Parallel Prefix Sum on the GPU (Scan)

Presented by Adam O'Donovan

Slides adapted from the online course slides for ME964 at Wisconsin taught by Prof. Dan Negrut and from slides Presented by David Luebke

Parallel Prefix Sum (Scan)

- Definition:

  The all-prefix-sums operation takes a binary associative operator $\oplus$ with identity I, and an array of n elements

  $$[a_0, a_1, \ldots, a_{n-1}]$$

  and returns the ordered set

  $$[I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})].$$

- Example:
  if $\oplus$ is addition, then scan on the set

  [3 1 7 0 4 1 6 3]

  returns the set

  [0 3 4 11 11 15 16 22]

Exclusive scan: last input element is not included in the result

(From Blelloch, 1990, "Prefix Sums and Their Applications)

Applications of Scan

- Scan is a simple and useful parallel building block
  - Convert recurrences from sequential …

```
for(j=1;j<n;j++)
   out[j] = out[j-1] + f(j);
```

  - … into parallel:

```
forall(j) in parallel
     temp[j] = f(j);
scan(out, temp);
```

- Useful in implementation of several parallel algorithms:

- radix sort
- quicksort
- String comparison
- Lexical analysis
- Stream compaction

- Polynomial evaluation
- Solving recurrences
- Tree operations
- Histograms
- Etc.

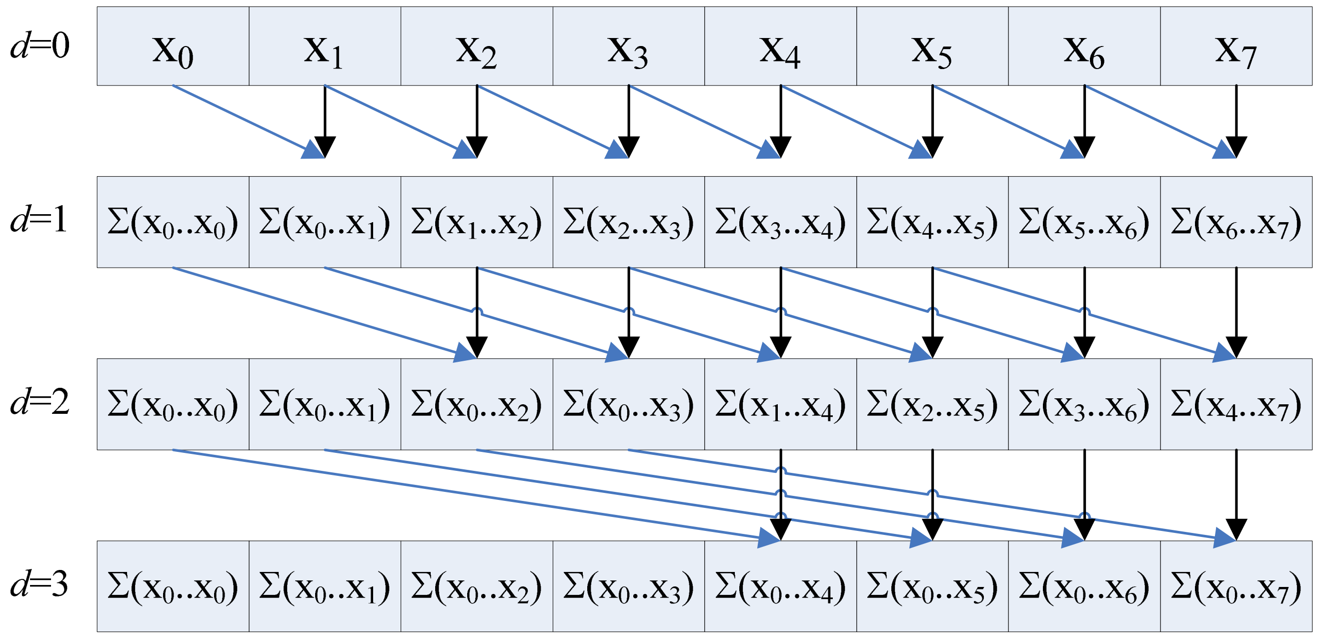
Scan on the CPU

```
void scan( float* scanned, float* input, int length)
{
  scanned[0] = 0;
  for(int i = 1; i < length; ++i)
  {
    scanned[i] = scanned[i-1] + input[i-1];
  }
}
```

- Just add each element to the sum of the elements before it
- Trivial, but sequential
- Exactly *n-1* adds: optimal in terms of work efficiency

Parallel Scan Algorithm: Solution One Hillis & Steele (1986)

- Note that a implementation of the algorithm shown in picture requires two buffers of length n (shown is the case $n=8=2^3$)
- Assumption: the number n of elements is a power of 2: $n=2^M$



d=0
x0 x1 x2 x3 x4 x5 x6 x7
d=1
Σ(x0..x0) Σ(x0..x1) Σ(x1..x2) Σ(x2..x3) Σ(x3..x4) Σ(x4..x5) Σ(x5..x6) Σ(x6..x7)
d=2
Σ(x0..x0) Σ(x0..x1) Σ(x0..x2) Σ(x0..x3) Σ(x1..x4) Σ(x2..x5) Σ(x3..x6) Σ(x4..x7)
d=3
Σ(x0..x0) Σ(x0..x1) Σ(x0..x2) Σ(x0..x3) Σ(x0..x4) Σ(x0..x5) Σ(x0..x6) Σ(x0..x7)

Picture courtesy of Mark Harris

The Plain English Perspective

- First iteration, I go with stride $1=2^0$
  - Start at $x[2^M]$ and apply this stride to all the array elements before $x[2^M]$ to find the mate of each of them. When looking for the mate, the stride should not land you before the beginning of the array. The sum replaces the element of higher index.
    - This means that I have 2^M-1 additions

- Second iteration, I go with stride $2=2^1$
  - Start at $x[2^M]$ and apply this stride to all the array elements before $x[2^M]$ to find the mate of each of them. When looking for the mate, the stride should not land you before the beginning of the array. The sum replaces the element of higher index.
    - This means that I have $2^M – 2^1$ additions

- Third iteration: I go with stride $4=2^2$
  - Start at $x[2^M]$ and apply this stride to all the array elements before $x[2^M]$ to find the mate of each of them. When looking for the mate, the stride should not land you before the beginning of the array. The sum replaces the element of higher index.
    - This means that I have $2^M – 2^2$ additions

- … (and so on)

The Plain English Perspective

- Consider the kth iteration (k is some arbitrary valid integer): I go with stride 2^{k-1}
  - Start at $x[2^M]$ and apply this stride to all the array elements before $x[2^M]$ to find the mate of each of them. When looking for the mate, the stride should not land you before the beginning of the array. The sum replaces the element of higher index.
    - This means that I have 2^M-2^{k-1} additions

- …

- Mth iteration: I go with stride 2^{M-1}
  - Start at $x[2^M]$ and apply this stride to all the array elements before $x[2^M]$ to find the mate of each of them. When looking for the mate, the stride should not land you before the beginning of the array. The sum replaces the element of higher index.
    - This means that I have 2^M-2^{M-1} additions

- NOTE: There is no (M+1)th iteration since this would automatically put me beyond the bounds of the array (if you apply an offset of 2^M to "$\&x[2^M]$ " it places you right before the beginning of the array – not good…)

Hillis & Steele Parallel Scan Algorithm

- Algorithm looks like this:

```
for $d := 0$ to M-1 do
    forall $k$ in parallel do
            if $k - 2^d \geq 0$ then
                    $x[out][k] := x[in][k] + x[in][k - 2^d]$
            else
                    $x[out][k] := x[in][k]$
    endforall
    swap($in,out$)
endfor
```

Double-buffered version of the sum scan

Operation Count
Final Considerations

- The number of operations tally:
  - $(2^M-2^0) + (2^M-2^1) + \ldots + (2^M-2^k) + \ldots + (2^M-2^{M-1})$
  - Final operation count:

$$M \cdot 2^M - (2^0 + \ldots + 2^{M-1}) = M \cdot 2^M - 2^M + 1 = n(\log(n) - 1) + 1$$

  - This is an algorithm with O(n*log(n)) work

- This scan algorithm is not that work efficient
  - Sequential scan algorithm does *n-1* adds
  - A factor of log(n) might hurt: 20x more work for 10^6 elements!

- A parallel algorithm can be slow when execution resources are saturated due to low algorithm efficiency

Hillis & Steele: Kernel Function

```
__global__ void scan(float *g_odata, float *g_idata, int n)
{
    extern  __shared__  float temp[]; // allocated on invocation

    int thid = threadIdx.x;
    int pout = 0, pin = 1;

    // load input into shared memory.
    // Exclusive scan: shift right by one and set first element to 0
    temp[thid] = (thid > 0) ? g_idata[thid-1] : 0;
    __syncthreads();

    for( int offset = 1; offset < n; offset <<= 1 )
    {
        pout = 1 - pout; // swap double buffer indices
        pin  = 1 - pout;

        if (thid >= offset)
            temp[pout*n+thid] += temp[pin*n+thid - offset];
           else
            temp[pout*n+thid] = temp[pin*n+thid];

        __syncthreads();
    }

    g_odata[thid] = temp[pout*n+thid1]; // write output
}
```

Hillis & Steele: Kernel Function, Quick Remarks

- The kernel is very simple, which is good
- Note the nice trick that was used to swap the buffers
- The kernel only works when the entire array is processed by one block
  - One block in CUDA has 512 threads, which means I can have up to 1024 elements
  - This needs to be improved upon, can't limit solution to what's been presented so far

Improving Efficiency

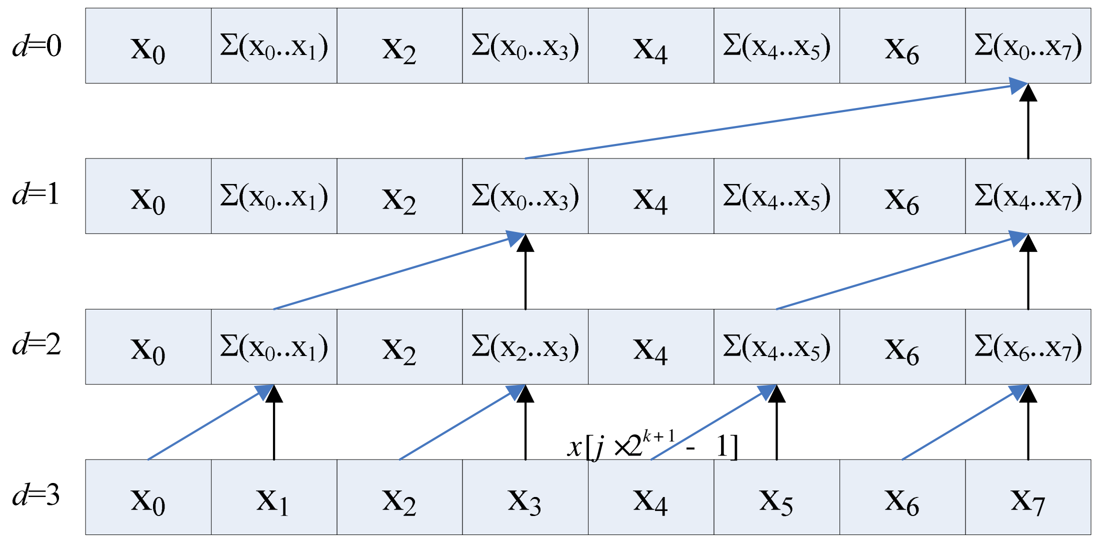
- A common parallel algorithm pattern:

  <u>*Balanced Trees*</u>

  - Build a balanced binary tree on the input data and sweep it to and then from the root
  - Tree is not an actual data structure, but a concept to determine what each thread does at each step

- For scan:
  - Traverse down from leaves to root building partial sums at internal nodes in the tree
    - Root holds sum of all leaves (this is a reduction algorithm!)
  - Traverse back up the tree building the scan from the partial sums

Picture and Pseudocode ~ Reduction Step~



d=0
x0 Σ(x0..x1) x2 Σ(x0..x3) x4 Σ(x4..x5) x6 Σ(x0..x7)
d=1
x0 Σ(x0..x1) x2 Σ(x0..x3) x4 Σ(x4..x5) x6 Σ(x4..x7)
d=2
x0 Σ(x0..x1) x2 Σ(x2..x3) x4 Σ(x4..x5) x6 Σ(x6..x7)
d=3
x0 x1 x2 x3 x4 x5 x6 x7
x[j ×2^{k+1} - 1]

i1 =

$$\begin{matrix} 1 & 3 & 5 & 7 \\ 3 & 7 & -1 & -1 \\ 7 & -1 & -1 & -1 \end{matrix}$$

i2 =

$$\begin{matrix} 0 & 2 & 4 & 6 \\ 1 & 5 & -1 & -1 \\ 3 & -1 & -1 & -1 \end{matrix}$$

NOTE: “-1” entries indicate no-ops

```
for k=0 to M-1
      offset = $2^k$
      for j=1 to $2^{M-k-1}$ in parallel do
            $x[j \cdot 2^{k+1}-1] = x[j \cdot 2^{k+1}-1] + x[j \cdot 2^{k+1}-2^k-1]$
      endfor
endfor
```

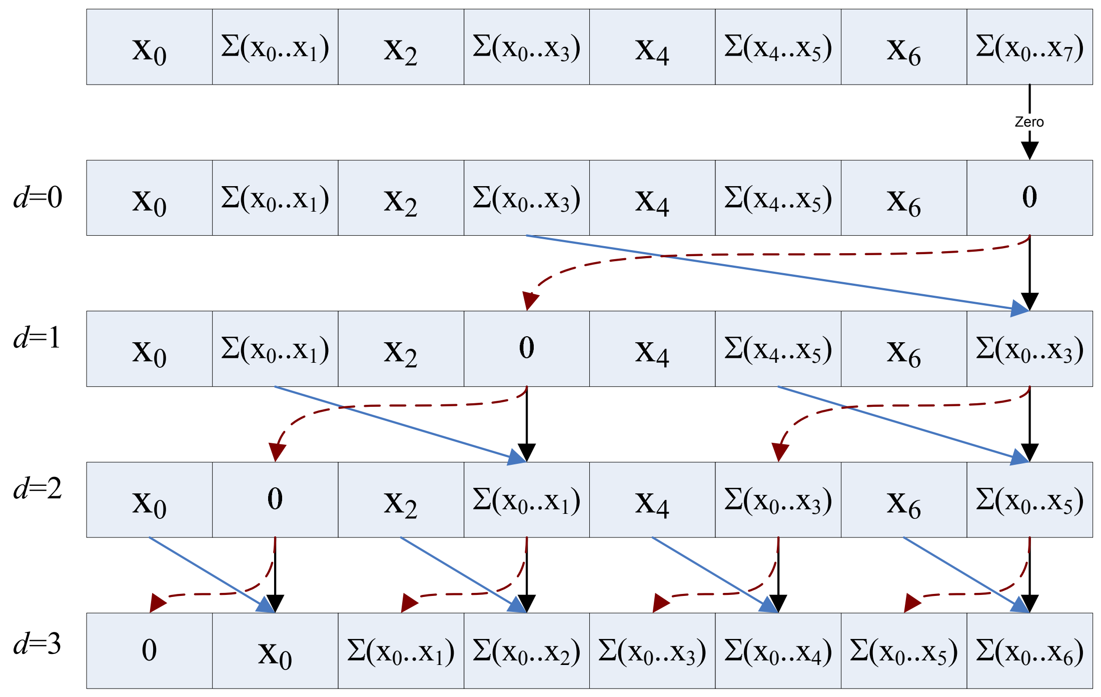
Operation Count, Reduce Phase

```
for k=0 to M-1
        offset = $2^k$
        for j=1 to $2^{M-k-1}$ in parallel do
                x[$j \cdot 2^{k+1}-1$] = x[$j \cdot 2^{k+1}-1$] + x[$j \cdot 2^{k+1}-2^k-1$]
        endfor
endfor
```

By inspection: $\sum_{k=0}^{M-1} 2^{M-k-1} = 2^M - 1 = n - 1$

Looks promising…

The Down-Sweep Phase



x_0 | $\Sigma(x_0..x_1)$ | x_2 | $\Sigma(x_0..x_3)$ | x_4 | $\Sigma(x_4..x_5)$ | x_6 | $\Sigma(x_0..x_7)$
Zero
$d=0$ x_0 | $\Sigma(x_0..x_1)$ | x_2 | $\Sigma(x_0..x_3)$ | x_4 | $\Sigma(x_4..x_5)$ | x_6 | 0
$d=1$ x_0 | $\Sigma(x_0..x_1)$ | x_2 | 0 | x_4 | $\Sigma(x_4..x_5)$ | x_6 | $\Sigma(x_0..x_3)$
$d=2$ x_0 | 0 | x_2 | $\Sigma(x_0..x_1)$ | x_4 | $\Sigma(x_0..x_3)$ | x_6 | $\Sigma(x_0..x_5)$
$d=3$ 0 | x_0 | $\Sigma(x_0..x_1)$ | $\Sigma(x_0..x_2)$ | $\Sigma(x_0..x_3)$ | $\Sigma(x_0..x_4)$ | $\Sigma(x_0..x_5)$ | $\Sigma(x_0..x_6)$

NOTE: This is just a mirror image of the reduction stage. Easy to come up with the indexing scheme…

```
for k=M-1 to 0
      offset = $2^k$
      for j=1 to $2^{M-k-1}$ in parallel do
            dummy = x[$j \cdot 2^{k+1} - 2^k - 1$]
            x[$j \cdot 2^{k+1} - 2^k - 1$] = x[$j \cdot 2^{k+1} - 1$]
            x[$j \cdot 2^{k+1} - 1$] = x[$j \cdot 2^{k+1} - 1$] + dummy
      endfor
endfor
```

Down-Sweep Phase, Remarks

- Number of operations for the down-sweep phase:
  - Additions: n-1
  - Swaps: n-1 (each swap shadows an addition)

- Total number of operations associated with this algorithm
  - Additions: 2n-2
  - Swaps: n-1
  - Looks very comparable with the work load in the sequential solution

- The algorithm is convoluted though, it won’t be easy to implement
  - Kernel shown on next slide

```
 __global__ void prescan(float *g_odata, float *g_idata, int n)
{
    extern  __shared__  float temp[];// allocated on invocation


    int thid = threadIdx.x;
    int offset = 1;

    temp[2*thid]   = g_idata[2*thid]; // load input into shared memory
    temp[2*thid+1] = g_idata[2*thid+1];

    for (int d = n>>1; d > 0; d >>= 1) // build sum in place up the tree
    {
      __syncthreads();

      if (thid < d)
      {
          int ai = offset*(2*thid+1)-1;
          int bi = offset*(2*thid+2)-1;

          temp[bi] += temp[ai];
      }
      offset *= 2;
    }

    if (thid == 0) { temp[n - 1] = 0; } // clear the last element

    for (int d = 1; d < n; d *= 2) // traverse down tree & build scan
    {
        offset >>= 1;
        __syncthreads();

        if (thid < d)
        {
           int ai = offset*(2*thid+1)-1;
           int bi = offset*(2*thid+2)-1;

           float t   = temp[ai];
           temp[ai]  = temp[bi];
           temp[bi] += t;
         }
    }

    __syncthreads();

    g_odata[2*thid]   = temp[2*thid]; // write results to device memory
    g_odata[2*thid+1] = temp[2*thid+1];
}
```

Bank Conflicts

- Current implementation has many ShMem bank conflicts
  - Can significantly hurt performance on current GPU hardware
  - The source of the conflicts: linear indexing with stride that is a power of 2 multiple of thread id (see below): "$j \cdot 2^{k+1}-1$"

```
for k=0 to M-1
      offset = $2^k$
      for j=1 to $2^{M-k-1}$ in parallel do
            x[$j \cdot 2^{k+1}-1$] = x[$j \cdot 2^{k+1}-1$] + x[$j \cdot 2^{k+1}-2^k-1$]
      endfor
endfor
```

- Simple modifications to current memory addressing scheme can save a lot of cycles

Bank Conflicts

- Occur when multiple threads access the same shared memory bank with different addresses
  - In our case, we have something like $2^{k+1} \cdot j-1$
    - k=0: two way bank conflict
    - k=1: four way bank conflict
    - …
- No penalty if all threads access different banks
  - Or if all threads access exact same address
- Recall that shared memory accesses with conflicts are serialized
  - N-bank memory conflicts lead to a set of N successive shared memory transactions

Initial Bank Conflicts on Load

- Each thread loads two shared mem data elements

- Tempting to interleave the loads (see lines 9 & 10, and 46 & 47)

```
temp[2*thid]   = g_idata[2*thid];
temp[2*thid+1] = g_idata[2*thid+1];
```

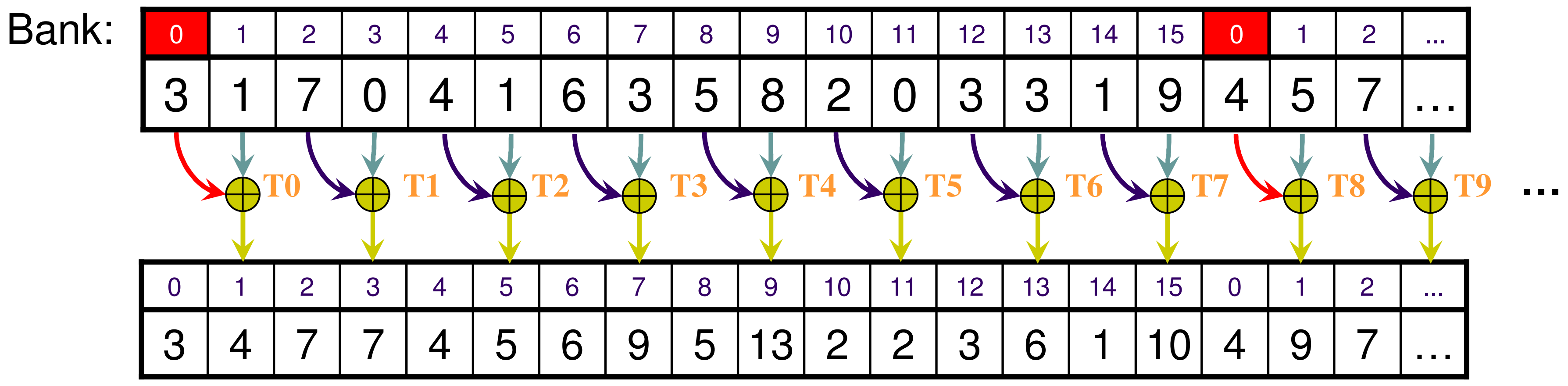
  - Thread 0 accesses banks 0 and 1
  - Thread 1 accesses banks 2 and 3
  - …
  - Thread 8 accesses banks 16 and 17. Oops, that's 0 and 1 again…
    - Two way bank conflict, can't be easily eliminated

- Better to load one element from each half of the array

```
temp[thid]           = g_idata[thid];
temp[thid + (n/2)] = g_idata[thid + (n/2)];
```

Bank Conflicts in the tree algorithm

- When we build the sums, during the first iteration of the algorithm each thread in a half-warp reads two shared memory locations and writes one:
- Th(0,8) access bank 0



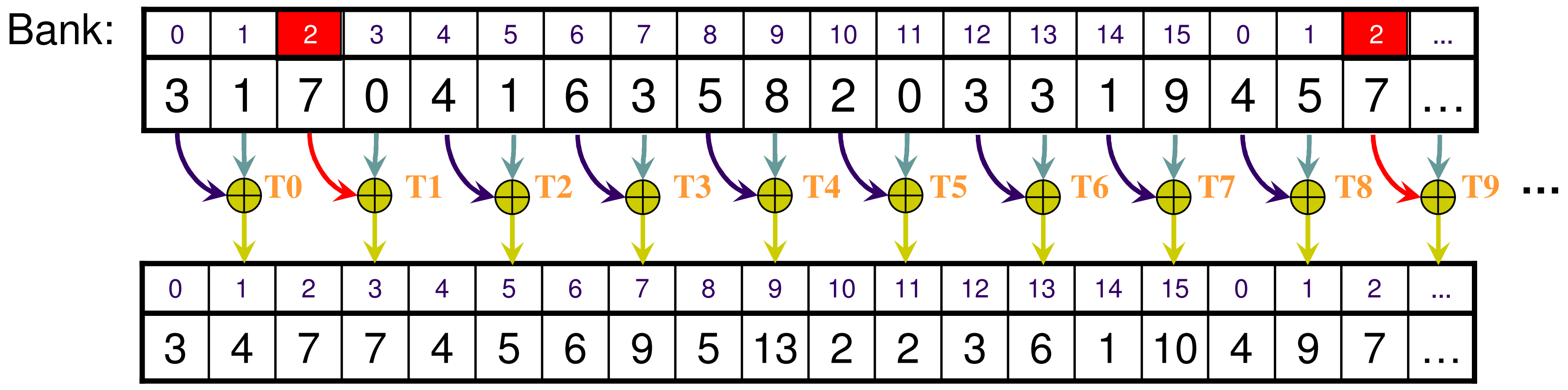
First iteration: 2 threads access each of 8 banks.

Each ⊕ corresponds to a single thread.

Like-colored arrows represent simultaneous memory accesses

Bank Conflicts in the tree algorithm

- When we build the sums, each thread reads two shared memory locations and writes one:
- Th(1,9) access bank 2, etc.



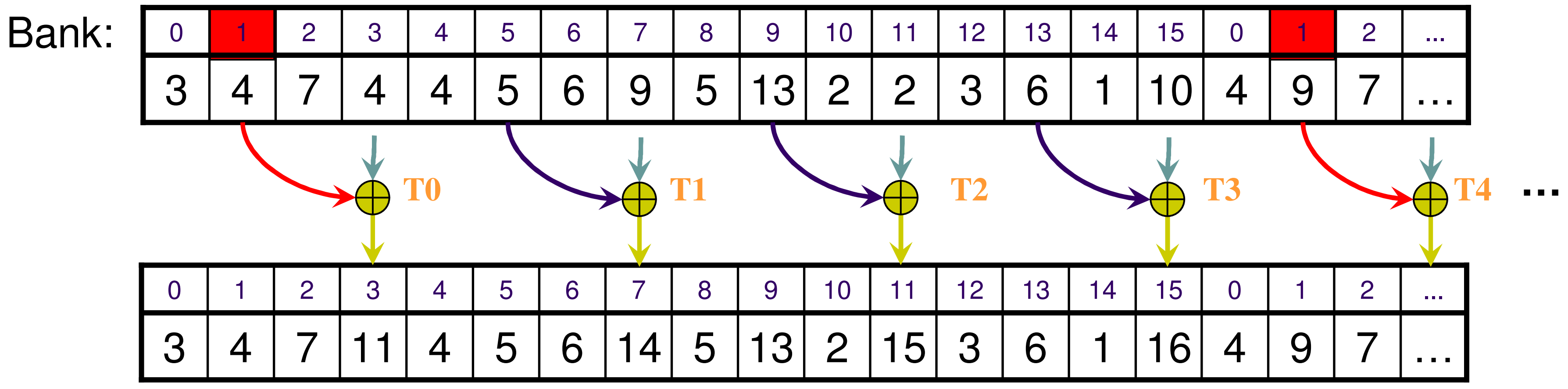
First iteration: 2 threads access each of 8 banks.

Each ⊕ corresponds to a single thread.

Like-colored arrows represent simultaneous memory accesses

Bank Conflicts in the tree algorithm

- 2nd iteration: even worse!
  - 4-way bank conflicts; for example:
    Th(0,4,8,12) access bank 1, Th(1,5,9,13) access Bank 5, etc.



2nd iteration: 4 threads access each of 4 banks.

Each ⊕ corresponds to a single thread.

Like-colored arrows represent simultaneous memory accesses

Managing Bank Conflicts in the Tree Algorithm

- Use padding to prevent bank conflicts
  - Add a word of padding every 16 words.
    - Now you work with a virtual 17 bank shared memory layout
  - Within a 16-thread half-warp, all threads access different banks
    - They are aligned to a 17 word memory layout
  - It comes at a price: you have memory words that are wasted
  - Keep in mind: you should also load data from global into shared memory using the virtual memory layout of 17 banks

Use Padding to Reduce Conflicts

- After you compute a ShMem address like this:

```
Address = 2 * stride * thid;
```

- Add padding like this:

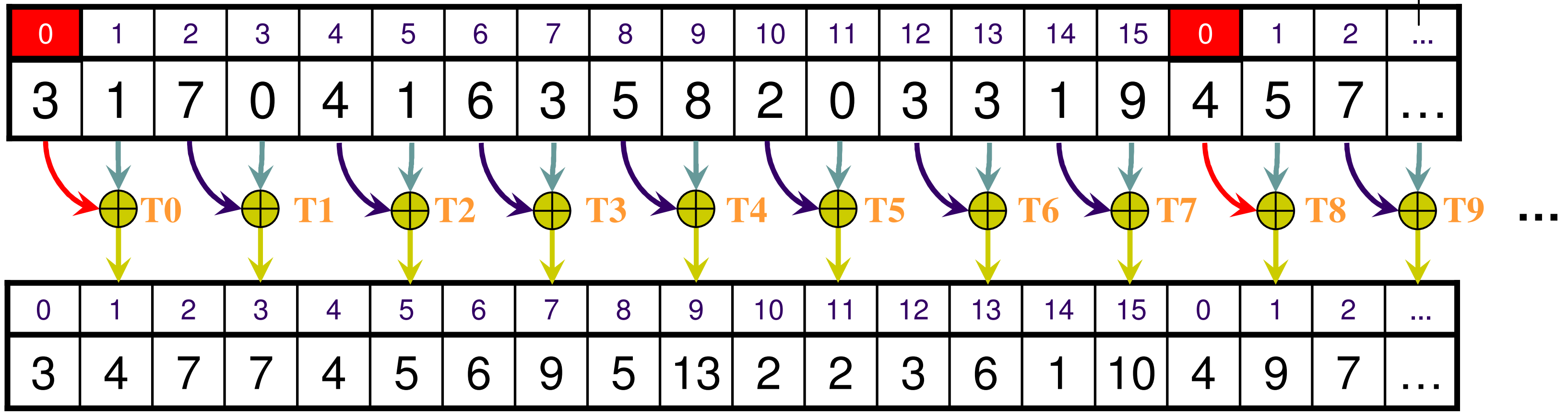
```
Address += (address >> 4); // divide by
NUM_BANKS
```

- This removes most bank conflicts

Managing Bank Conflicts in the Tree Algorithm

Original scenario.

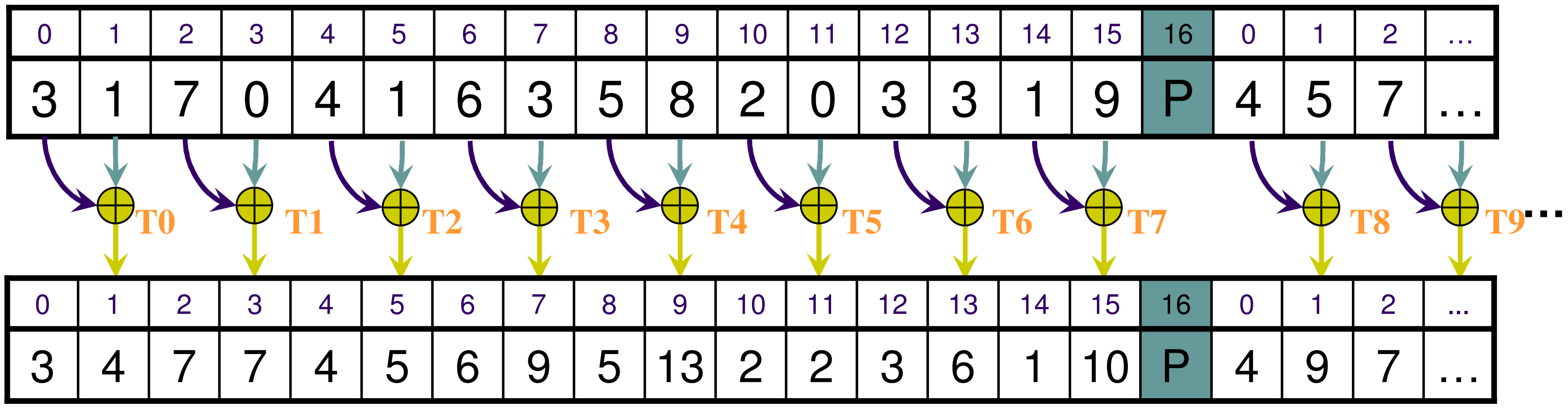
Bank:



Modified scenario, virtual 17 bank memory layout.

Actual physical memory (true bank number)
(0) (1) (2) (3)

Virtual Bank:



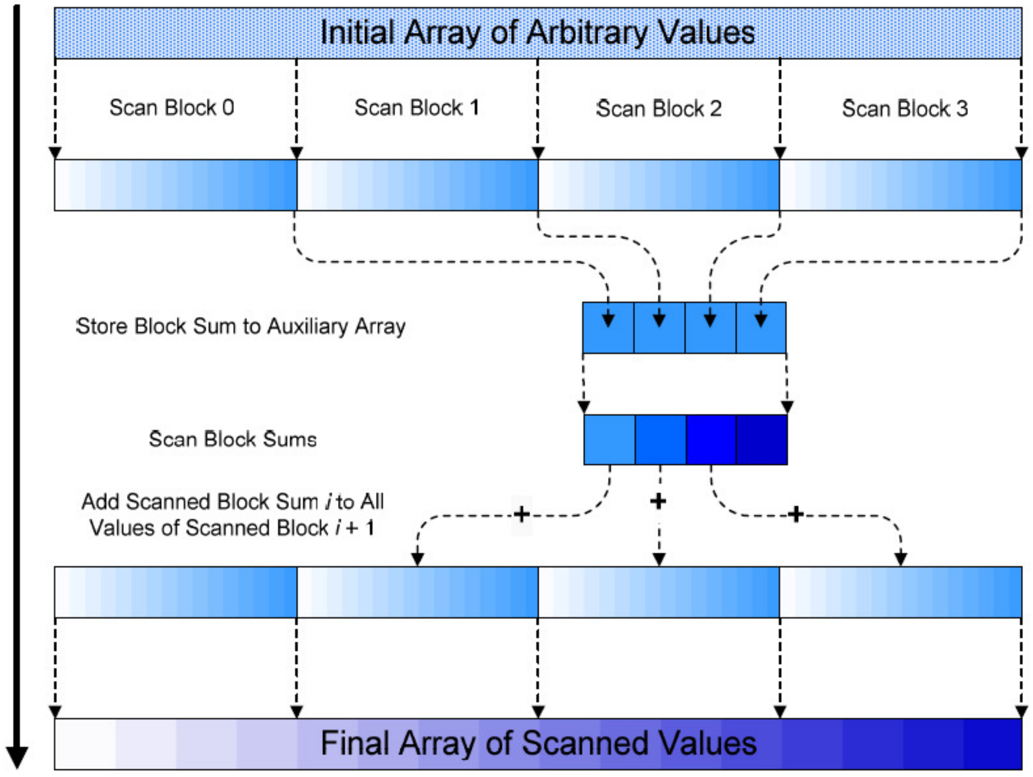
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 0 1 2 …
3 1 7 0 4 1 6 3 5 8 2 0 3 3 1 9 P 4 5 7 …
T0 T1 T2 T3 T4 T5 T6 T7 T8 T9…
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 0 1 2 ...
3 4 7 7 4 5 6 9 5 13 2 2 3 6 1 10 P 4 9 7 …

Note that only arrows with the same color happen simultaneously.

Expanding to arbitrary sized Array

- At this point we have assumed that the array is processed by one block
- Extending to larger arrays requires spliting the array into block sized chunks
- Perform scan independantly on each chunch while storing the total sum in a new array sums.
- Another kernel then updates each block with all the sums to its left.

Expanding to arbitrary sized Array

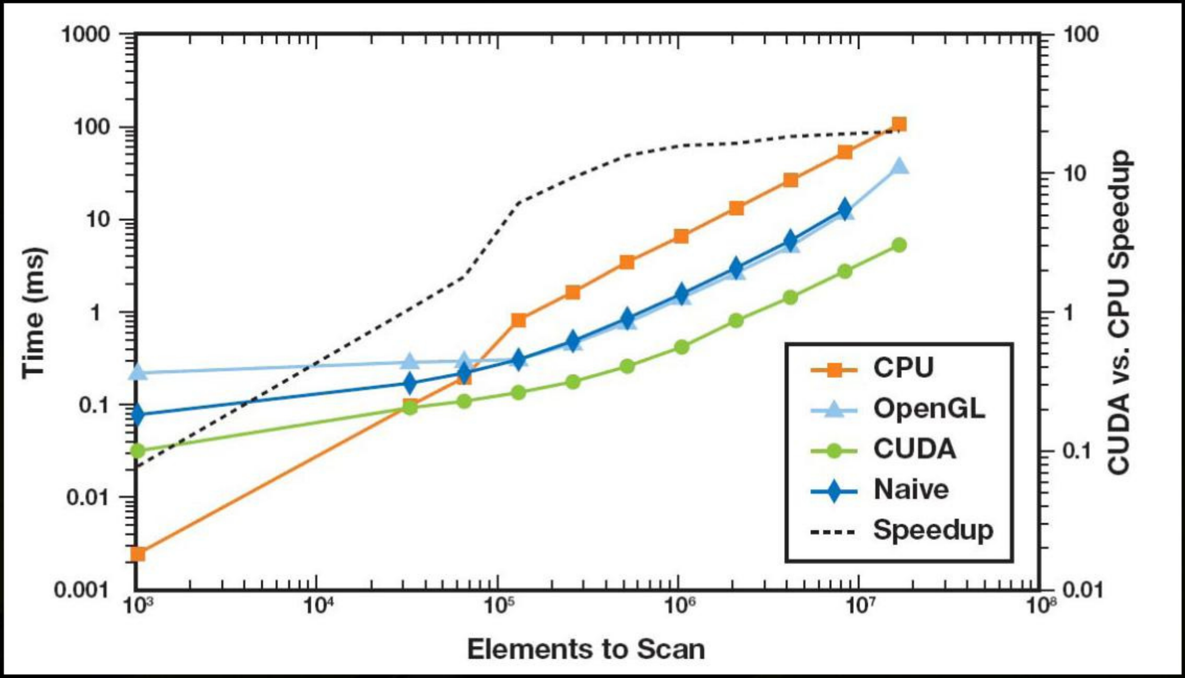


Initial Array of Arbitrary Values
Scan Block 0
Scan Block 1
Scan Block 2
Scan Block 3
Store Block Sum to Auxiliary Array
Scan Block Sums
Add Scanned Block Sum *i* to All Values of Scanned Block *i* + 1
+
+
+
Final Array of Scanned Values

Figure 5: Algorithm for performing a sum scan on a large array of values.

Timing results

CUDA Scan Performance



GeForce 8800 GTX, Intel Core2 Duo Extreme 2.93 GHz

- GPU vs. CPU **20x** CUDA vs. OpenGL **7x**

Scan Primitives

- Enumerate
  - Input: 2 arrays of size N composed of True False Value Pairs
  - Output: an array of size N where each element is the total number of true elements to its left.
  - Easily implemented if we represent true with 1 and false with zero
  - Simply scan the true false array

1	0	1	1	0	1
0	1	1	2	3	3

Scan Primitives

- Array Compaction
  - Input: 2 arrays of size N composed of True False Value Pairs
  - Output: An array that consits of only the values associated with a True key.
  - Simply scan the true false array
  - Everywhere the array is true we now have an index into the output array

A	B	C	D	E	F
1	0	1	1	0	1
0	1	1	2	3	3

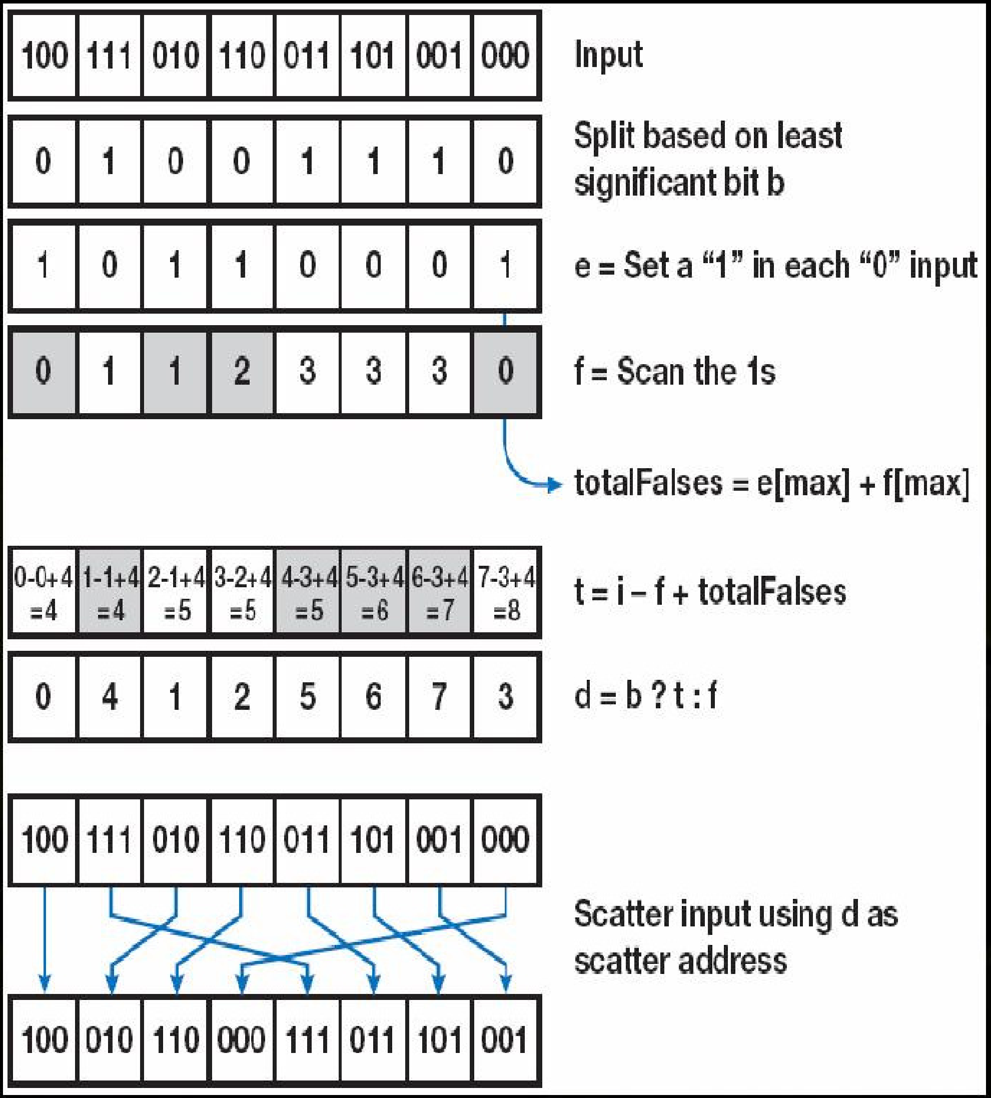
A	C	D	F

Primitives Built Atop Scan

- Split & Split-And-Segment [great for Radix-Sort]
  - Divide input vector into two pieces, with all the elements marked *false (0)* to the left side of the output, all elements marked *true (1)* to the right side.

```
[ 1 0 1 0 0 1 0 ]
[ 0 1 0 1 1 0 1 ]     → Invert them
[ 0 0 1 1 2 3 3 ]     → f = prescan (find new false indices)
        NF = 4        → Total number of falses
[ 0 1 2 3 4 5 6 ]     → Thread ids
[ 4 5 5 6 6 6 7 ]     → t = id – f + NF
```

Application: Radix Sort

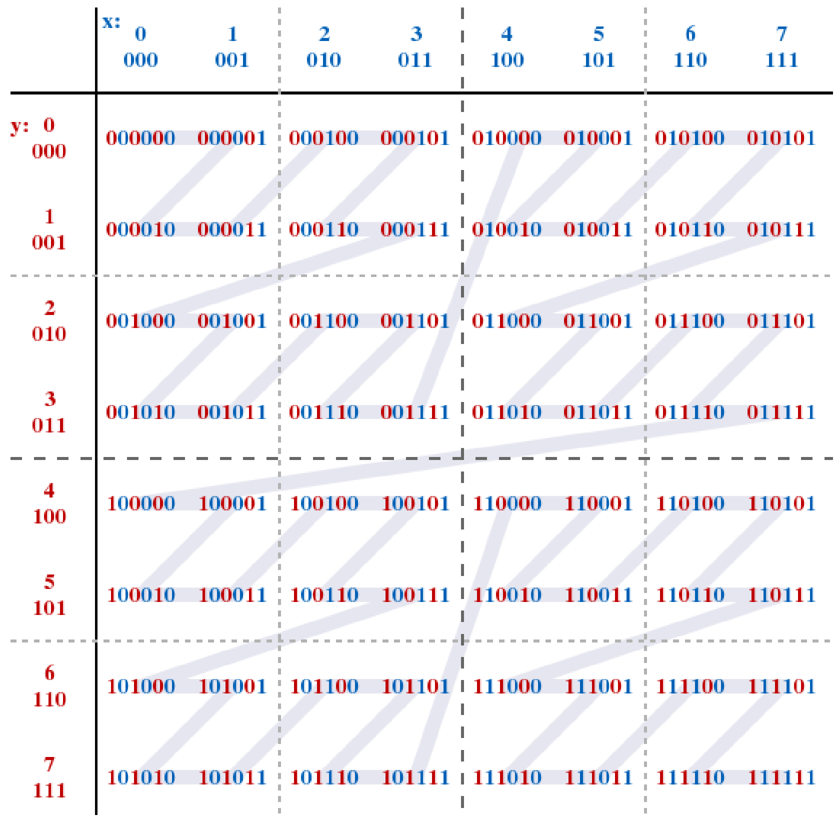


- **Sort 4M key-value pairs (32-bit keys): 80ms**
- **Perform split operation on each bit using scan**

Quad Tree Construction

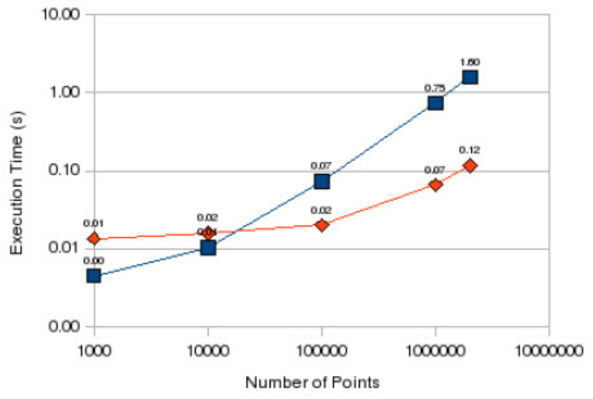
- Using Scan,Radix Sort, and bit interleaving we can create a quad tree data structure
- Algorithm
  - Interleave bits of the coordinate values
  - Sort interleaved array using Radix sort
  - Find locations of boundaries of Quadtree blocks using scan
  - Store array of indexes to boundary locations

Quad Tree Construction

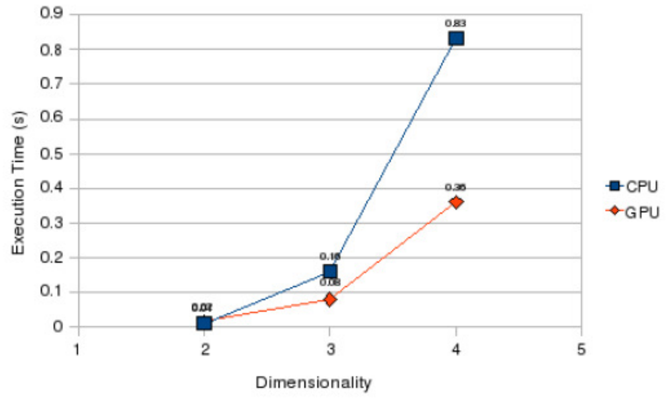


	x: 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
y: 0 000	000000	000001	000100	000101	010000	010001	010100	010101
1 001	000010	000011	000110	000111	010010	010011	010110	010111
2 010	001000	001001	001100	001101	011000	011001	011100	011101
3 011	001010	001011	001110	001111	011010	011011	011110	011111
4 100	100000	100001	100100	100101	110000	110001	110100	110101
5 101	100010	100011	100110	100111	110010	110011	110110	110111
6 110	101000	101001	101100	101101	111000	111001	111100	111101
7 111	101010	101011	101110	101111	111010	111011	111110	111111

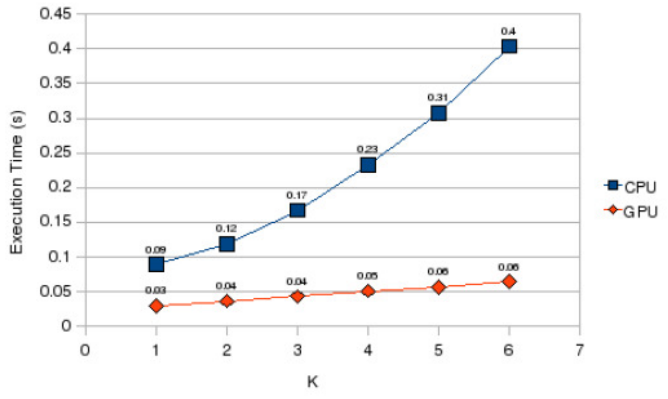
Results



(a)



(b)



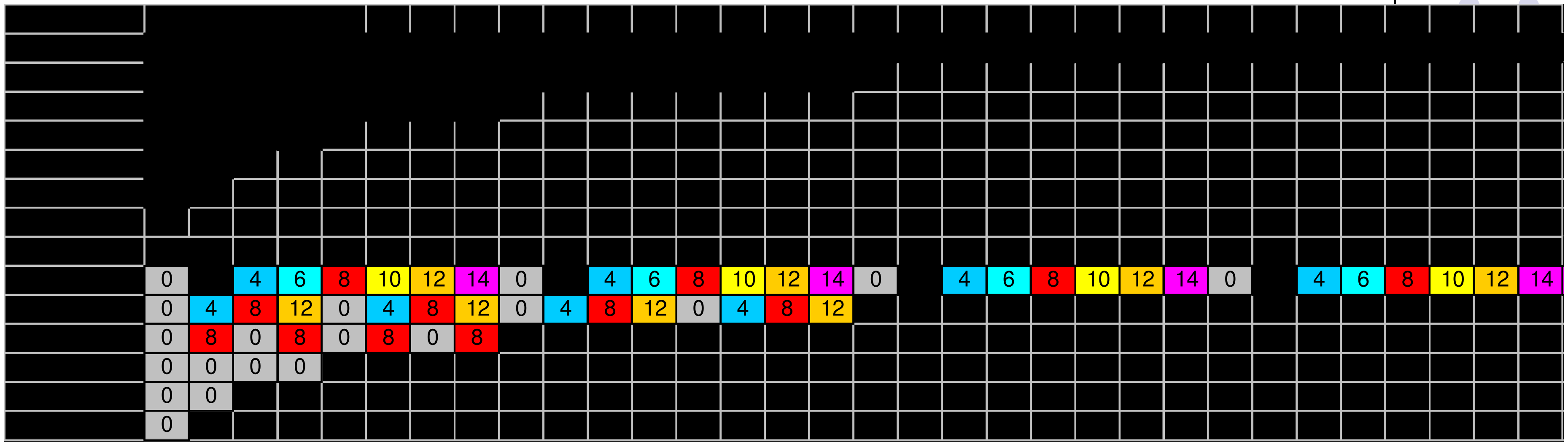
(c)

(c) n = 100k, d = 2, varying k.

Taking it one step further: completely eliminating the bank conflicts (individual reading)

Scan Bank Conflicts (1)

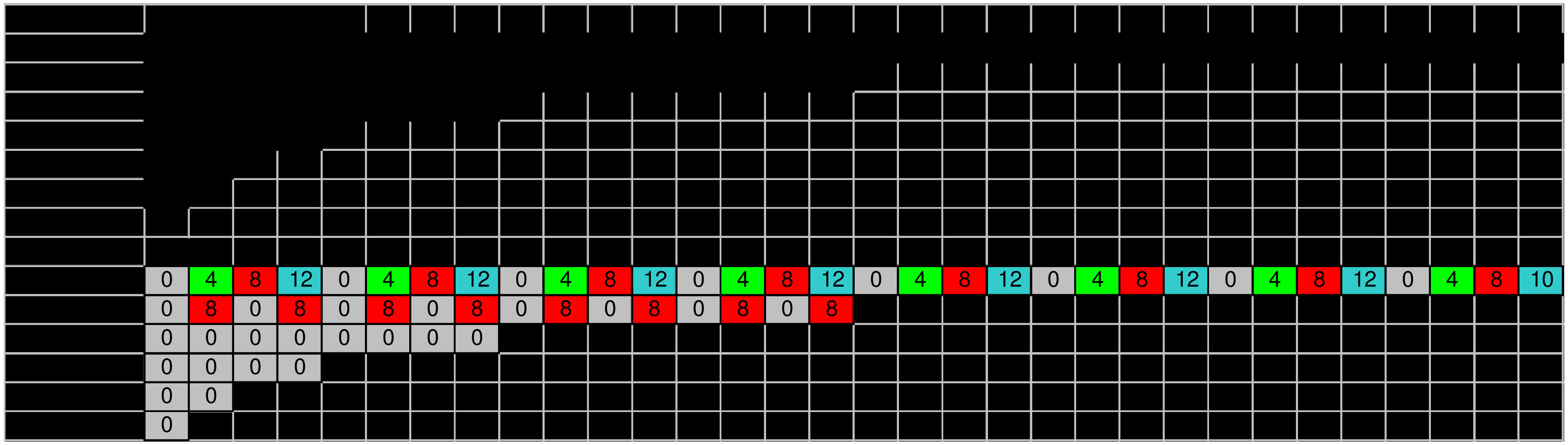
- A full binary tree with 64 leaf nodes:



- Multiple 2-and 4-way bank conflicts
- Shared memory cost for whole tree
  - 1 32-thread warp = 6 cycles per thread w/o conflicts
    - Counting 2 shared mem reads and one write (s[a] += s[b])
  - 6 * (2+4+4+4+2+1) = 102 cycles
  - 36 cycles if there were no bank conflicts (6 * 6)

Scan Bank Conflicts (2)

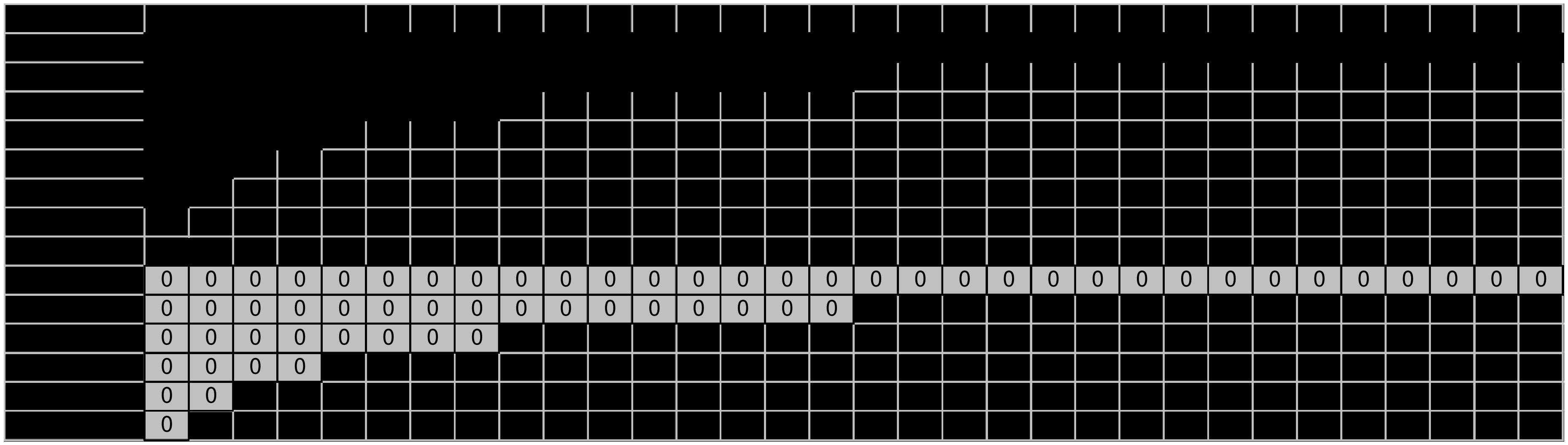
- It's much worse with bigger trees!
- A full binary tree with 128 leaf nodes
  - Only the last 6 iterations shown (root and 5 levels below)



- Cost for whole tree:
  - 12*2 + 6*(4+8+8+4+2+1) = 186 cycles
  - 48 cycles if there were no bank conflicts! 12*1 + (6*6)

Scan Bank Conflicts (3)

- A full binary tree with 512 leaf nodes
  - Only the last 6 iterations shown (root and 5 levels below)



- Cost for whole tree:
  - 48*2+24*4+12*8+6* (16+16+8+4+2+1) = 570 cycles
  - 120 cycles if there were no bank conflicts!

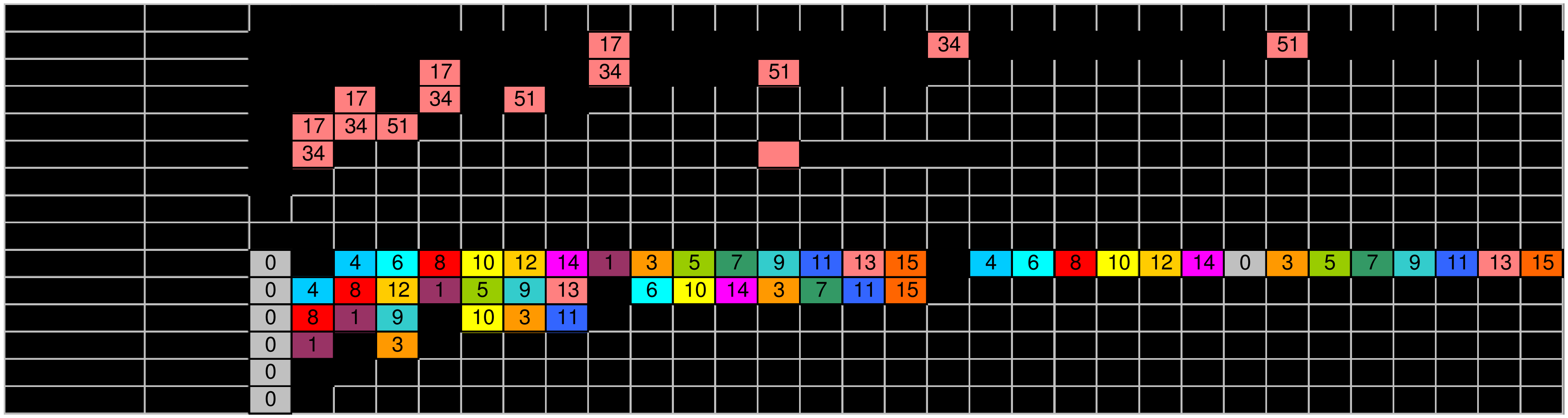
Fixing Scan Bank Conflicts

- Insert padding every NUM_BANKS elements

```
const int LOG_NUM_BANKS = 4; // 16 banks on G80
int tid = threadIdx.x;
int s = 1;
// Traversal from leaves up to root
for (d = n>>1; d > 0; d >>= 1)
{
   if (thid <= d)
   {
         int a = s*(2*tid); int b = s*(2*tid+1)
         a += (a >> LOG_NUM_BANKS); // insert pad word
         b += (b >> LOG_NUM_BANKS); // insert pad word
         shared[a] += shared[b];
   }
}
```

Fixing Scan Bank Conflicts

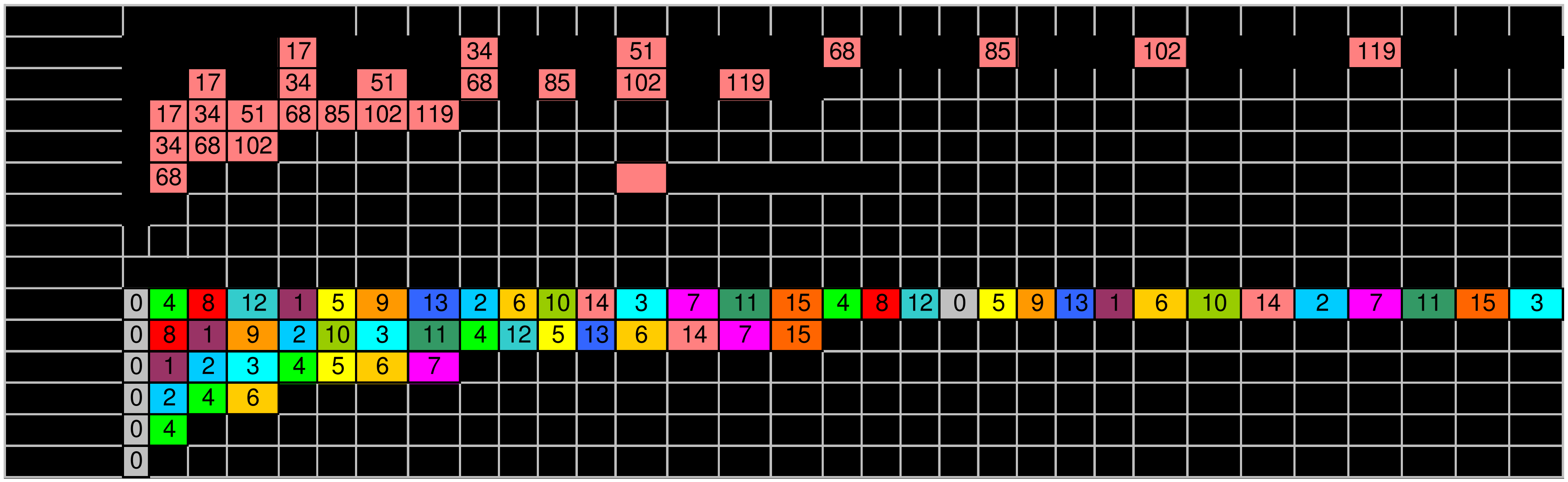
- A full binary tree with 64 leaf nodes



- No more bank conflicts!
  - However, there are ~8 cycles overhead for addressing
    - For each s[a] += s[b] (8 cycles/iter. * 6 iter. = 48 extra cycles)
  - So just barely worth the overhead on a small tree
    - 84 cycles vs. 102 with conflicts vs. 36 optimal

Fixing Scan Bank Conflicts

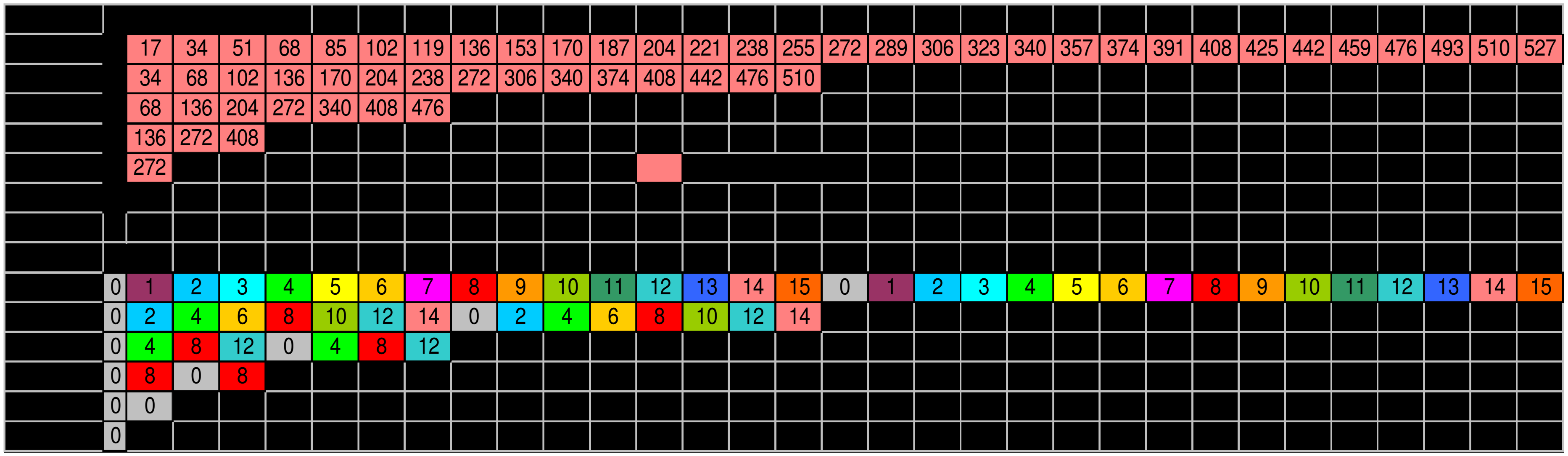
- A full binary tree with 128 leaf nodes
  - Only the last 6 iterations shown (root and 5 levels below)



- No more bank conflicts!
  - Significant performance win:
    - 106 cycles vs. 186 with bank conflicts vs. 48 optimal

Fixing Scan Bank Conflicts

- A full binary tree with 512 leaf nodes
  - Only the last 6 iterations shown (root and 5 levels below)



- Wait, we still have bank conflicts
  - Method is not foolproof, but still much improved
  - 304 cycles vs. 570 with bank conflicts vs. 120 optimal

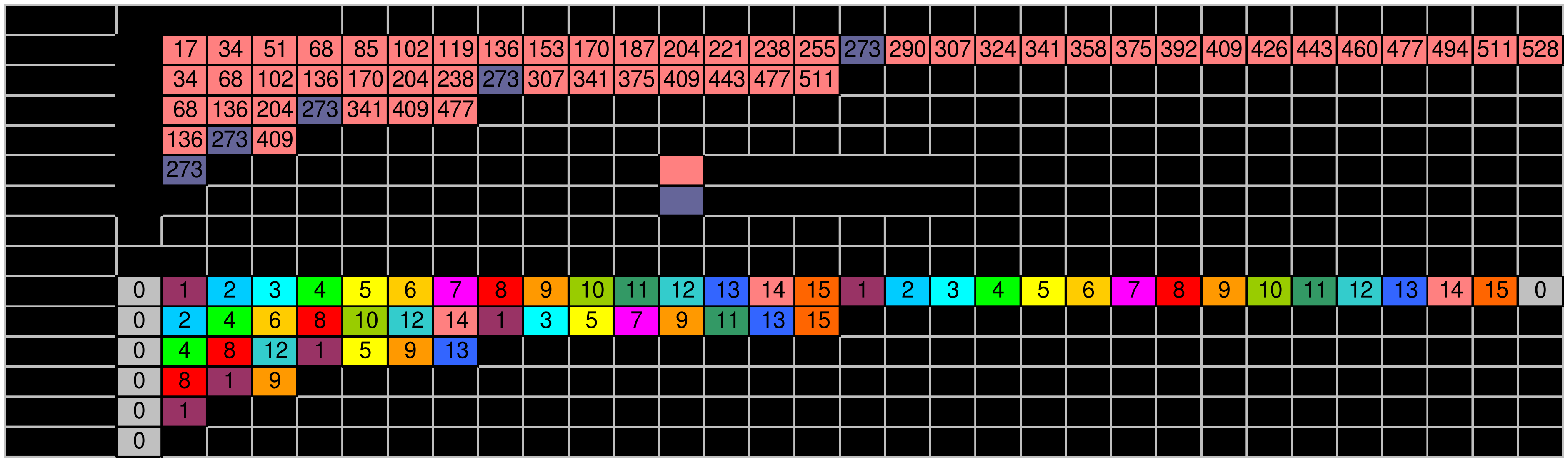
Fixing Scan Bank Conflicts

- It’s possible to remove *all* bank conflicts
  - Just do multi-level padding
  - Example: two-level padding:

```
const int LOG_NUM_BANKS = 4; // 16 banks on G80
int tid = threadIdx.x;
int s = 1;
// Traversal from leaves up to root
for (d = n>>1; d > 0; d >>= 1)
{
    if (thid <= d)
    {
      int a = s*(2*tid); int b = s*(2*tid+1)
      int offset = (a >> LOG_NUM_BANKS); // first level
      a += offset + (offset >>LOG_NUM_BANKS); // second level
      offset = (b >> LOG_NUM_BANKS);     // first level
      b += offset + (offset >>LOG_NUM_BANKS); // second level
        temp[a] += temp[b];
    }
}
```

Fixing Scan Bank Conflicts

- A full binary tree with 512 leaf nodes
  - Only the last 6 iterations shown (root and 5 levels below)



- No bank conflicts
  - But an extra cycle overhead per address calculation
  - Not worth it: 440 cycles vs. 304 with 1-level padding
- With 1-level padding, bank conflicts only occur in warp 0
  - Very small remaining cost due to bank conflicts
  - Removing them hurts all other warps