Block SpMV on GPU

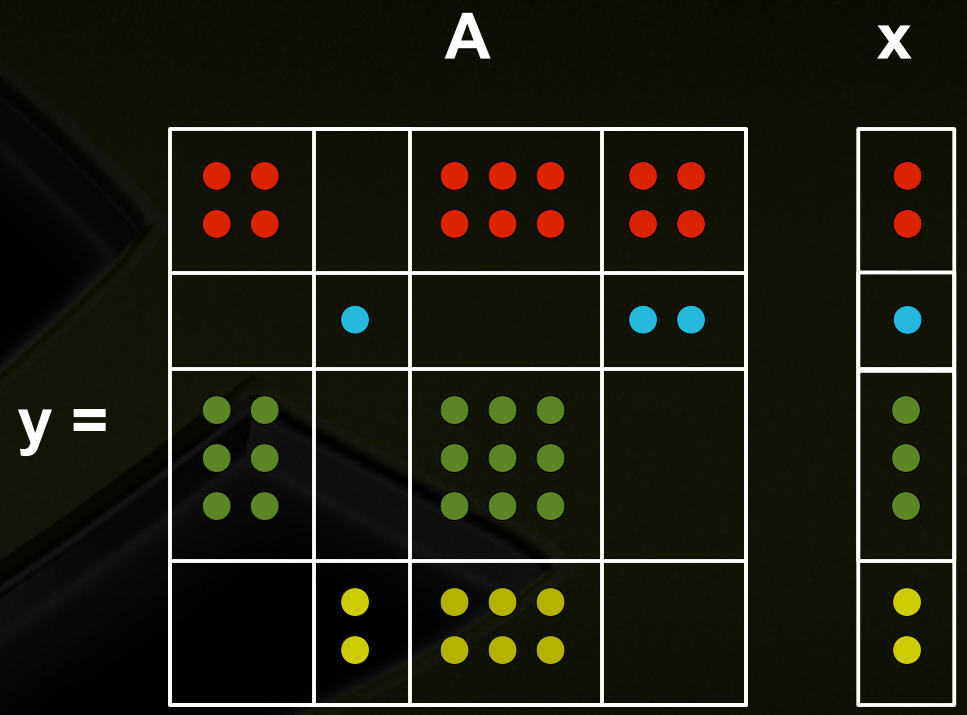
Steve Rennich
NVIDIA HPC DevTech

Block SpMV

- Many matrices arising from engineering analysis have a 'natural' block structure
- Sparse Matrix-Vector Multiplication (SpMV) is a commonly used operation – iterative methods
- Optimize Block SpMV algorithm for the GPU
- Approach / algorithm might both be useful

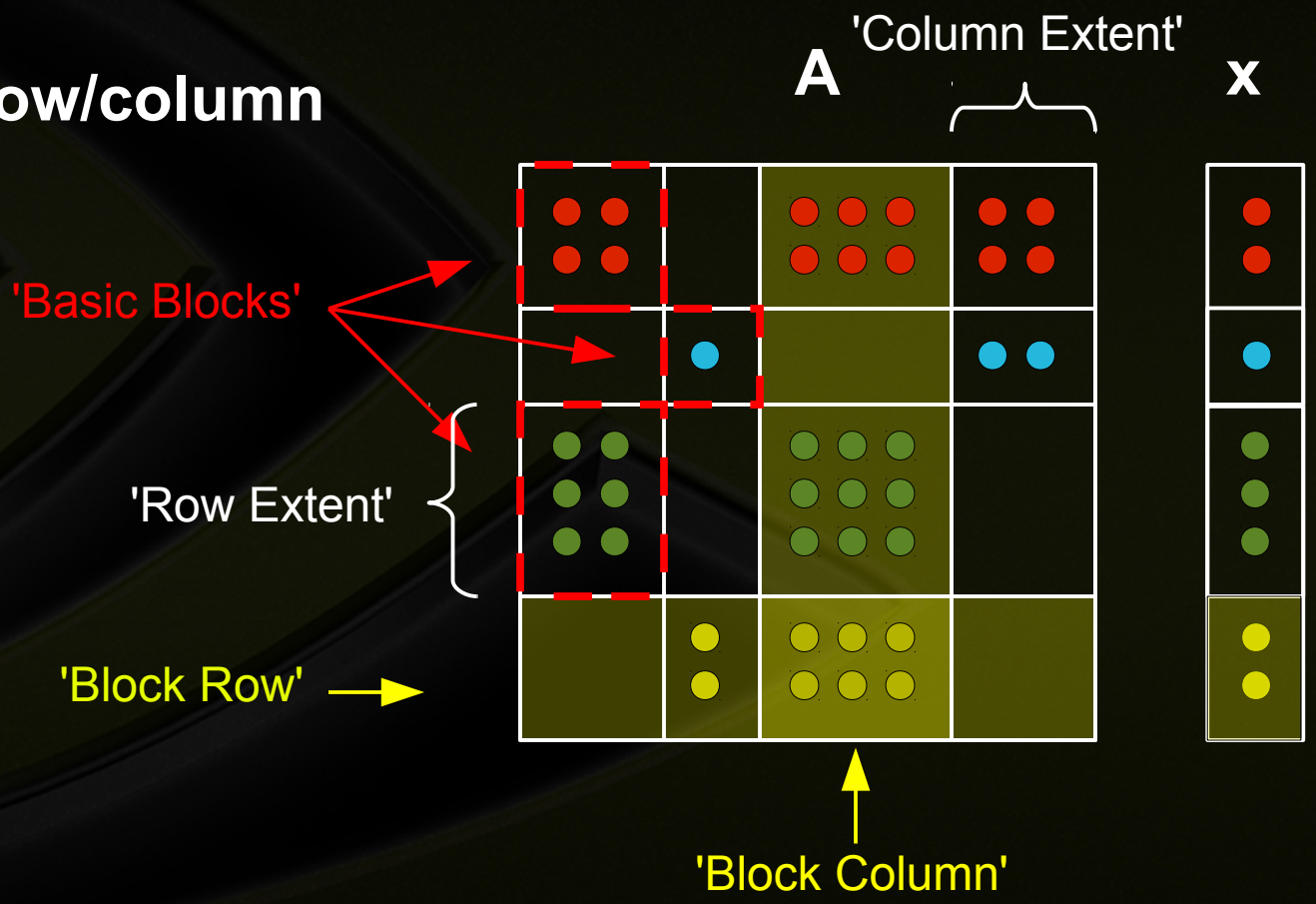
Blocked SpMV

- Compute y = Ax
- A has non-uniform block structure
- x is dense
- Leverage block structure for improved performance



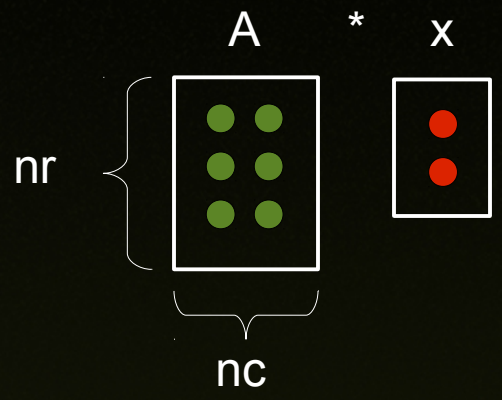
Matrix Structure

- 'Naturally' Blocked
- Variable row/column extent



Bandwidth Analysis

- Double Precision
- Memory Bound
- C2070 – ECC off – 144 GB/s



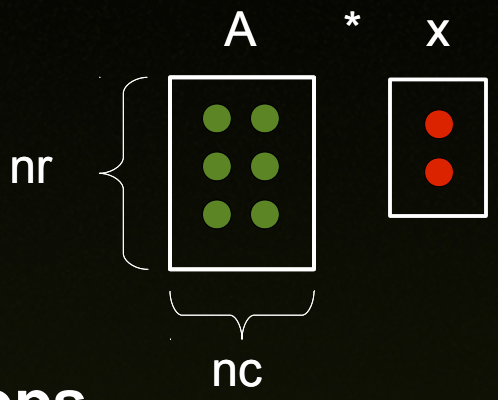
- Standard approach:

8 + 4 + 8 = 20 bytes → **14.4** Gflops

A column index x

(using unsigned ints for column index supports N <= 4.2B)

Bandwidth Analysis



- Double Precision
- Memory Bound
- C2070 – ECC off – 144 GB/s
- Standard approach – **14.4** Gflops
- Upper Bound – **36** Gflops (vs. **6.4** Gflops for socket: **x5670**)

- Block-based:

$$8 + 4 / (nr\ nc) + 8 / nr =$$

A — column index — x

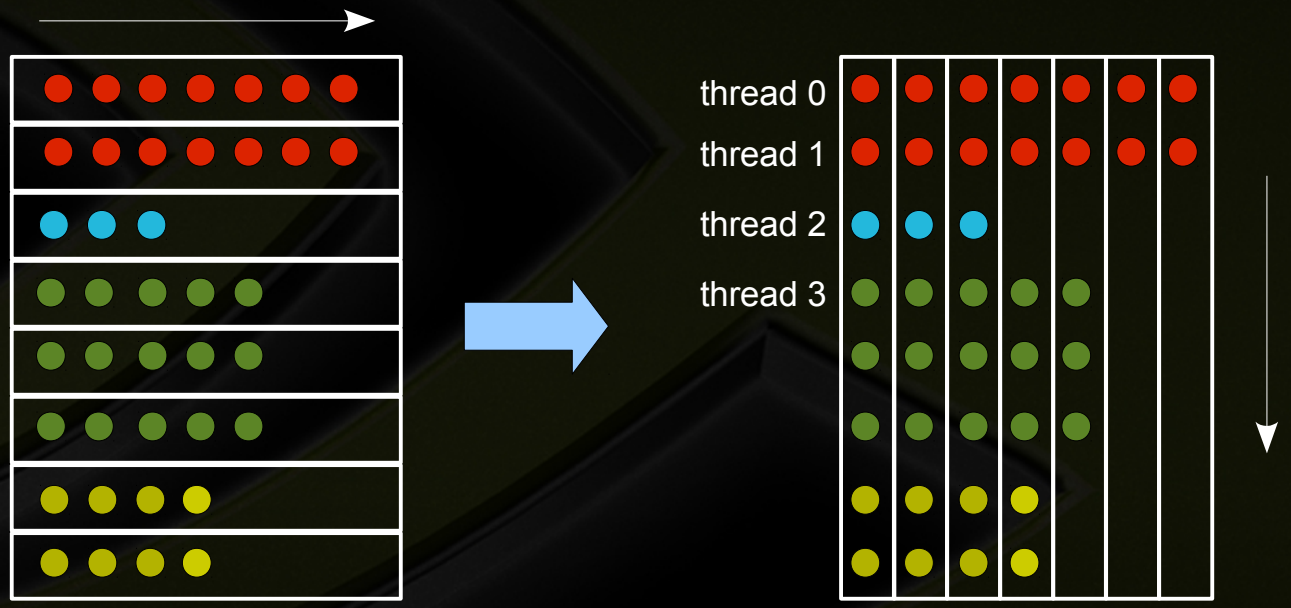
nr	nc	bytes	Gflops
2	2	13	**22.2**
3	3	11.1	**25.9**
6	6	9.4	**30.5**

Requirements for maximum performance

- Sufficient parallelism
  - 1 thread per row
- Coherent memory access
  - ELLPACK (or similar) data structure
- Coherent execution
  - Reorder rows – Load balancing
  - Separate kernels for each column extent – Warp divergence
- Limited data transfer
  - Block structure minimizes column index data
  - Row and column extent are implicit
- Cache as much as possible
  - Optimal use of texture cache for x data

Coherent Memory Access

- Data Structure for A matrix values
  - Convert CSR → ELLPACK



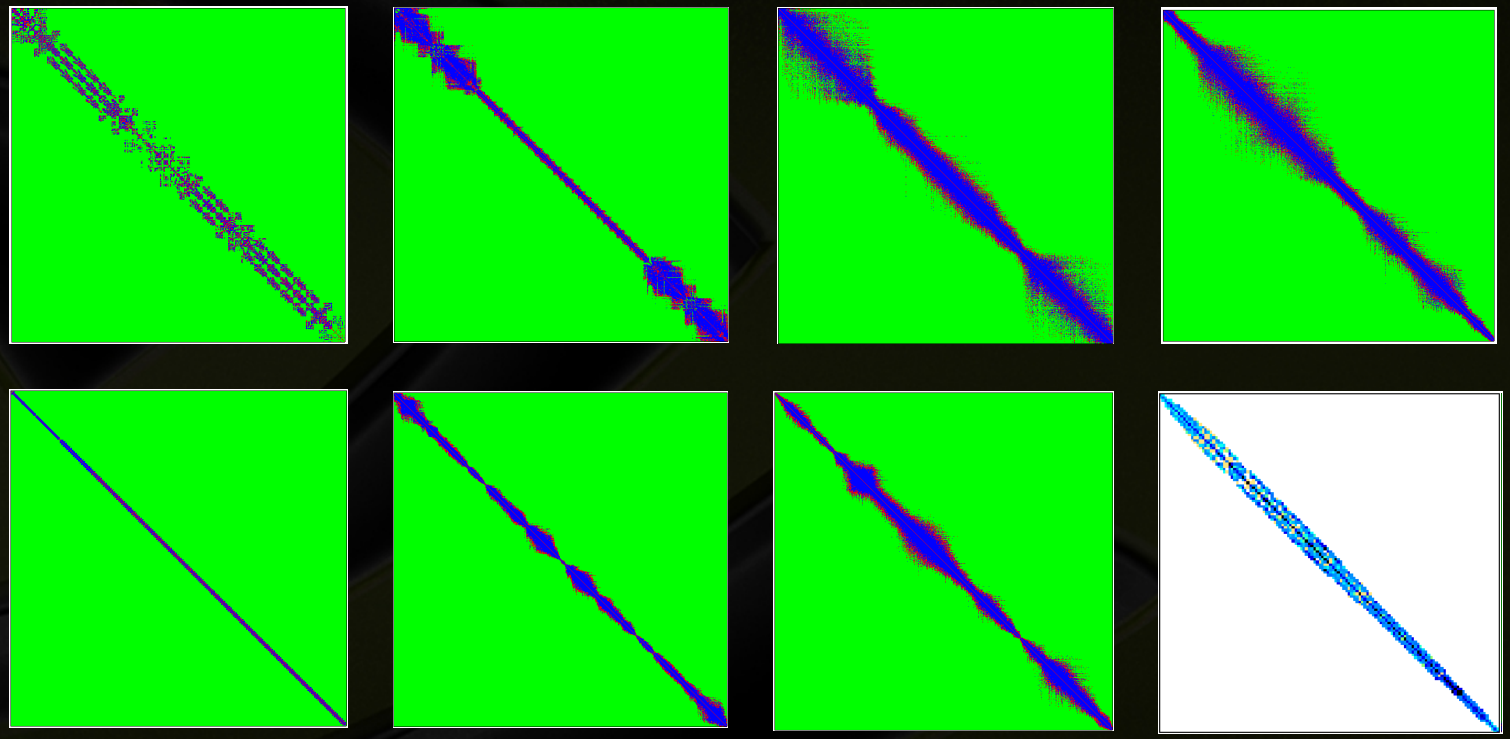
- Achieves fully coalesced memory access for A

Coherent Memory Access

- Resolved using ELLPACK data structure
- Next issues:
  - Coherent execution – idle threads
  - Wasted memory on device

FE Test Matrices

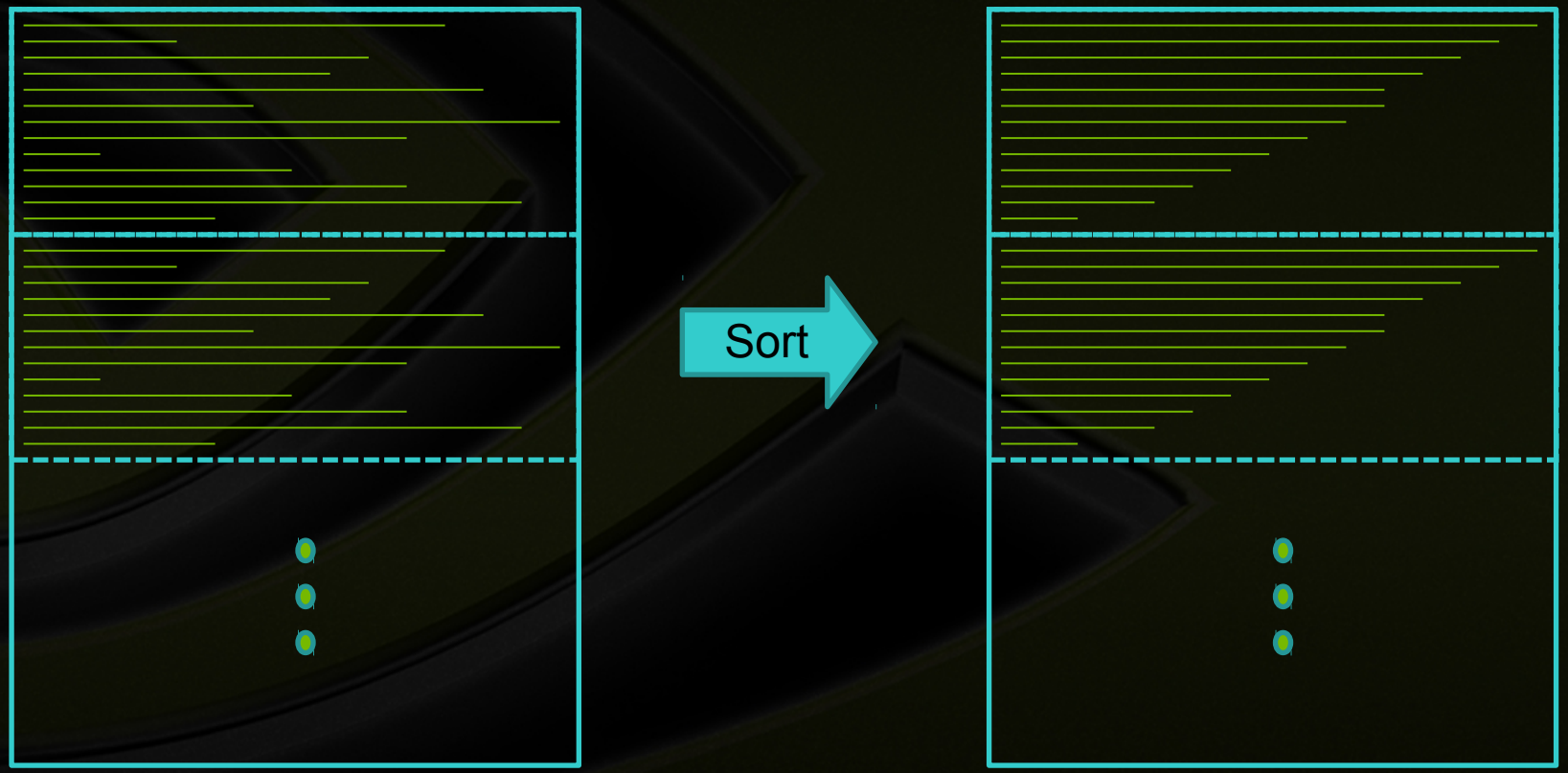
- After re-ordering, well clustered around diagonal



Florida Sparse Matrix Collection DNVS/shipsec1

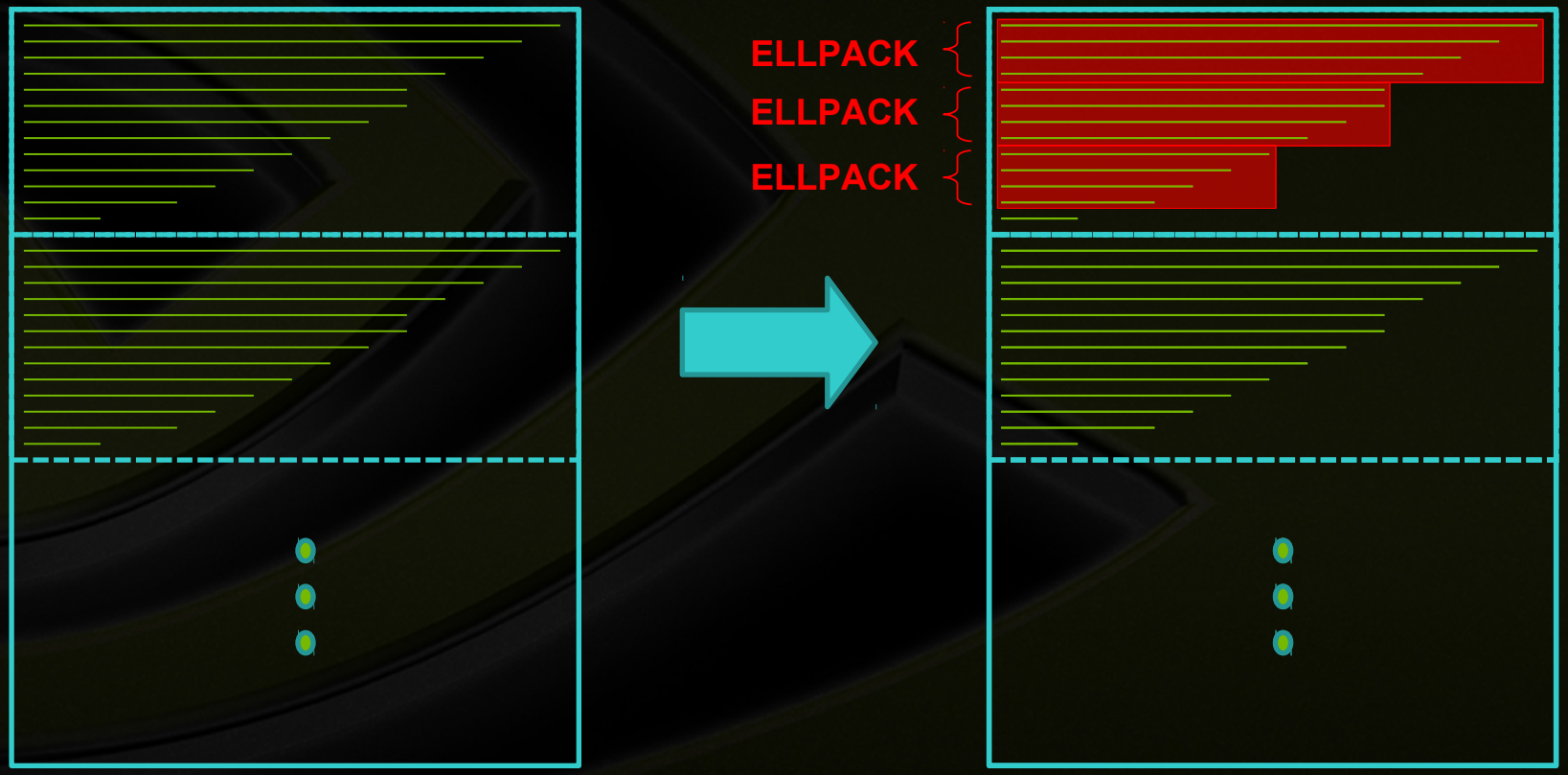
Wasted Memory: Row Reordering

- Break matrix into sections and sort within sections
  - Using sections of 64k rows – similar to JDS



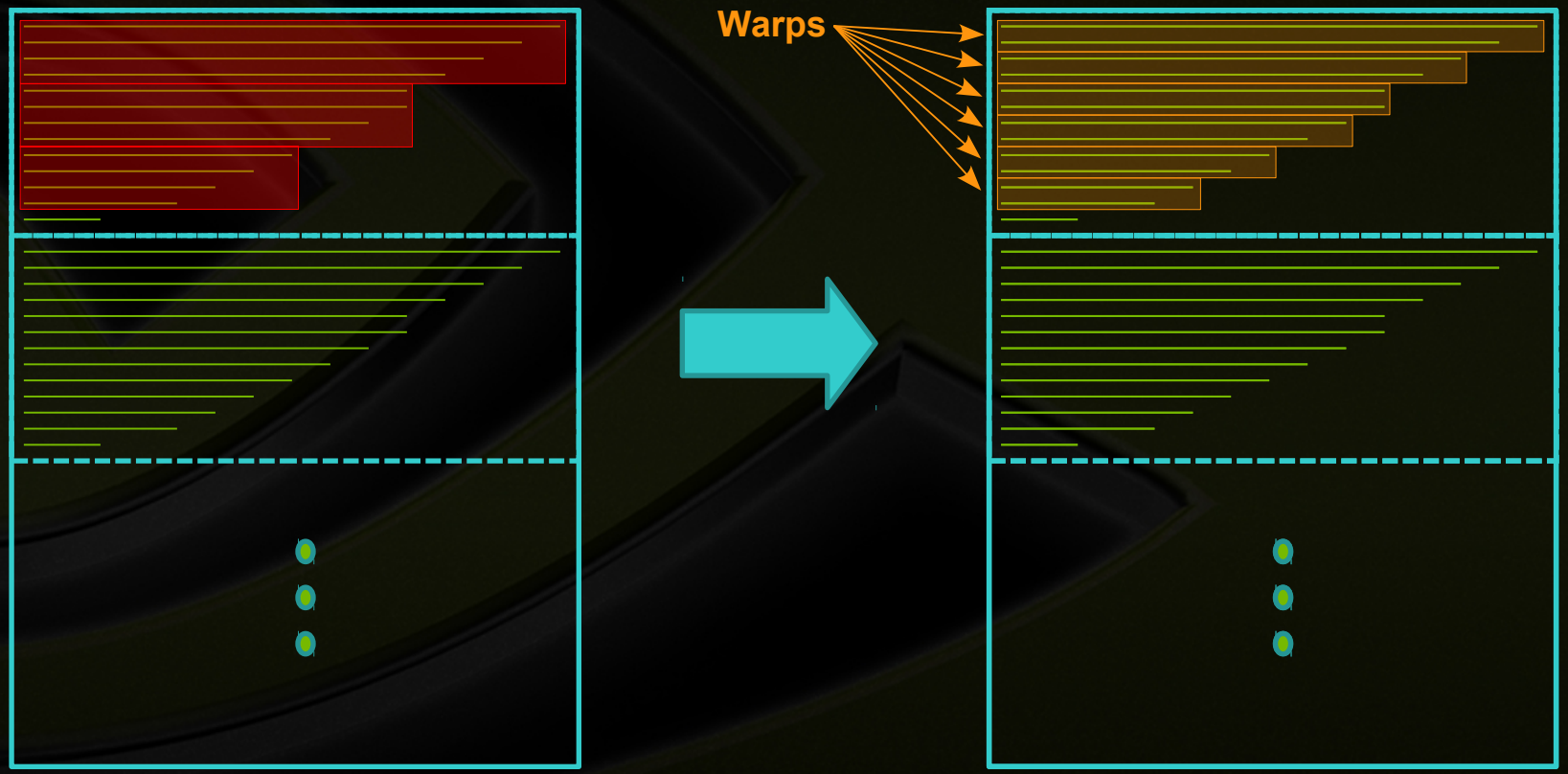
Wasted Memory: Row Reordering

- ELLPACK data structure applied in 64 line sections
  - Combined w/ sorting eliminates most wasted data (<3% waste)



Coherent Execution: Row Reordering

- Sorted rows also promotes coherent execution
  - Threads in warp have very similar workloads

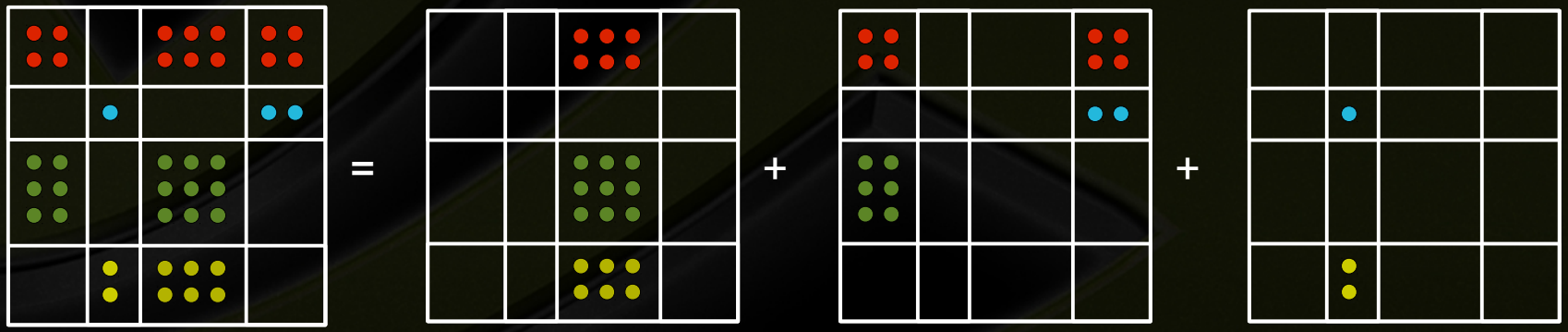


Coherent Execution / Memory Eff.

- Resolved issues:
  - Coherent execution – idle threads
    - resolved by sorting
  - Wasted memory on device
    - resolved by using multiple ELLPACK data structures
- Next issue:
  - Coherent execution – warp divergence

Separate kernel for each col. extent

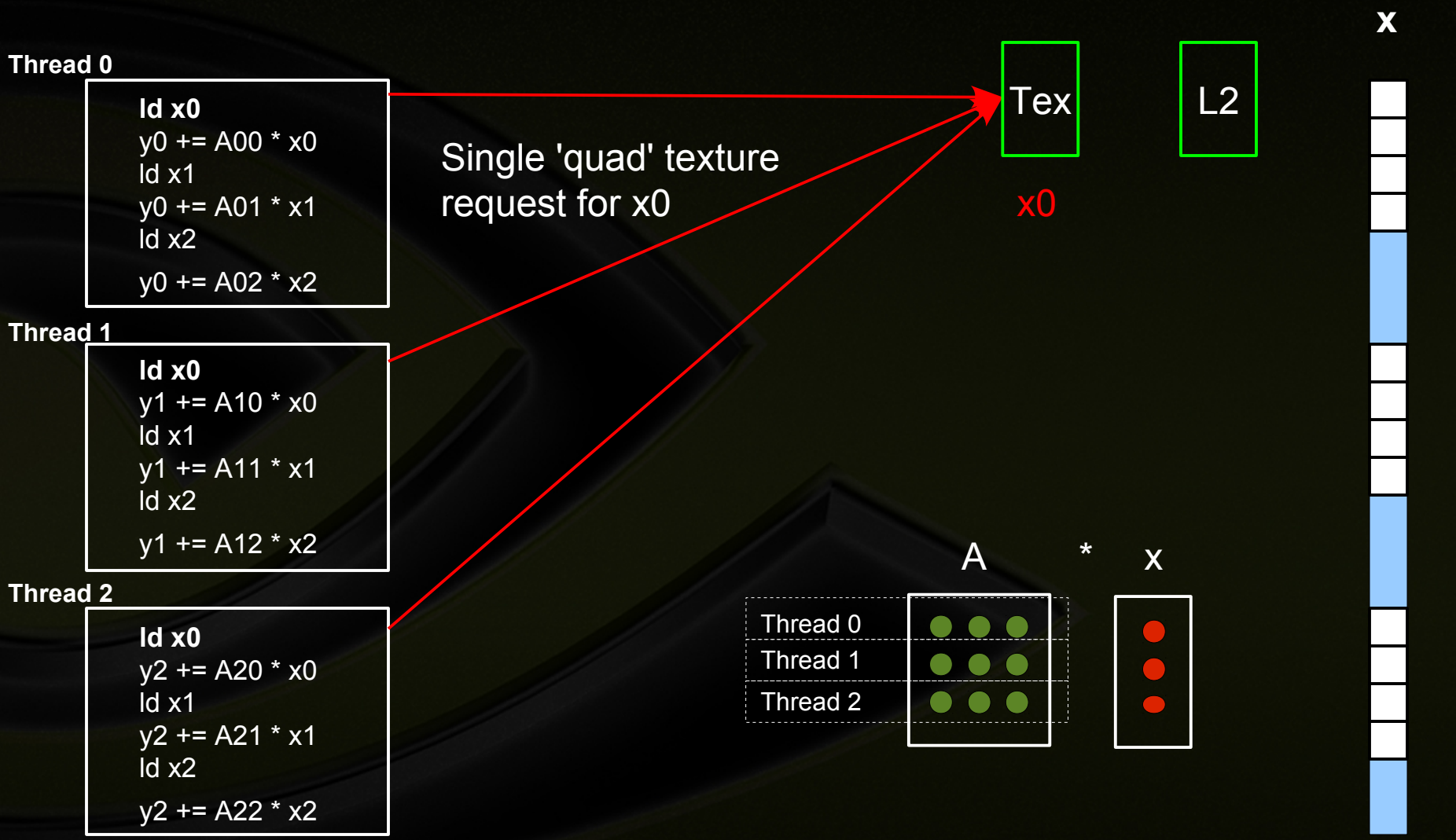
- Minimizes warp divergence
- Reduces data transfer
  - Column extent is now implicit
- Adds work to the data structure translation



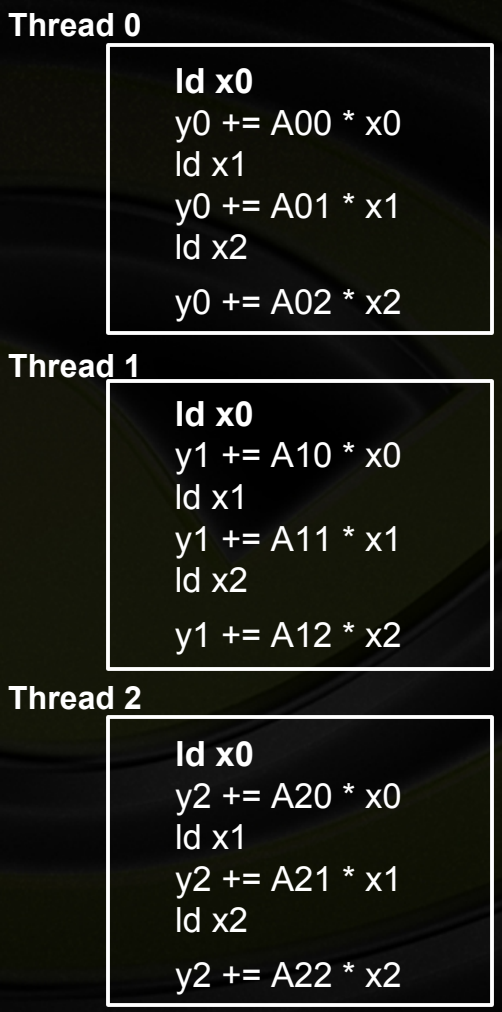
Warp Divergence

- Resolved issues:
  - Warp divergence
    - resolved by decomposing A matrix into submatrices with constant column extent
- Next issue:
  - Caching X values
  - Use Texture Cache
    - 32B texture cache line (4 doubles)
    - Not churned by A and column index values
    - Fast

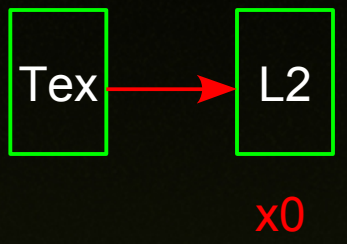
Standard caching of X in texture

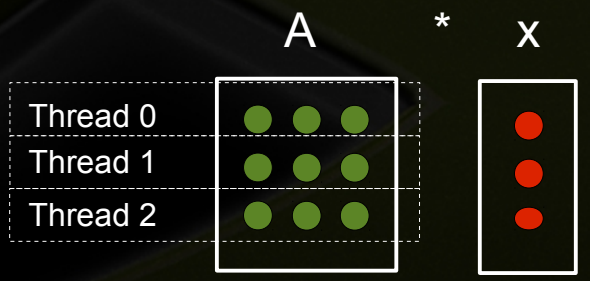


Standard caching of X in texture



SMs now stalled waiting for data





Standard caching of X in texture

Thread 0

```
ld x0
y0 += A00 * x0
ld x1
y0 += A01 * x1
ld x2
y0 += A02 * x2
```

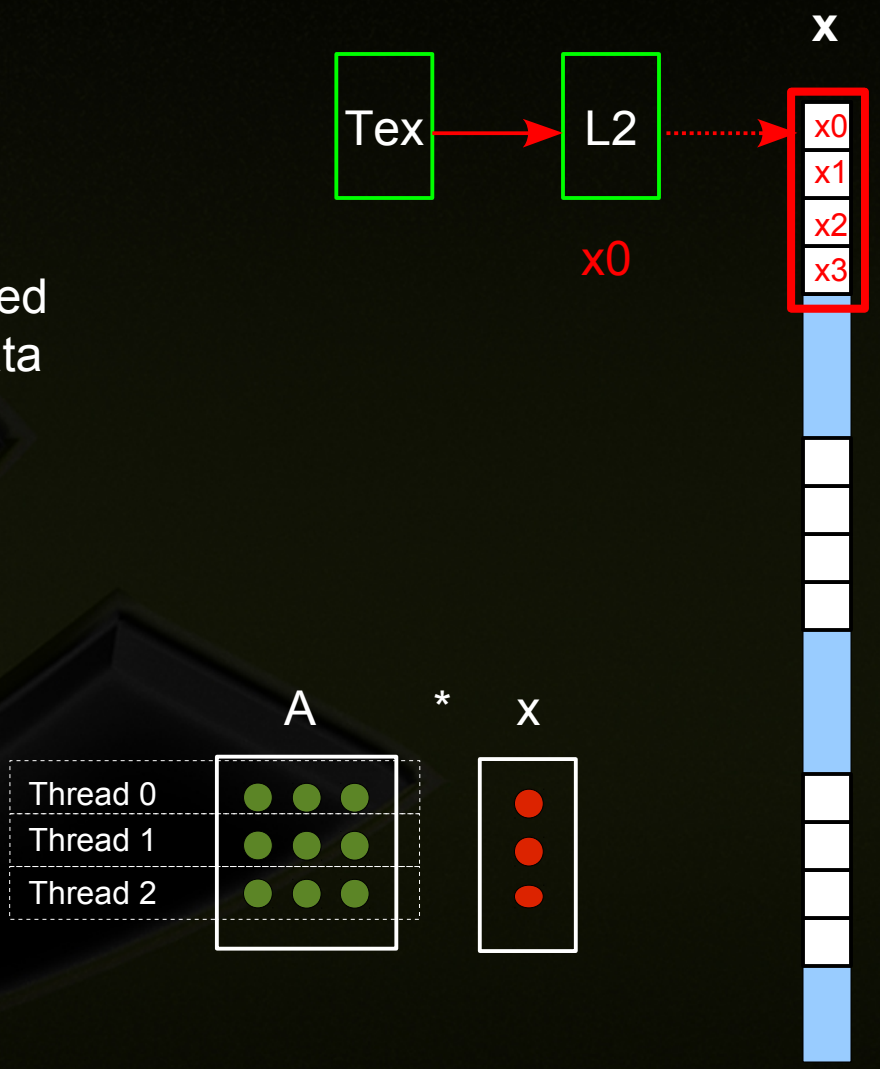
Thread 1

```
ld x0
y1 += A10 * x0
ld x1
y1 += A11 * x1
ld x2
y1 += A12 * x2
```

Thread 2

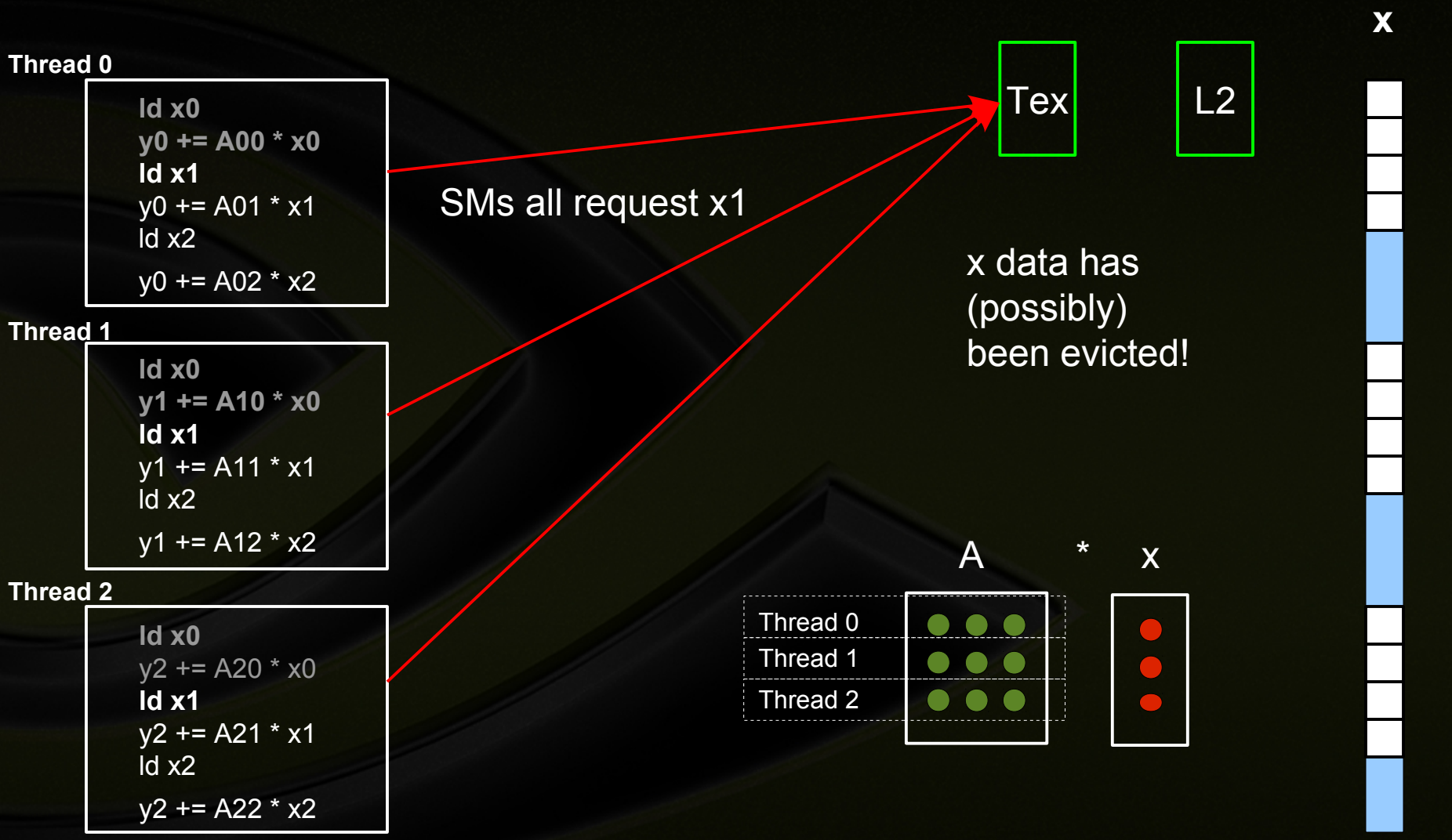
```
ld x0
y2 += A20 * x0
ld x1
y2 += A21 * x1
ld x2
y2 += A22 * x2
```

SMs still stalled waiting for data

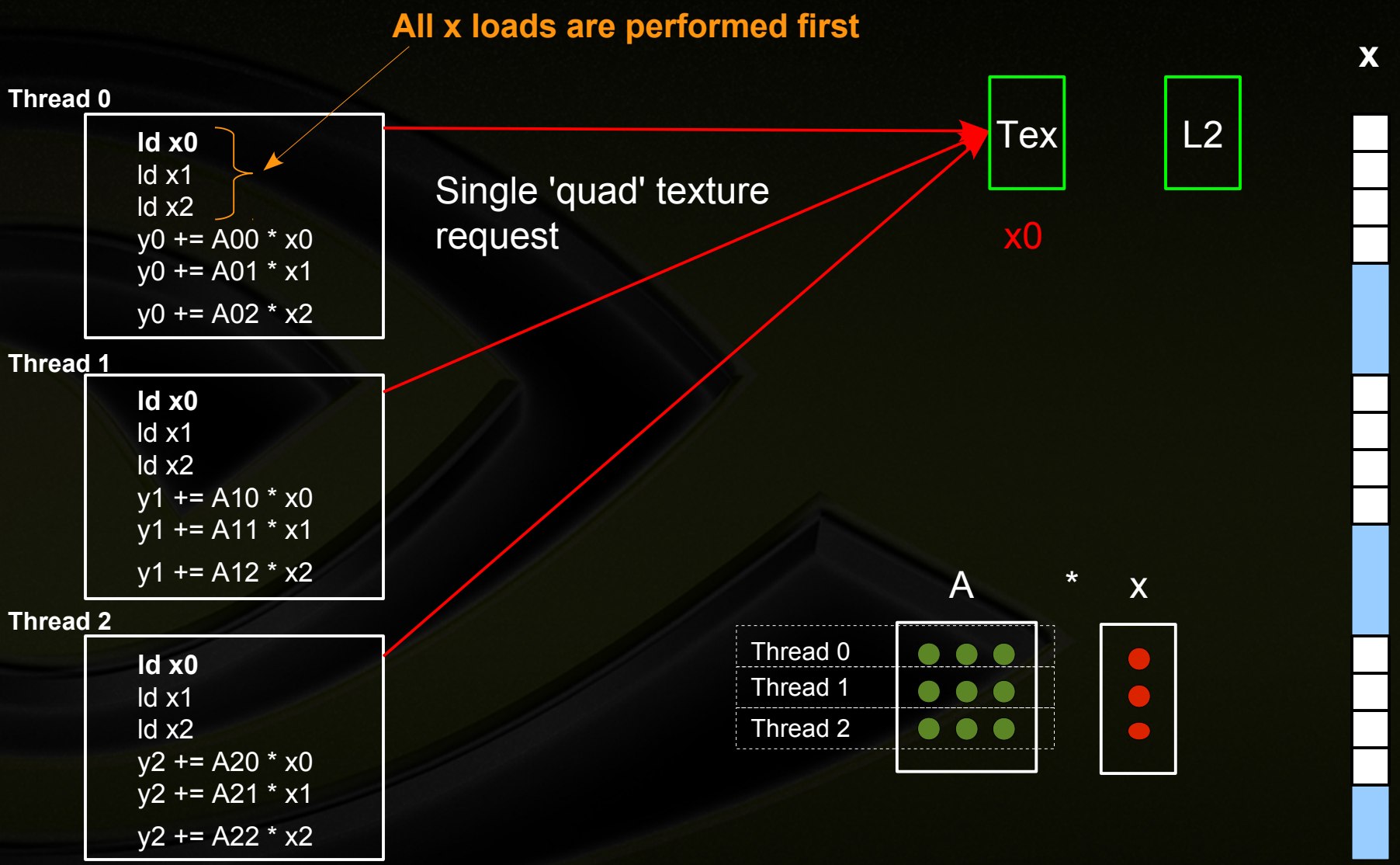


Standard caching of X in texture

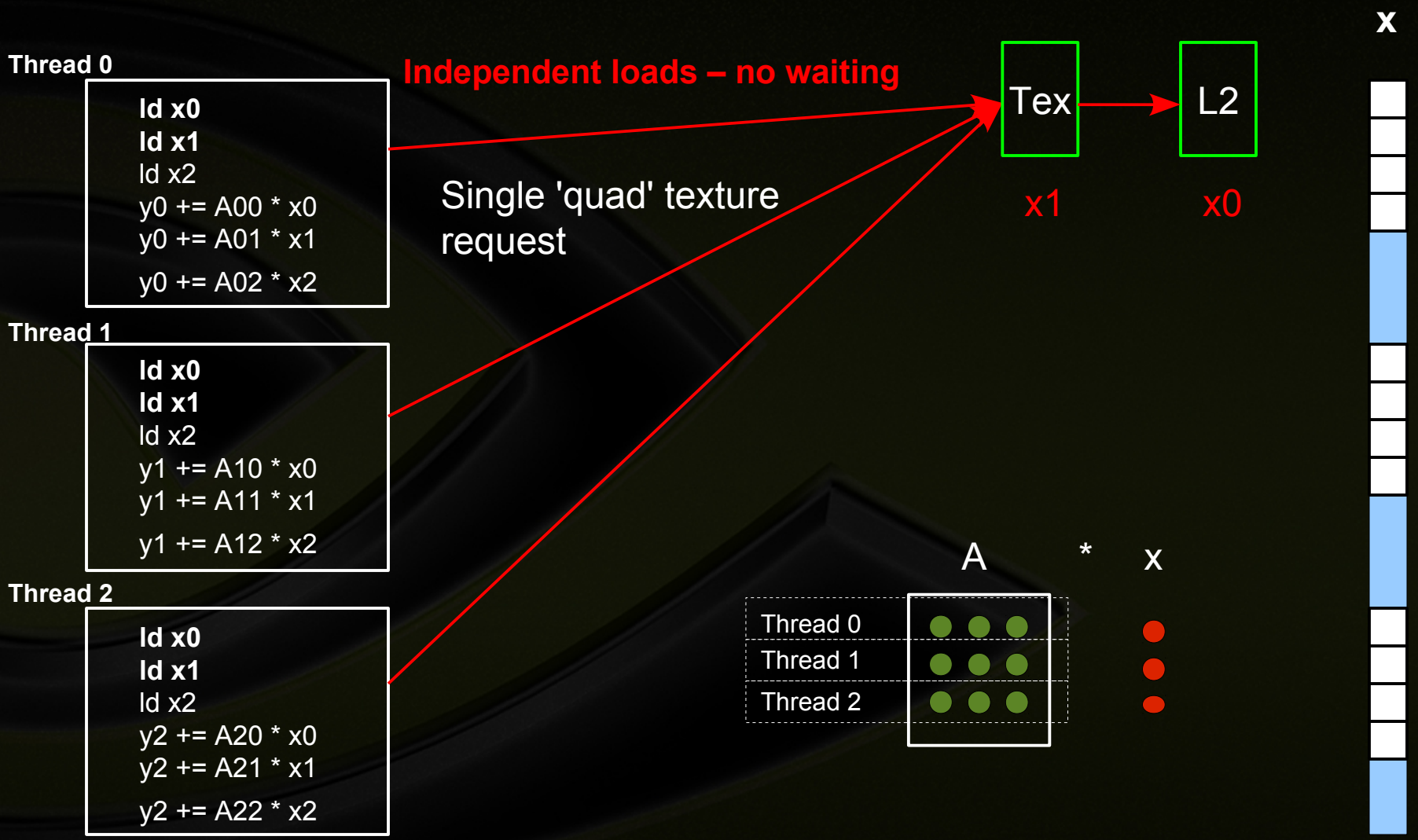
Standard caching of X in texture



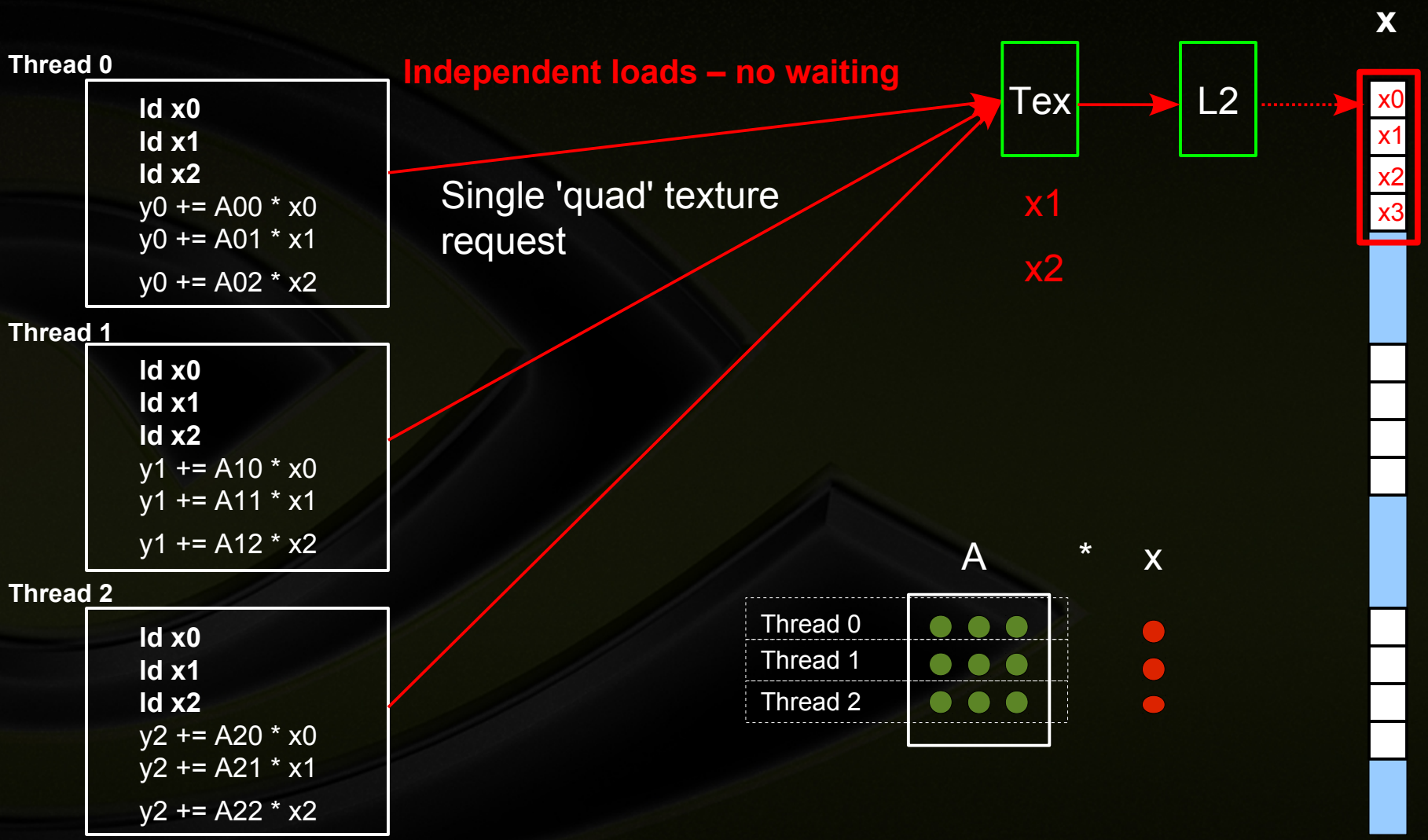
Optimal caching of X in texture



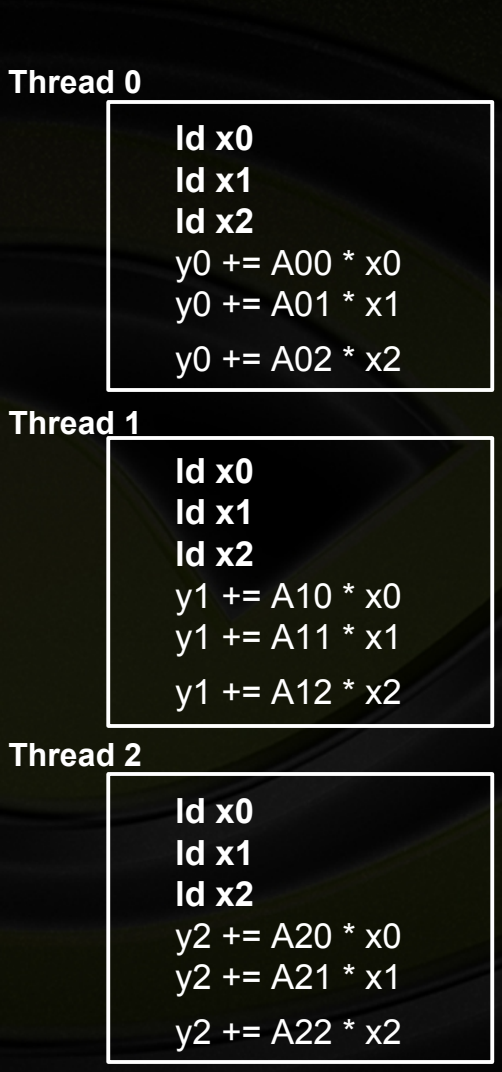
Optimal caching of X in texture



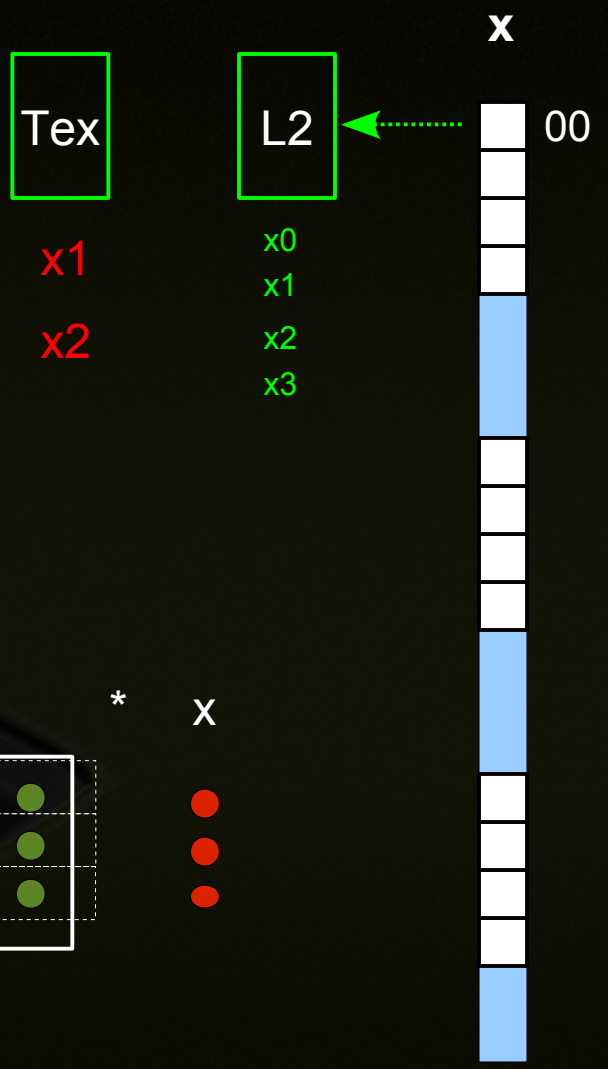
Optimal caching of X in texture

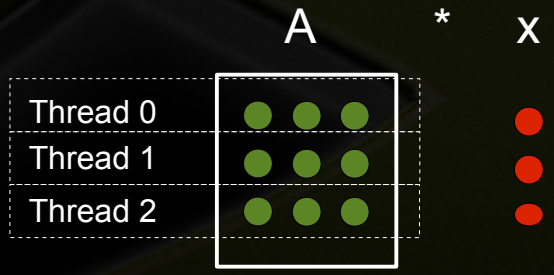


Optimal caching of X in texture

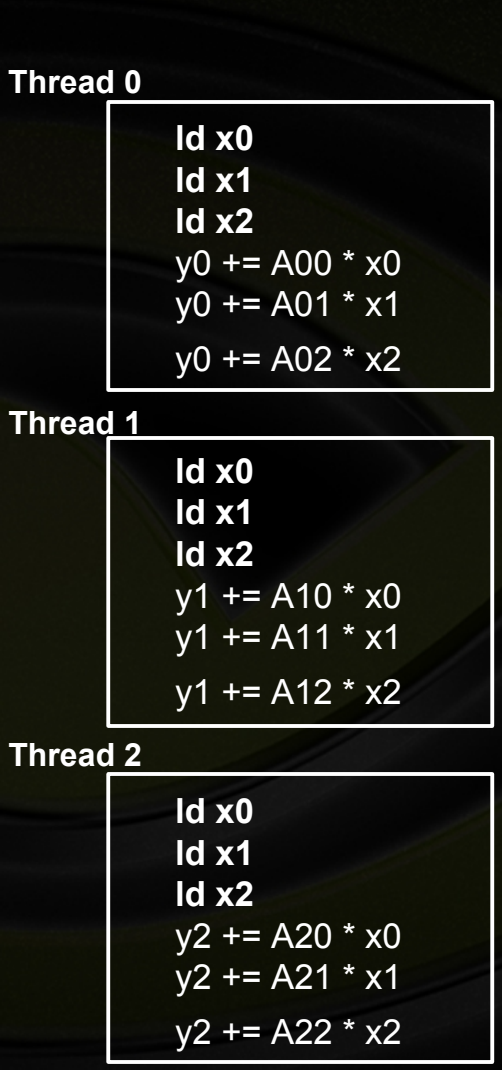


SMs now stalled waiting for data

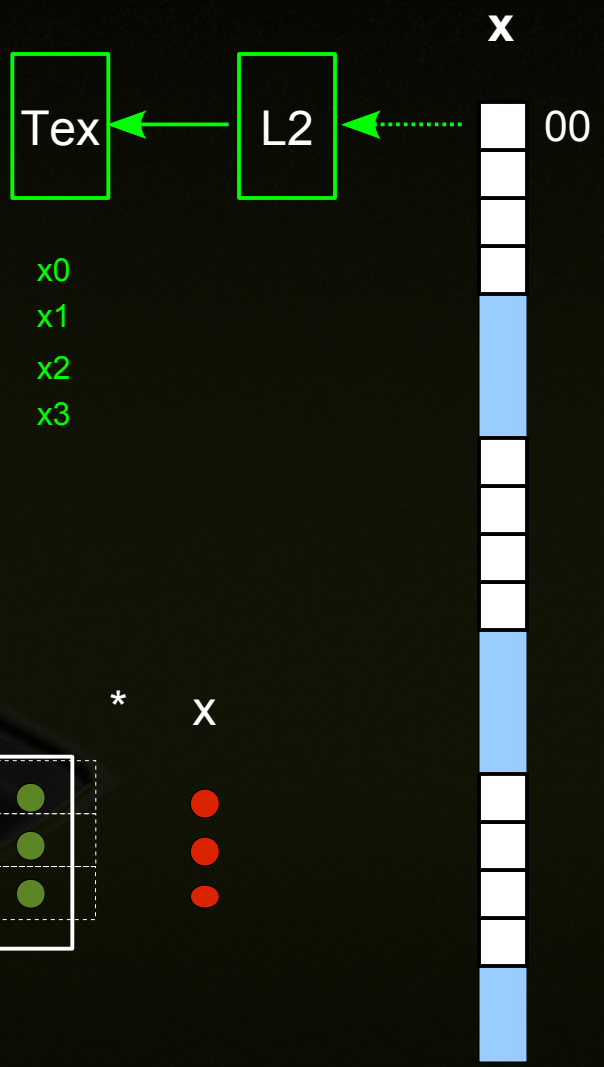


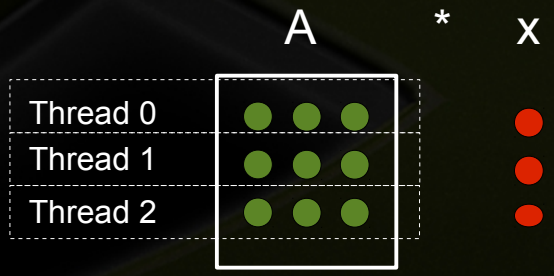


Optimal caching of X in texture

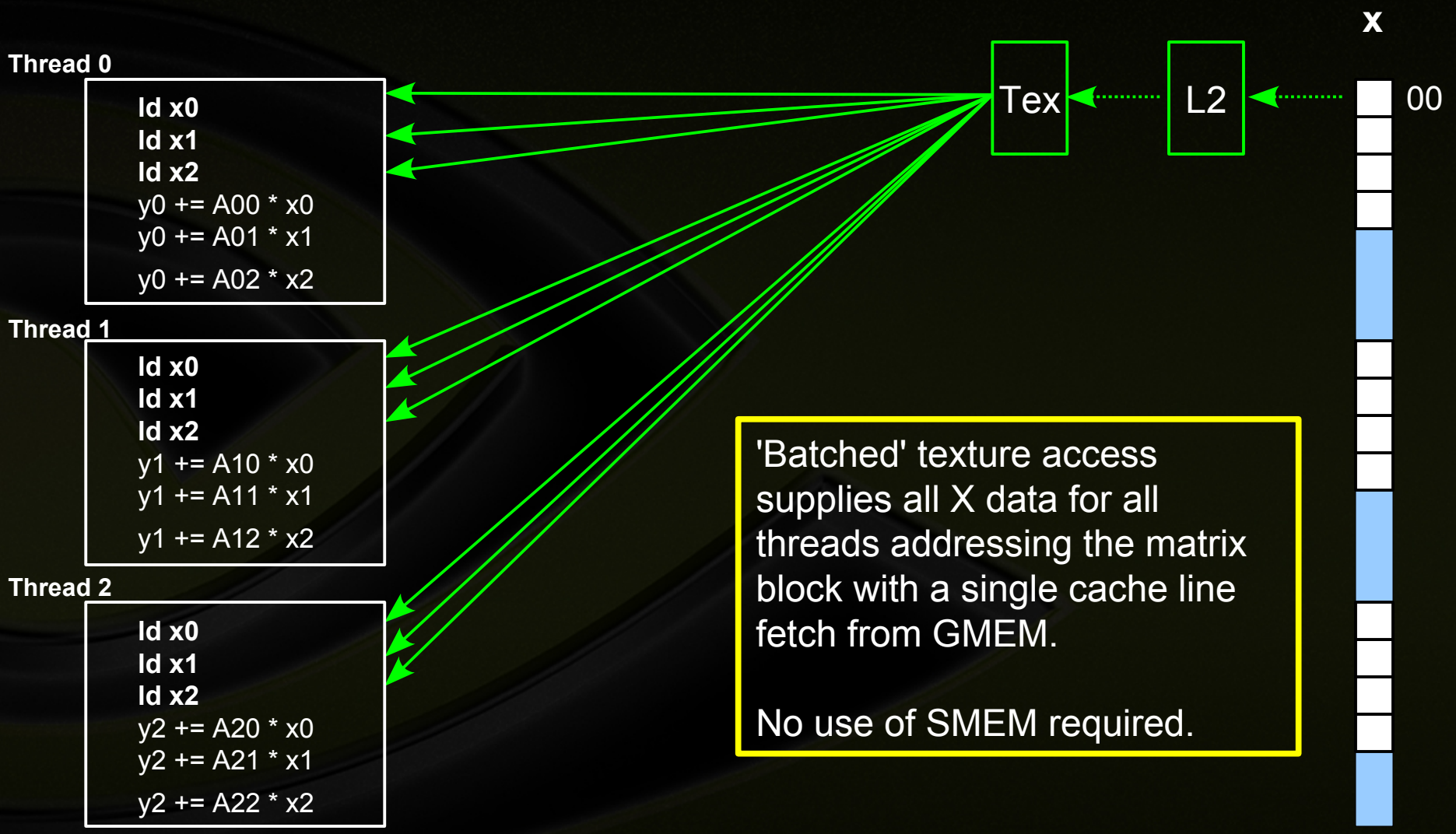


SMs still stalled waiting for data





Optimal caching of X in texture



'Batched' texture access supplies all X data for all threads addressing the matrix block with a single cache line fetch from GMEM.

No use of SMEM required.

BSpMV Kernel

templated (nice)

based on column extent

```
template <unsigned char nExt>
__global__ void AxBBkernelT( ...

  // Initializations (nblocks, ibrow, vp, astride)
  ...

  // Loop over nonzero blocks in this row
  for ( unsigned int iblock=0; iblock<nblocks; ++iblock) {

    // compute column for this block - has been aligned
    col = padding[nExt] * nzBlocks[ blockstride*iblock+ibrow ];

    // Loop over column extent: y = Ax
    for ( int i=0; i<nExt; i++ ) {
      texval[i] = tex1Dfetch (tex_x_double, col++ );
      ry += vals[vp] * __hiloint2double ( texval[i].y, texval[i].x );
      vp+= astride;
    }
  }
...
```

nvcc does the unrolling and reordering since nExt is a const.

Additional Algorithm Details

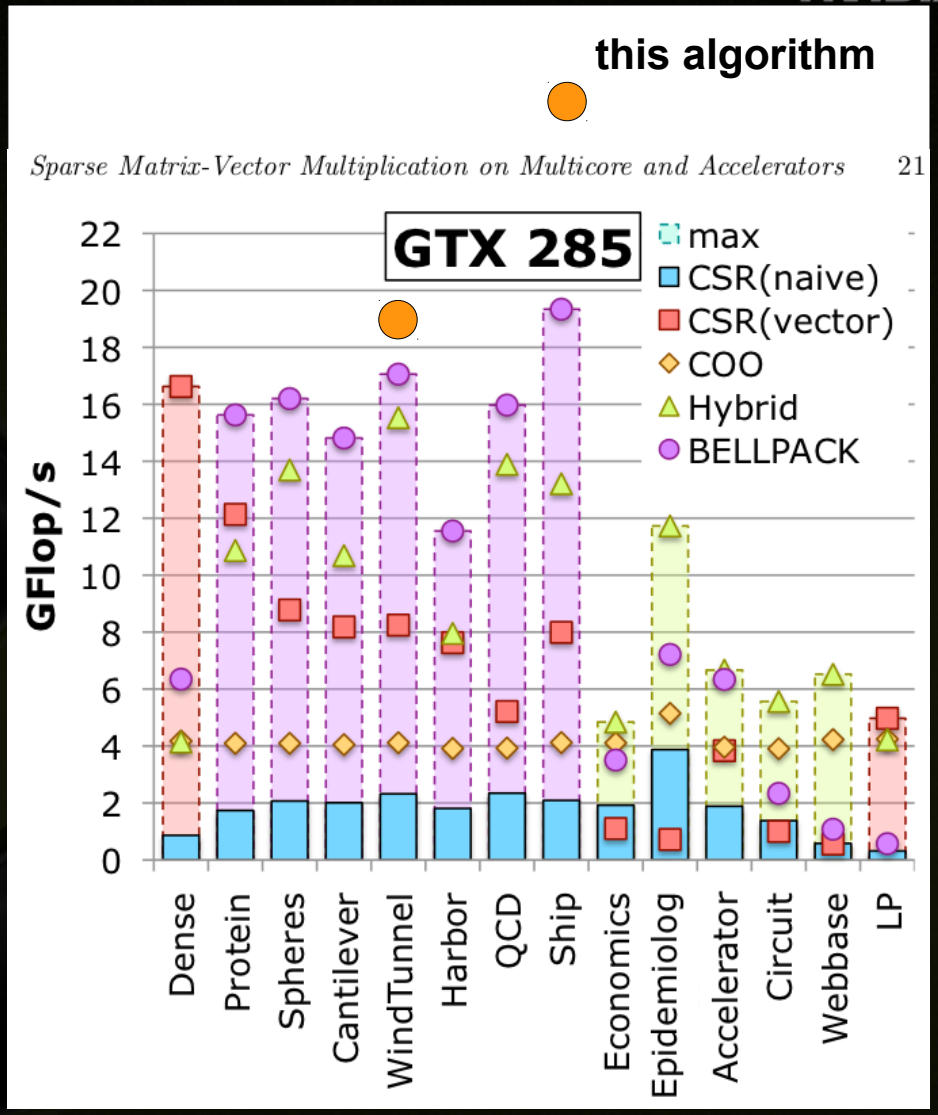
- X vector is 'padded' to column extent by 'block row'
  - A given 'column extent', n, only accesses X 'blocks' with the same extent
  - So all other x 'blocks' are set to n
  - x location can be indexed directly from block index
  - (i.e. 3 doubles are 'padded' to 4 (32 Bytes))
  - Removes a level of indirection / reduces communication
- Requires a 'swizzle' for every column extent
  - Separate kernel
- Row permutation, reverse permutation and summation of intermediate results are all done on the GPU
  - Integrated with BSpMV kernel

Competitive Performance

- Dongarra, Bader, Kurzak, "**Scientific Computing with Multicore and Accelerators**", Chapman & Hall 2010
- Chapter 4: Williams, Bell, Choi, Garland Oliker, Vuduc, "**Sparse Matrix-Vector Multiplication on Multicore and Accelerators**"
- Florida Sparse Matrix Collection
  - Williams set

- **BELLPACK peak of 19.5 Gflops on GTX285**
  - **Ship model**
  - 159 GB/s (vs. 144GB/s on C2070)
  - Expect ~17.5 Gflops on C2070

- **Present algorithm achieves 23.8 Gflops on C2070**
  - **Ship model**
  - 1.35x improvement



Algorithm Performance

- 27 Gflops achieved (28.5 in kernel)
  - For best case of block extents: 6 x 6
  - Close to expected peak of 30.5 (simple analysis)
  - ~4.2 x vs. socket's theoretical max (x5670)
  - ~6 x vs. socket's published max perf.
- Performance Expectations vs. CPU
  - 27 Gflops/s achieved on GPU – not leveraging symmetry
  - Kernel's theoretical max is 6.4 Gflops/s (x5670 socket)
  - Perfect leveraging of symmetry would give 12.8 Gflops
  - Max observed CPU perf is ~4 Gflops/s (~6x speedup with GPU)
  - Expect 3x vs SandyBridge

Practical Considerations

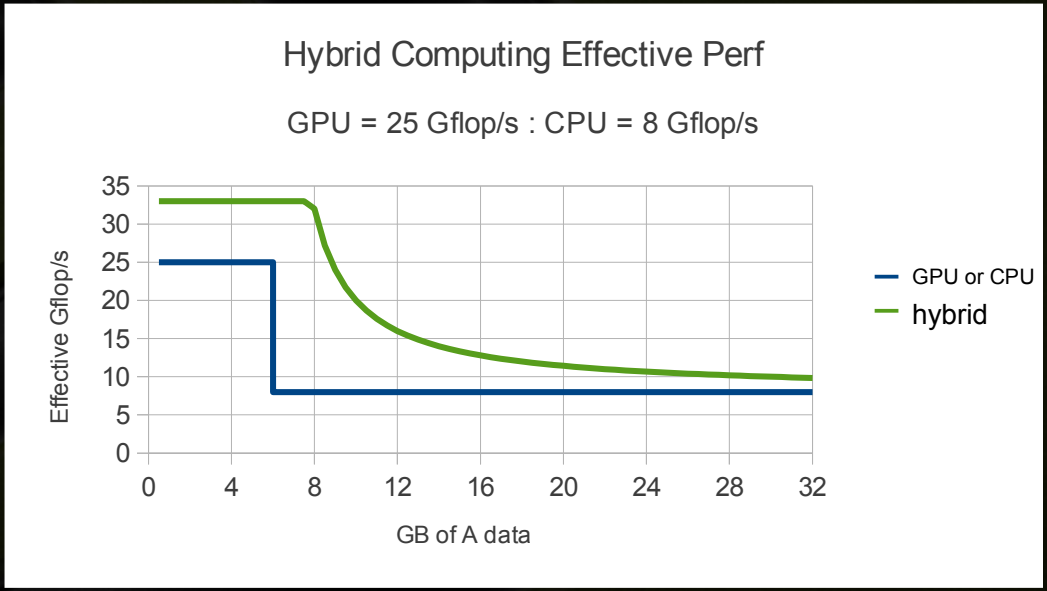
- GPU performance is dependent on block size
  - Larger is better - Prefer multiples of 4
  - 27 Gflops/s achieved for a block size of 6x6
  - 25 Gflops/s achieved for a block size of 3x3
  - Performance is poor for thermal analysis (1x1 blocks) (~8.5 Gflops/s)
- GPU-friendly datastructure is very important for performance
  - Extremely unlikely the parent code will adopt this datastructure
- Datastructure translation costs ~200 GPU iterations (40 on CPU)
  - If nonzero structure cam be reused translation cost is 40 GPU iterations (or about 8 CPU itrations)

Further Improvements

- Multi-GPU support
  - Scales well to multiple GPUs
  - Large models see 1.95x across 2 GPUs
- Hybrid computing
  - Leverage GPU + CPU for marginal perf. Improvement
  - Alleviates device memory limitation
- Leveraging Symmetry
  - Use Shared Memory cache (in progress)

Hybrid Computing – Large Matrices

- For large matrices, only a portion of the matrix is multiplied on the GPU
  - Eliminates device memory 'cliff'
  - Any size matrix will see a performance benefit




Thank
You
NVIDIA

Wind Tunnel

- 200k rows
- Nz per row ~55
- Large variety of extents
  - ndof = 1 : 4701
  - ndof = 2 : 1230
  - ndof = 3 : 1237
  - ndof = 4 : 17
  - ndof = 5 : 148
  - ndof = 6 : 34373