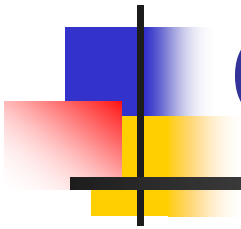
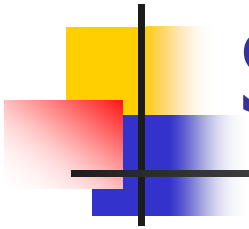


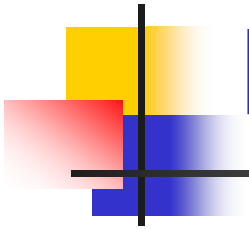
CUDA Tricks

Presented by
Damodaran Ramani



Synopsis

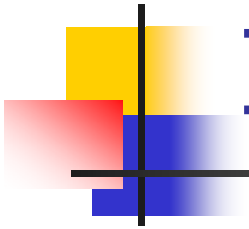
- Scan Algorithm
  - Applications
- Specialized Libraries
  - CUDPP: CUDA Data Parallel Primitives Library
  - Thrust: a Template Library for CUDA Applications
  - CUDA FFT and BLAS libraries for the GPU



References

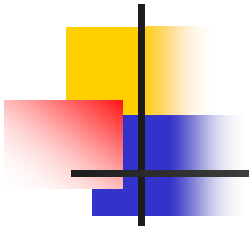
- Scan primitives for GPU Computing.
  - Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens

- Presentation on scan primitives by Gary J. Katz based on the article Parallel Prefix Sum (Scan) with CUDA - Harris, Sengupta and Owens (GPU GEMS Chapter 39)



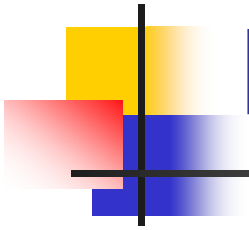
Introduction

- GPUs massively parallel processors
- Programmable parts of the graphics pipeline operates on primitives (vertices, fragments)
- These primitive programs spawn a thread for each primitive to keep the parallel processors full

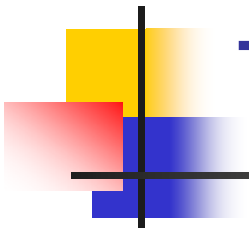


- Stream programming model (particle systems, image processing, grid-based fluid simulations, and dense matrix algebra)
- Fragment program operating on n fragments (accesses - O(n))
- Problem arises when access requirements are complex (eg: prefix-sum – $O(n^2)$)

Prefix-Sum Example

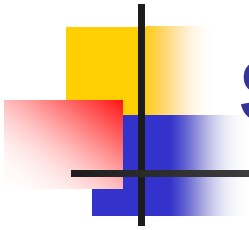


- in: 3 1 7 0 4 1 6 3
- out: 0 3 4 11 11 14 16 22



Trivial Sequential Implementation

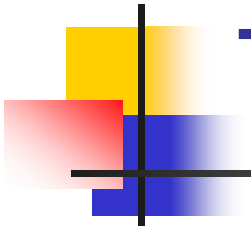
```
void scan(int* in, int* out, int n)
{
        out[0] = 0;
        for (int i = 1; i < n; i++)
                out[i] = in[i-1] + out[i-1];
}
```



Scan: An Efficient Parallel Primitive

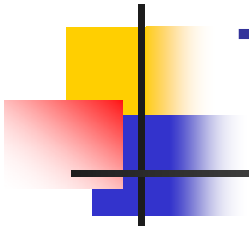
- Interested in finding efficient solutions to parallel problems in which *each output requires global knowledge of the inputs*.

- Why CUDA? (General Load-Store Memory Architecture, On-chip Shared Memory, Thread Synchronization**)**



Threads & Blocks

- GeForce 8800 GTX (16 multiprocessors, 8 processors each)
- CUDA structures GPU programs into parallel *thread blocks of up to* 512 SIMD-parallel threads.
- Programmers specify the number of thread blocks and threads per block, and the hardware and drivers map thread blocks to parallel multiprocessors on the GPU.
- Within a thread block, threads can communicate through shared memory and cooperate through sync.
- Because only threads within the same block can cooperate via shared memory and thread synchronization,programmers must partition computation into multiple blocks.(complex programming, large performance benefits)



The Scan Operator

- Definition:
  - The scan 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 array

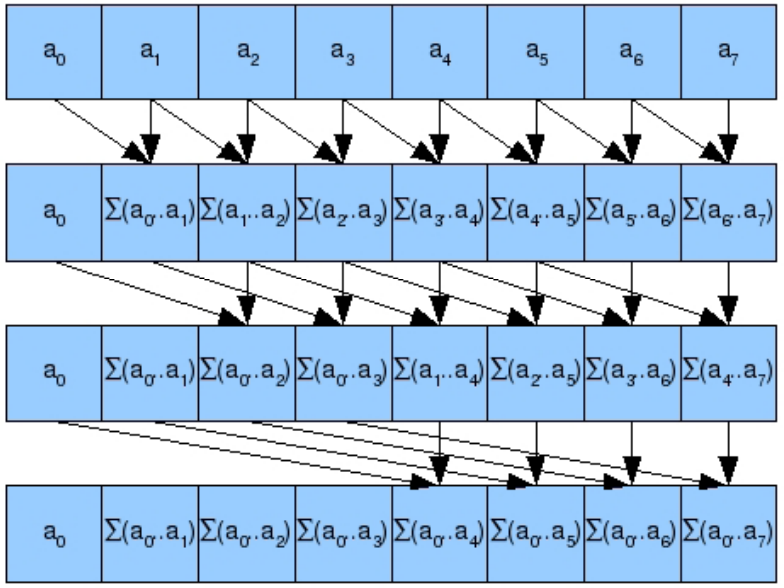
    $[I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})]$

    Types – *inclusive, exclusive, forward, backward*

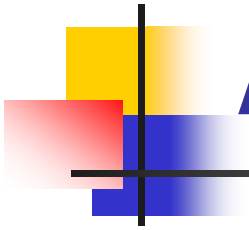
Parallel Scan

```
for(d = 1; d < log2n; d++)
   for all k in parallel
     if( k >= 2^d )
        x[out][k] = x[in][k – 2^(d-1)] + x[in][k]
     else
        x[out][k] = x[in][k]
```

Complexity $O(n \log_2 n)$



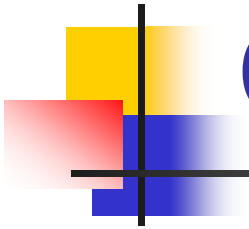
A work efficient parallel scan



- Goal is a parallel scan that is O(n) instead of O($n \log_2 n$)
- Solution:
  - Balanced Trees: Build a binary tree on the input data and sweep it to and from the root.

  Binary tree with n leaves has $d = \log_2 n$ levels, each level d has 2^d nodes

  One add is performed per node, therefore O(n) add on a single traversal of the tree.



O(n) unsegmented scan

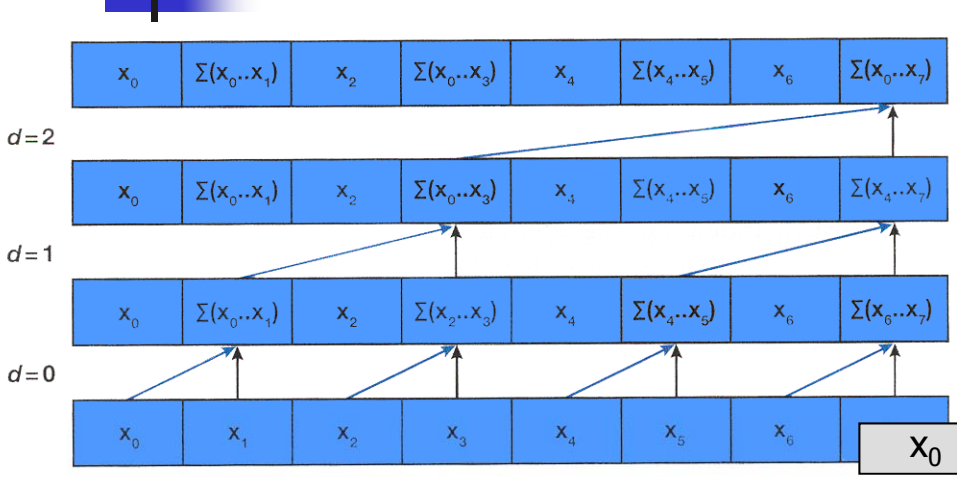
- Reduce/Up-Sweep

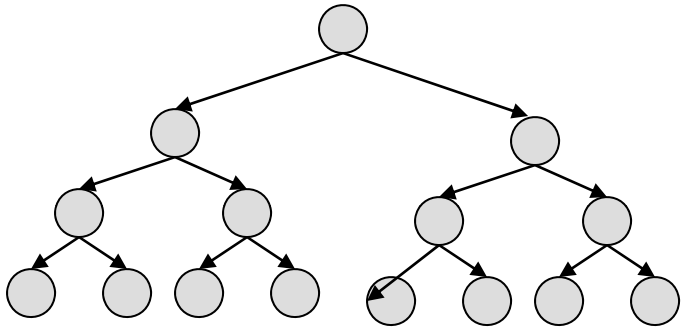
```
for(d = 0; d < log2n-1; d++)
   for all k=0; k < n-1; k+=2^(d+1) in parallel
     x[k+2^(d+1)-1] = x[k+2^d-1] + x[k+2^(d+1)-1]
```

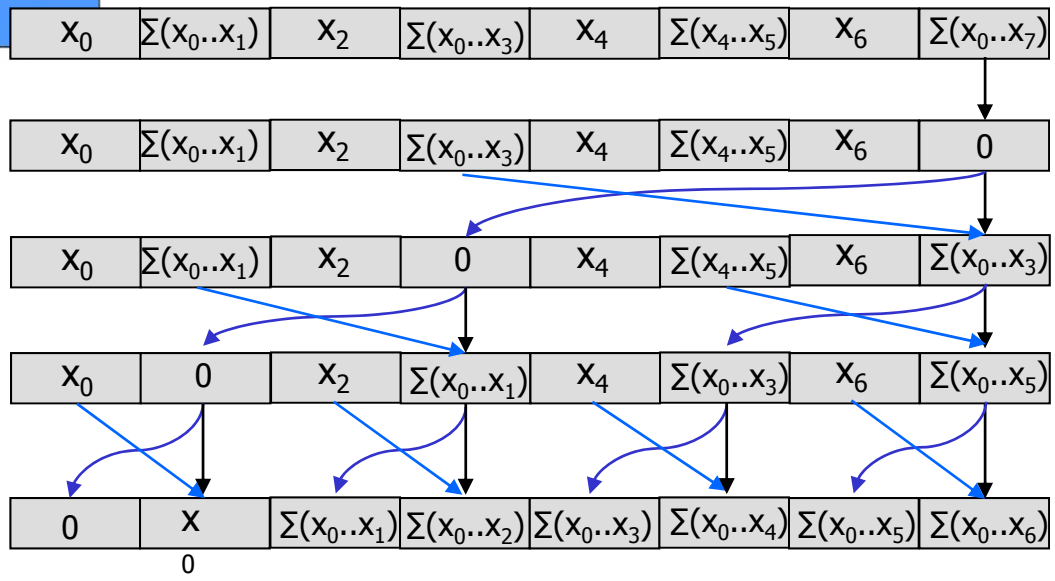
- Down-Sweep

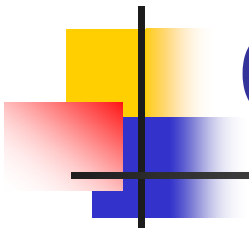
```
x[n-1] = 0;
for(d = log2n – 1; d >=0; d--)
   for all k = 0; k < n-1; k += 2^(d+1) in parallel
      t = x[k + 2^d – 1]
      x[k + 2^d - 1] = x[k + 2^(d+1) -1]
      x[k + 2^(d+1) - 1] = t + x[k + 2^(d+1) – 1]
```

Tree analogy









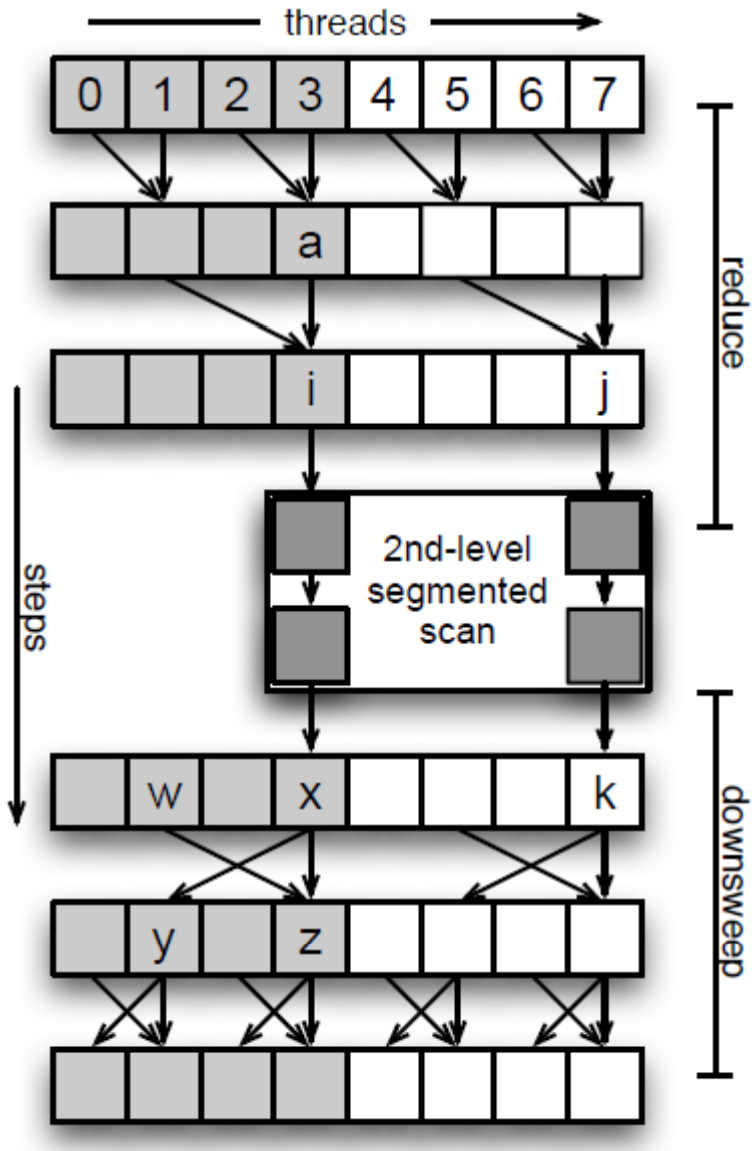
O(n) Segmented Scan

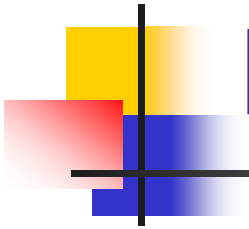
Up-Sweep

1: **for** $d = 1$ to $\log_2 n - 1$ **do**
2: **for all** $k = 0$ to $n - 1$ by 2^{d+1} in parallel **do**
3: **if** $f[k + 2^{d+1} - 1]$ is not set **then**
4: $x[k + 2^{d+1} - 1] \leftarrow x[k + 2^d - 1] + x[k + 2^{d+1} - 1]$
5: $f[k + 2^{d+1} - 1] \leftarrow f[k + 2^d - 1] \mid f[k + 2^{d+1} - 1]$

- Down-Sweep

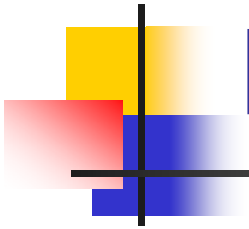
```
1:  x[n − 1] ← 0
2:  for d = log2 n − 1 down to 0 do
3:      for all k = 0 to n − 1 by 2^(d+1) in parallel do
4:          t ← x[k + 2^d − 1]
5:          x[k + 2^d − 1] ← x[k + 2^(d+1) − 1]
6:          if f_i[k + 2^d] is set then
7:              x[k + 2^(d+1) − 1] ← 0
8:          else if f[k + 2^d − 1] is set then
9:              x[k + 2^(d+1) − 1] ← t
10:         else
11:             x[k + 2^(d+1) − 1] ← t + x[k + 2^(d+1) − 1]
12:         Unset flag f[k + 2^d − 1]
```





Features of segmented scan

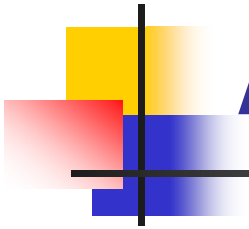
- 3 times slower than unsegmented scan
- Useful for building broad variety of applications which are not possible with unsegmented scan.



Primitives built on scan

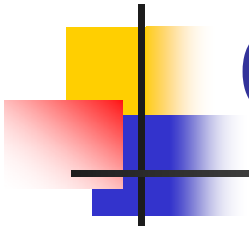
- Enumerate
  - enumerate([t f f t f t t]) = [0 1 1 1 2 2 3]
  - Exclusive scan of input vector
- Distribute (copy)
  - distribute([a b c][d e]) = [a a a][d d]
  - Inclusive scan of input vector
- Split and split-and-segment

Split divides the input vector into two pieces, with all the elements marked false on the left side of the output vector and all the elements marked true on the right.



Applications

- Quicksort
- Sparse Matrix-Vector Multiply
- Tridiagonal Matrix Solvers and Fluid Simulation
- Radix Sort
- Stream Compaction
- Summed-Area Tables



Quicksort

```
[5 3 7 4 6]    # initial input
[5 5 5 5 5]    # distribute pivot across segment
[f f t f t]    # input > pivot?
[5 3 4][7 6]   # split-and-segment
[5 5 5][7 7]   # distribute pivot across segment
[t f f][t f]   # input >= pivot?
[3 4 5][6 7]   # split-and-segment, done!
```

Sparse Matrix-Vector Multiplication

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix} += \begin{pmatrix} a & 0 & b \\ c & d & e \\ 0 & 0 & f \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix}$$

$$\begin{aligned} \text{value} &= [a,b,c,d,e,f] \\ \text{index} &= [0,2,0,1,2,2] \\ \text{rowPtr} &= [0,2,5] \end{aligned}$$

$$\begin{aligned} \text{product} &= [x_0a, x_2b, x_0c, x_1d, x_2e, x_2f] && (1) \\ &= [[x_0a, x_2b][x_0c, x_1d, x_2e][x_2f]] && (2) \\ &= [[x_0a + x_2b, x_2b] \\ &\quad [x_0c + x_1d + x_2e, x_1d + x_2e, x_2e][x_2f]] && (3) \\ y &= y + [[x_0a + x_2b, x_0c + x_1d + x_2e, x_2f] && (4) \end{aligned}$$

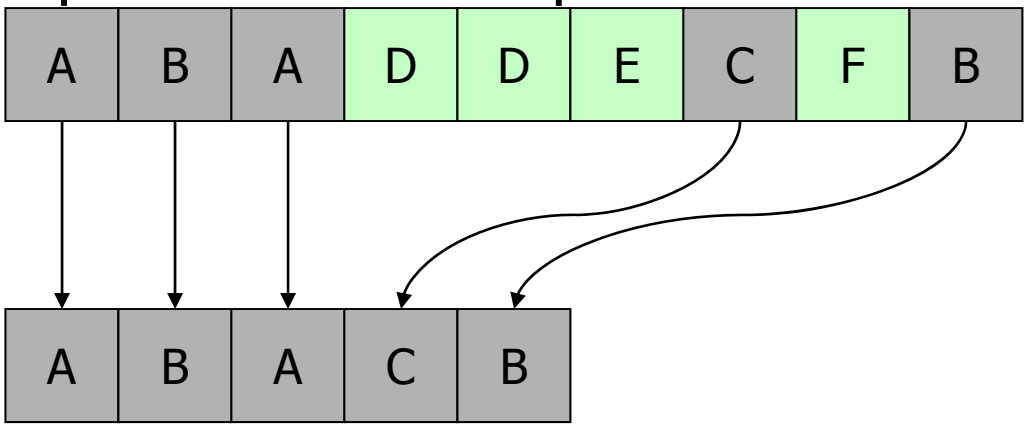
1. The first kernel runs over all entries. For each entry, it sets the corresponding `flag` to 0 and performs a multiplication on each entry: `product = x[index] * value`.
2. The next kernel runs over all rows and sets the head flag to 1 for each `rowPtr` in `flag` through a scatter. This creates one segment per row.
3. We then perform a backward segmented inclusive sum scan on the e elements in `product` with head flags in `flag`.
4. To finish, we run our final kernel over all rows, adding the value in `y` to the gathered value from `products[idx]`.

Stream Compaction

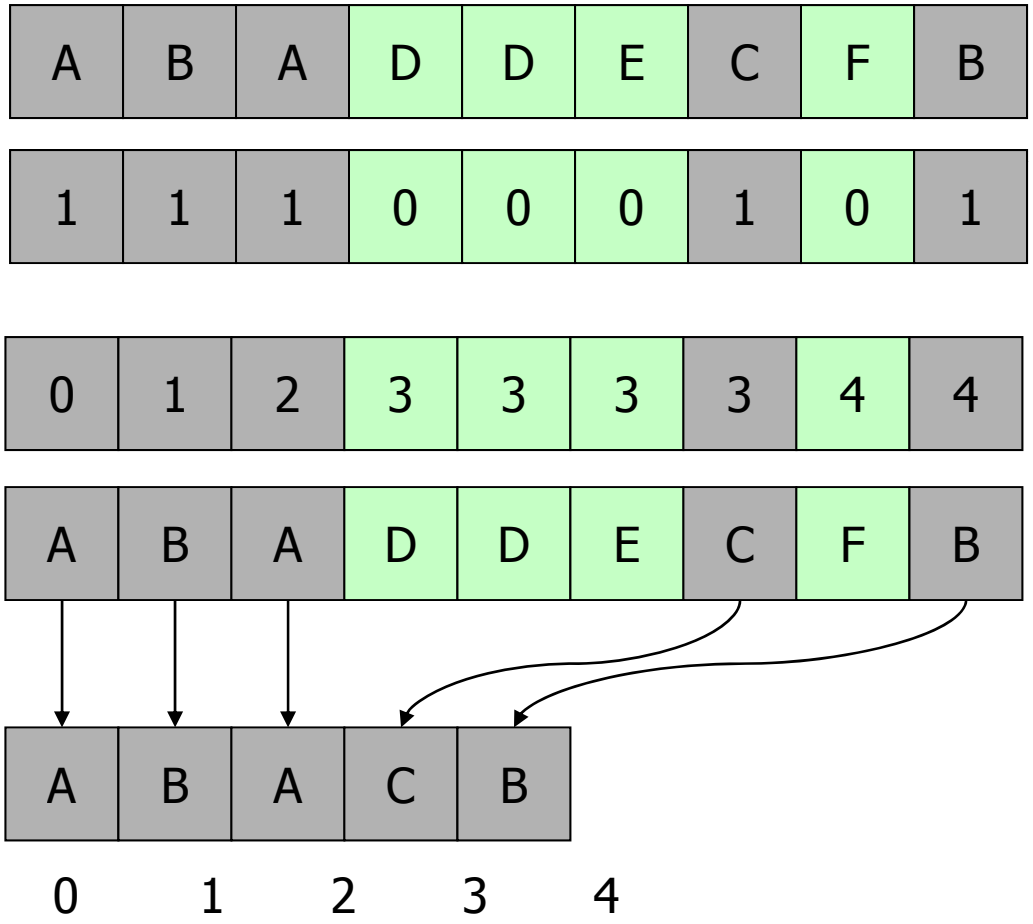
Definition:

- Extracts the 'interest' elements from an array of elements and places them continuously in a new array

- Uses:
  - Collision Detection
  - Sparse Matrix Compression



Stream Compaction



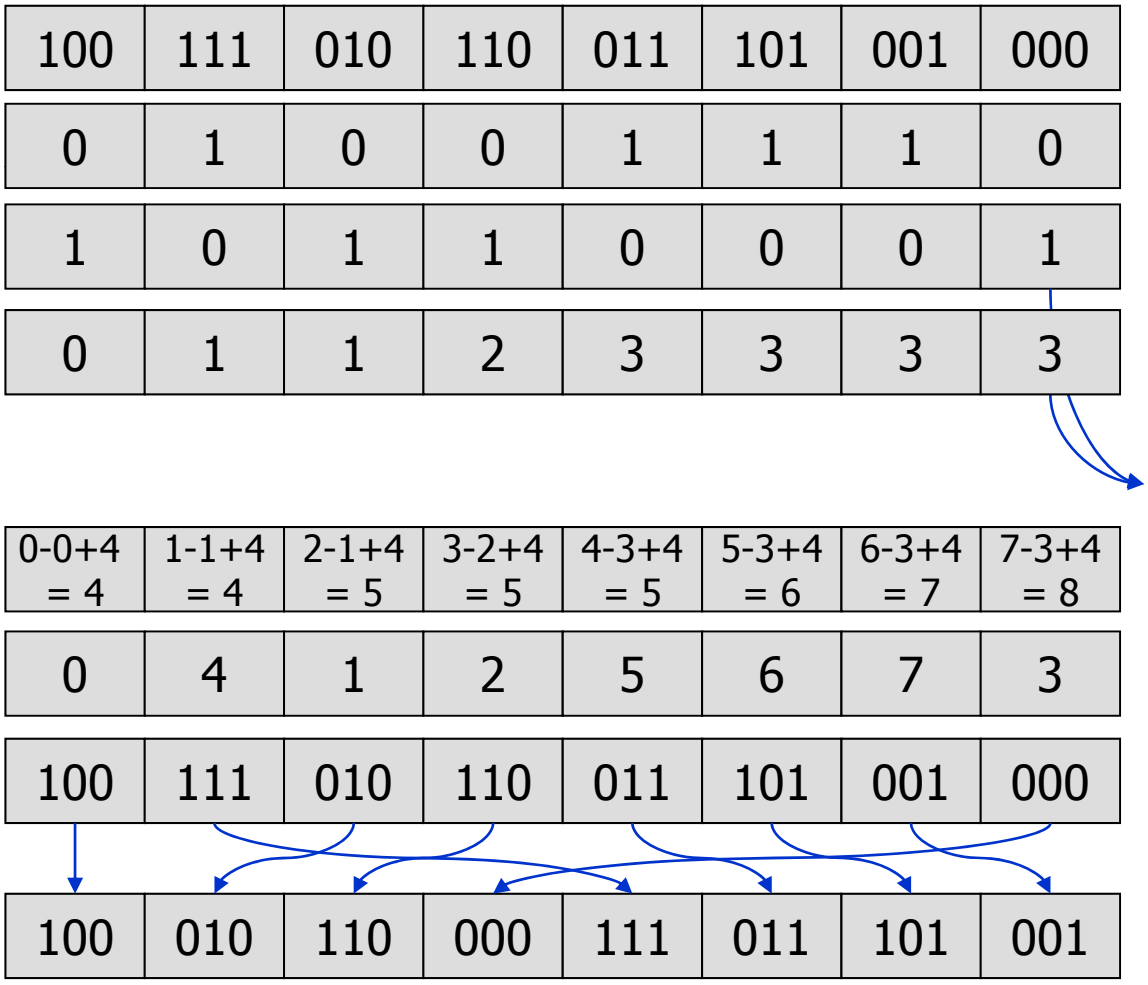
Input: We want to preserve the gray elements

Set a '1' in each gray input

Scan

Scatter gray inputs to output using scan result as scatter address

Radix Sort Using Scan



Input Array

b = least significant bit
e = Insert a 1 for all false sort keys

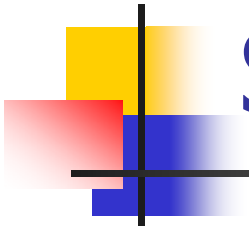
f = Scan the 1s

Total Falses = e[n-1] + f[n-1]

t = index – f + Total Falses

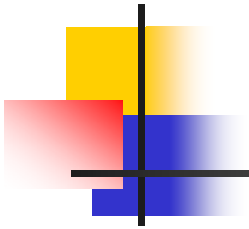
d = b ? t : f

Scatter input using d as scatter address



Specialized Libraries

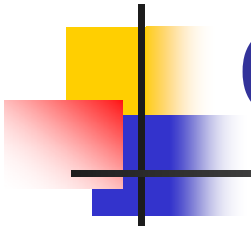
- CUDPP: CUDA Data Parallel Primitives Library
  - CUDPP is a library of data-parallel algorithm primitives such as parallel prefix-sum (”scan”), parallel sort and parallel reduction.



CUDPP_DLL CUDPPResult cudppSparseMatrixVectorMultiply(CUDPPHandle *sparseMatrixHandle*,void * *d_y*,const void * *d_x*)

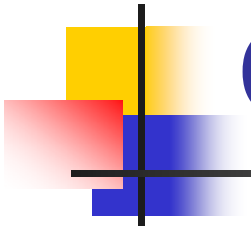
Perform matrix-vector multiply y = A*x for arbitrary sparse matrix A and vector x.

```
CUDPPScanConfig config;
    config.direction = CUDPP_SCAN_FORWARD;
    config.exclusivity = CUDPP_SCAN_EXCLUSIVE;
    config.op = CUDPP_ADD;
    config.datatype = CUDPP_FLOAT;
    config.maxNumElements = numElements;
    config.maxNumRows = 1;
    config.rowPitch = 0;
cudppInitializeScan(&config);
cudppScan(d_odata, d_idata, numElements, &config);
```



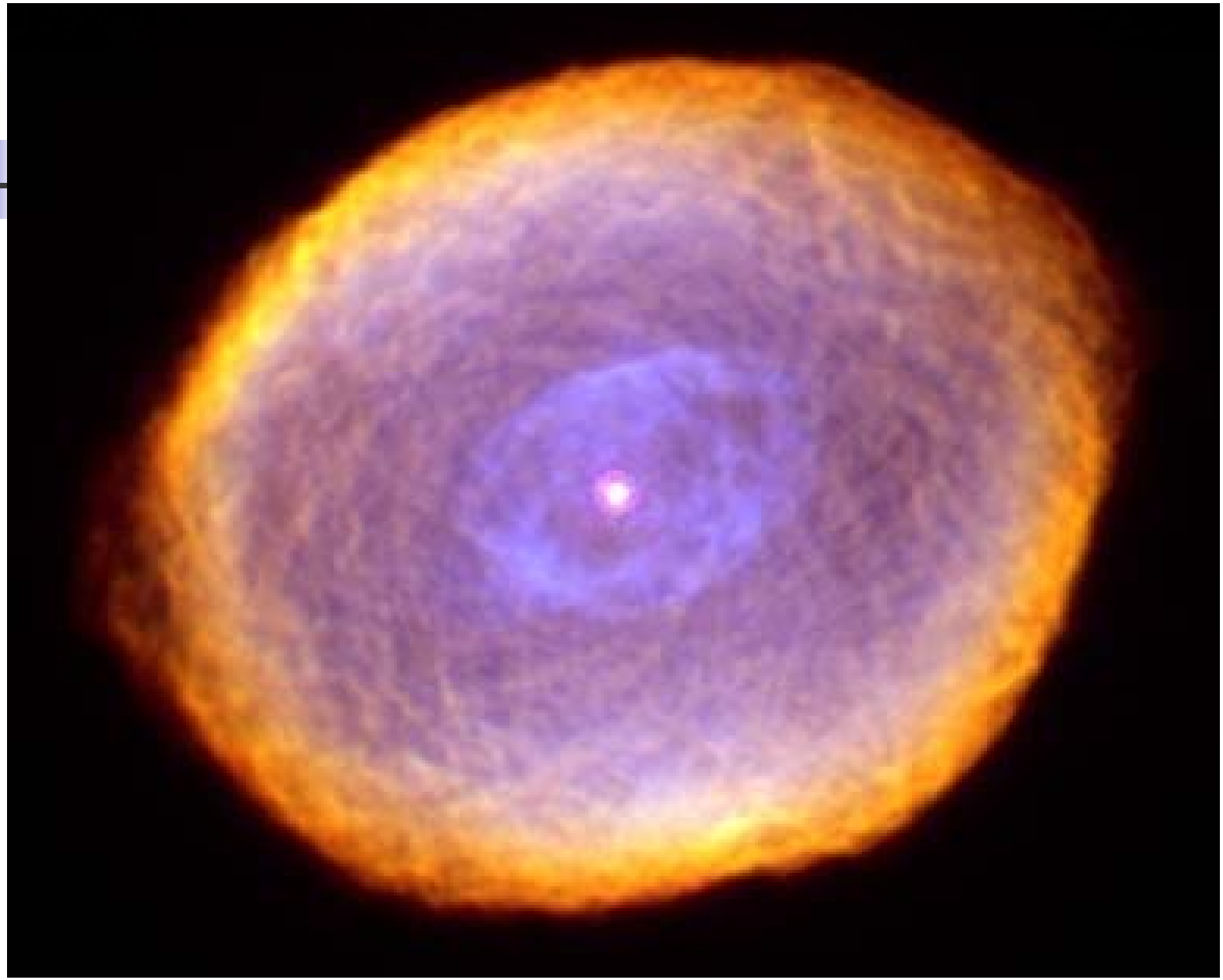
CUFFT

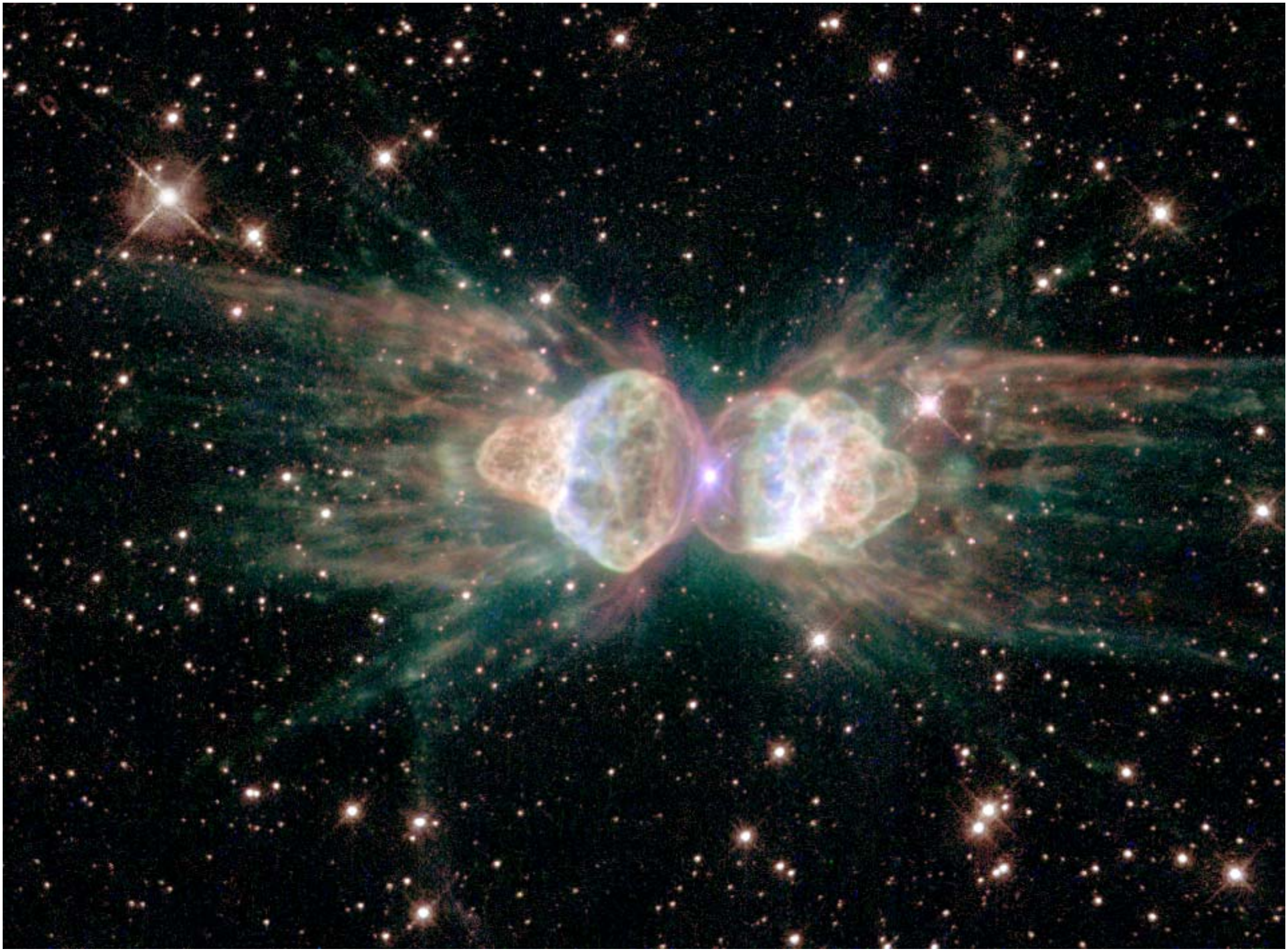
- No. of elements<8192 slower than fftw
- >8192, 5x speedup over threaded fftw and 10x over serial fftw.

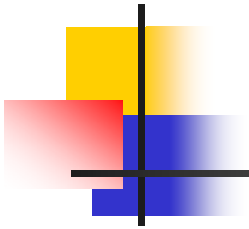


CUBLAS

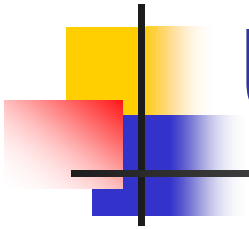
- Cuda Based Linear Algebra Subroutines
- Saxpy, conjugate gradient, linear solvers.
- 3D reconstruction of planetary nebulae.
  - http://graphics.tu-bs.de/publications/Fernandez08TechReport.pdf







- GPU Variant 100 times faster than CPU version
- Matrix size is limited by graphics card memory and texture size.
- Although taking advantage of sparce matrices will help reduce memory consumption, sparse matrix storage is not implemented by CUBLAS.



Useful Links

- http://www.science.uwaterloo.ca/~hmerz/CUDA_benchFFT/
- http://developer.download.nvidia.com/compute/cuda/2_0/docs/CUBLAS_Library_2.0.pdf
- http://gpgpu.org/developer/cudpp
- http://gpgpu.org/2009/05/31/thrust