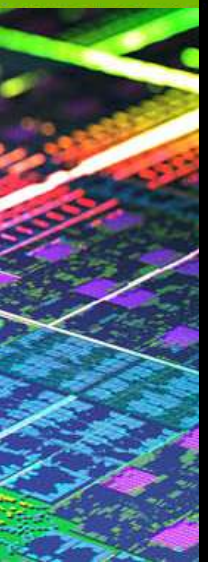

1Kby768



4Kby4K

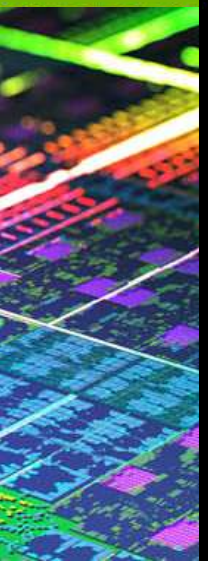
# Results: Typical execution times

| | GTC2010 | GTC2013 | | | |
|---|---|---|---|---|---|
| Image | 1-/max-gather & LabelRoot | 1-/max-gather | | 1-/max-gather & LabelRoot | |
| | | SHMEM | SHUFFLE | SHMEM | SHUFFLE |
| 100by300 | 6.27 ms | 3.0 ms | 2.57 ms | 1.66 ms | 1.5 ms |
| 1Kby768 | 8.2 ms | 8.9 ms | 7.15 ms | 4.6 ms | 3.7 ms |
| 4Kby4K | 385 ms | 380 ms | 266 ms | 112 ms | 85 ms |



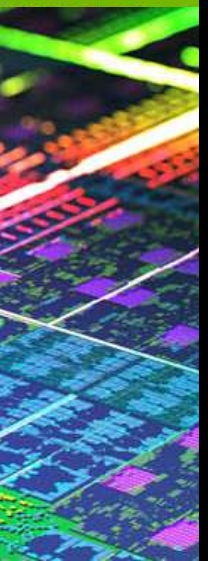
# Summary

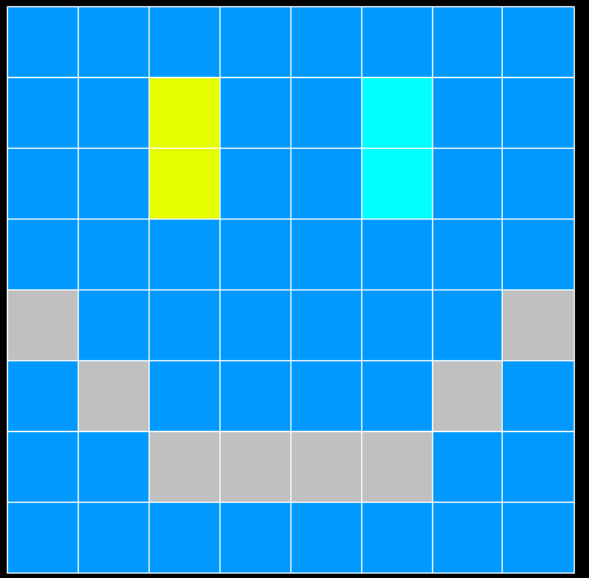
- Shuffle instruction speeds up internal label propagation.
- Horizontal communication pattern of Shuffle is not an issue: shared memory still needed, but *only for 2D transpose*.
- max-gather offsets, precomputed from cell connectivity speeds up label propagation considerably.
  Shmem bank conflicts in label gathers avoided through shuffle.
- Completely data-parallel, gather-based algorithm
  (no atomic operations)
- LabelRoot gives final boost, despite random gmem fetches



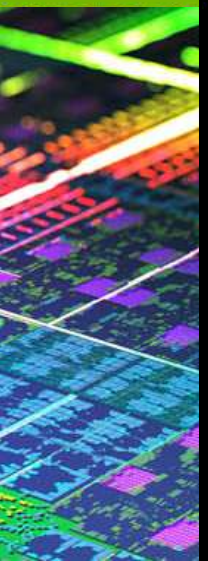
# Future Work

- Label List generation (based on data compaction)
- 8-connectivity
- 3D implementation
- Investigate thread-private arrays in lmem,
  e.g. by having each thread handle 32 labels, and
  communicating label updates via shuffle.
- PTX-level optimization

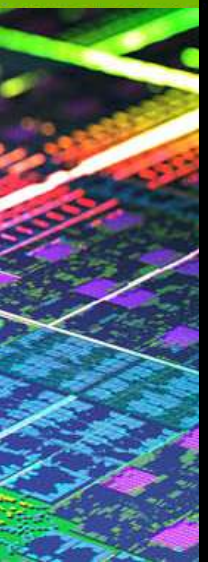




Thank you !

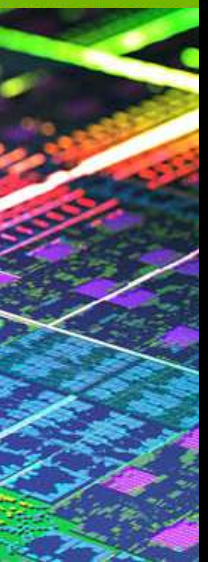


# Additional Material



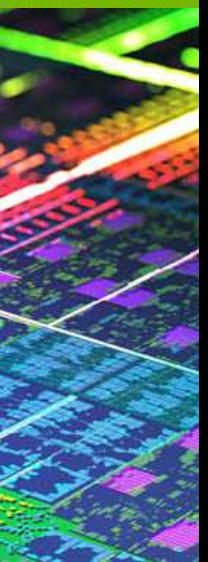
# Insights on Shuffle (3)

- But shmem is still needed, despite shuffle?
- YES. But:
  Transpose via shmem is now a perfectly regular access pattern
- Remaining shmem bank conflicts are easily addressed by allocating the labels in 33x33 array (instead of 32x32).
  (More details on shmem bank conflicts, see Programming Guide)



# Insights on Shuffle

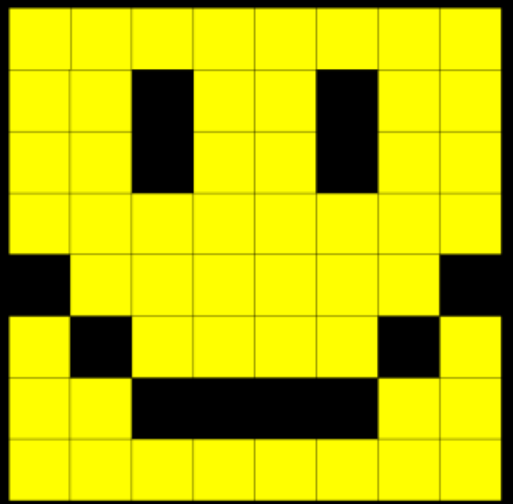
- Shuffle is best for reductions, where it can save considerable amounts of shared memory
- But in X-Y communication patterns beyond 32 participating threads, you _still_ need the full amount of shared memory!
- BUT: The irregular access patterns that were necessary before (remember link propagation) and caused shmem bank conflicts now happen using the SHFL-command!
  Shmem has totally regular and predictable accesses (transpose).



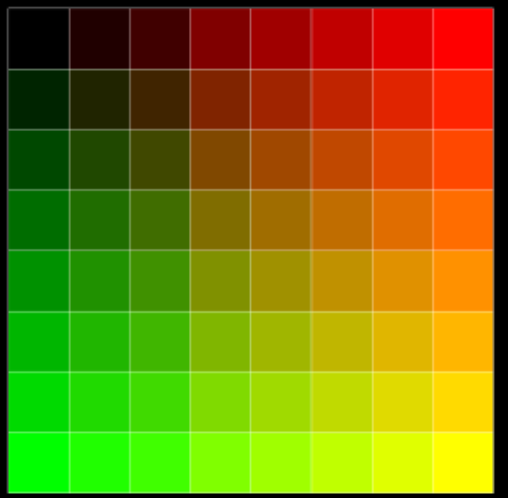
# Algorithm Optimization: LabelRoot

# Root cells

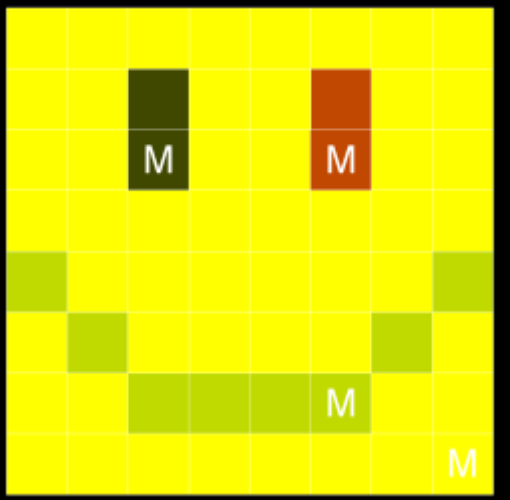
- In each region, one cell keeps its original label
- All other cells: Their label originates from this one cell
- Thus, each labelled region has a ***label root***



Label Init: Lower/Right values are larger



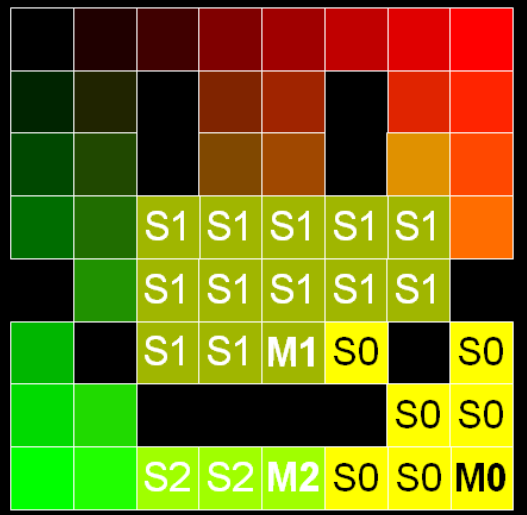
Label Init: Lower/Right values are larger



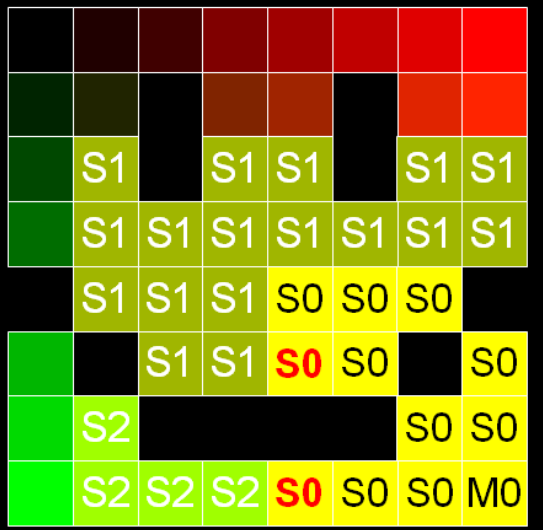

Labelled result
M = label root cell

# Root cells: Label propagation

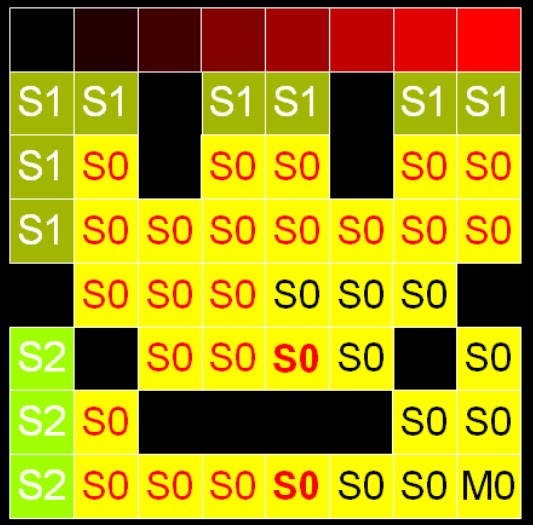
- If root changes label, all dependent cells may change label
- Hence: Always gather current label from label root cell!
- Purpose: Commonly labelled regions flip “at once”.



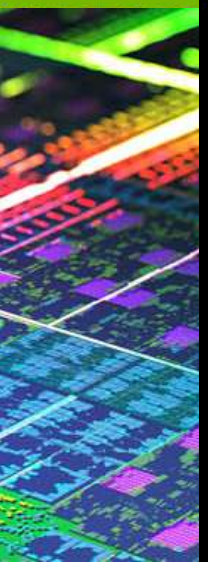

Pass 0: Three regions: Roots M*n*, Deps S*n*




Pass 1: Region M0 "captures" Roots M1, M2

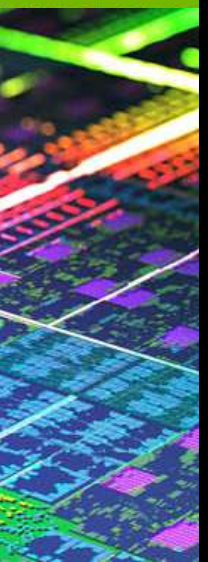



Pass 2: Root cell lookup makes S0's and S1's flip!



# Pseudo-Code: Simple Algorithm

```
// Step I - Label Init
for (all pixels) {
  pixel.label = encodeLabel(pixel.x, pixel.y);
}

// Step II - Propagate Labels
while (AnyLabelChanges) {
  for (all pixels) {
    for (all directions) {
      neighborLabel = gather(neighbor, direction);
      pixel.label = max(pixel.label, neighborLabel);
    }
  }
}
```



# Pseudo-Code: Optimized Algorithm

```
// Step I - Label Init
for (all pixels)
  pixel.label = encodeLabel(pixel.x, pixel.y);
// Precalculate links
loadconnectivitybits(); precomputeLinks();
// Step II - Propagate Labels
while (AnyLabelChanges) {
  for (all pixels) {
     for (all directions) {
       // Use max-gather
       neighborLabel1 = gather(neighbor, direction);
       neighborLabelMax = gather(neighbor, pixel.maxgather(direction));
       pixel.label = max(pixel.label, neighborLabel1, neighborLabelMax);
       // LabelRoot
       if (pixel.label != pixel.originalLabel) {
         rootRef = decodeLabel(pixel.label);
         pixel.label = max(pixel.label, rootRef.label); }}}}
```

# Algorithm Summary

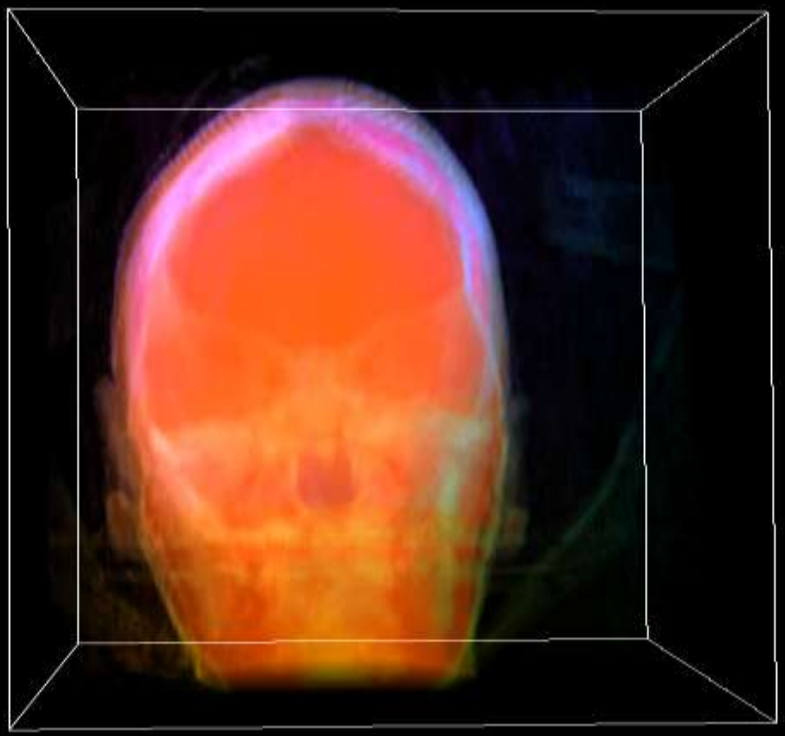
- Simple Algorithm
  - Initialize cells with unique labels
  - While not converged (changes occur), *for each cell*:
    - Gather labels from connected, neighboring cells (1-gather).
    - If gathered label greater than own Update own label !

Optimized Algorithm

- Initialize cells with unique labels
- Precompute Links from connectivity
- While not converged (changes occur), *for each cell*:
  - Gather labels from neighboring cells (1-gather).
  - Gather labels from far-away cells via Links (max-gather).
  - Gather label from current label root (if a dependent)
  - If any gathered label greater than own: Update own label !

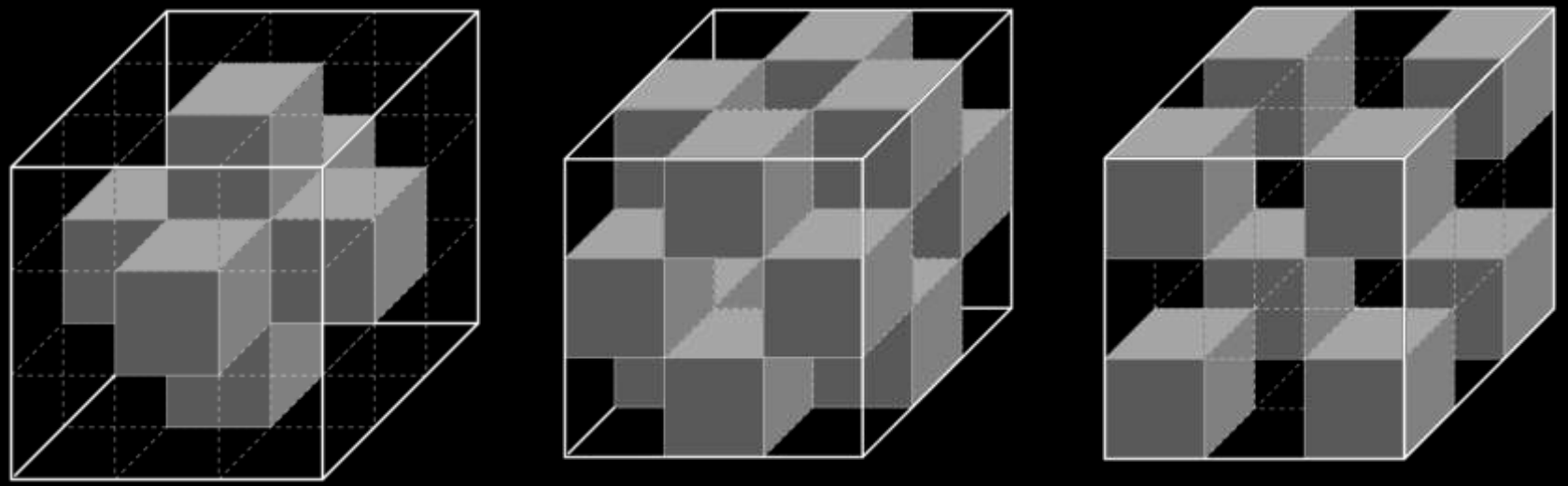
# Extension to 3D

- Extend algorithm to 3D (cells = voxels)
- Choice of connectivity scheme
- Labels are now a function of x,y,z
- Labels can be converted to and from 3D coordinates
- 8bit x,y,z -> RGB 8bit



# 3D Connectivity

- Choice of connectivity scheme from three building blocks:



# Label lists (Sketch)

- *Q: How can I extract a list of all discovered regions?*
- Step 1: Each region has one master cell.
  Isolate all cells that have retained their own label!
- Step 2: With list of master cells and their labels, each region's cells can be extracted by filtering for that label.
- Both steps can be solved by Data Compaction!
  (e.g. Thrust (Scan), HistoPyramids)
- Future Work!