Sparse Matrix-Vector Multiplication & Texture

Lesson 22

David Tarjan (most slides from Nathan Bell's presentation on SpMV)

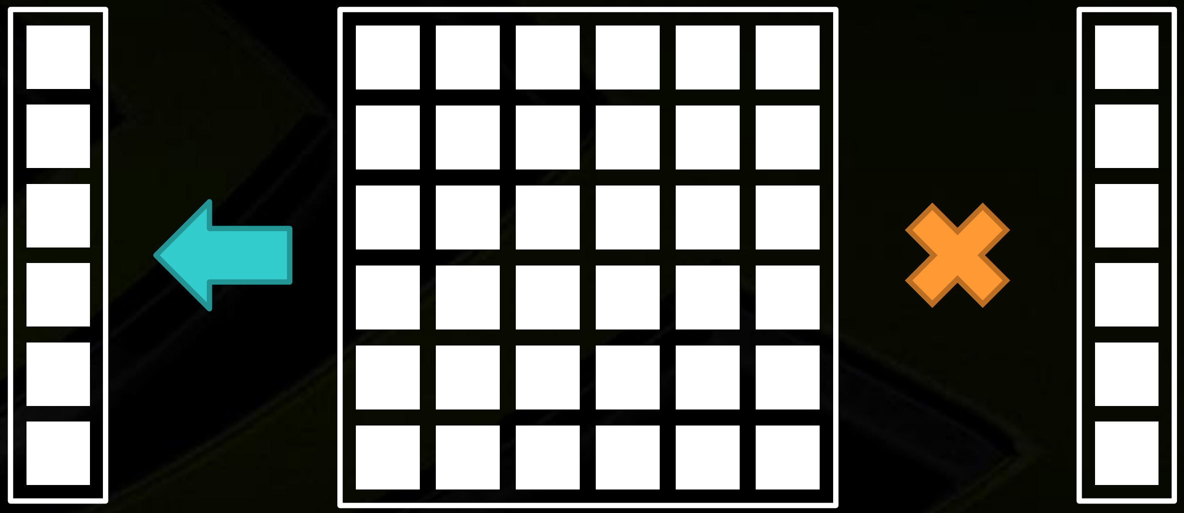
NVIDIA

Overview

- Sparse Matrix Vector Product
  - Performance Considerations
  - Matrix Formats
- Texture
  - Overview
  - Example Usage for SpMV

Dense Matrix-Vector Multiplication

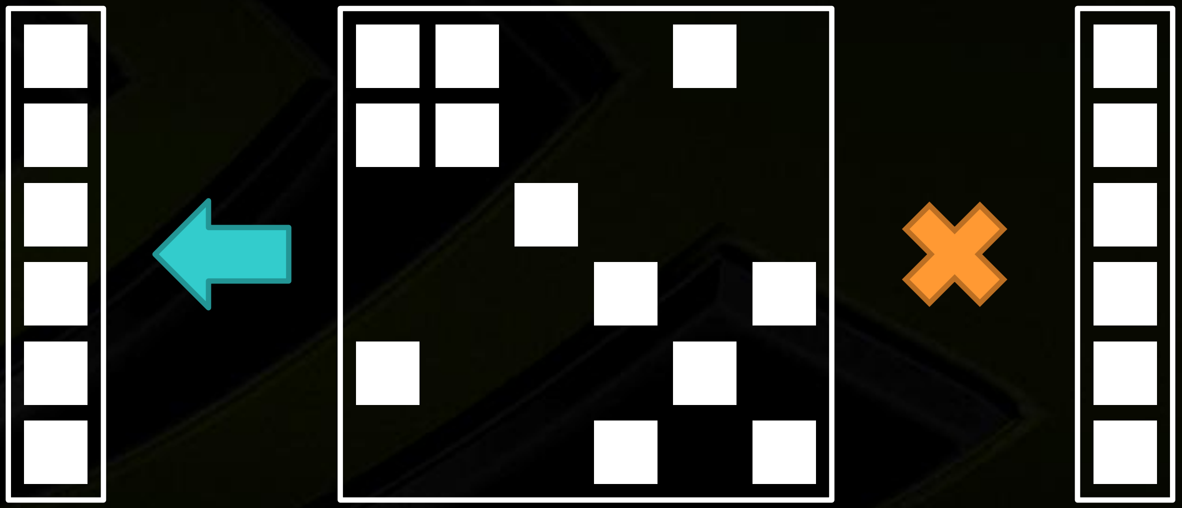
- SGEMV / DGEMV in BLAS
  - Memory-bound performance



Dense Matrix

Sparse Matrix-Vector Multiplication

- **One multiply-add per nonzero entry**
  - **Some reuse of vector data**



Sparse Matrix

Performance

- Performance is *Memory-Bound*
  - 5-20 GFLOP/s is typical
- Low Arithmetic Intensity
  - 2 flops :   8+ bytes (`float`)
  - 2 flops : 16+ bytes (`double`)
- Primary objective
  - Use memory bandwidth efficiently

Performance

- **Tesla C2050 floating point performance**
  - **Single 1,030 GFLOP/s (peak)**
  - **Double   515 GFLOP/s (peak)**
- **Tesla C2050 memory bandwidth**
  - **144 GB/s (peak)**
- **Intensity Threshold**
  - **7.14 FLOP : Byte (single)**
  - **3.57 FLOP : Byte (double)**

Performance

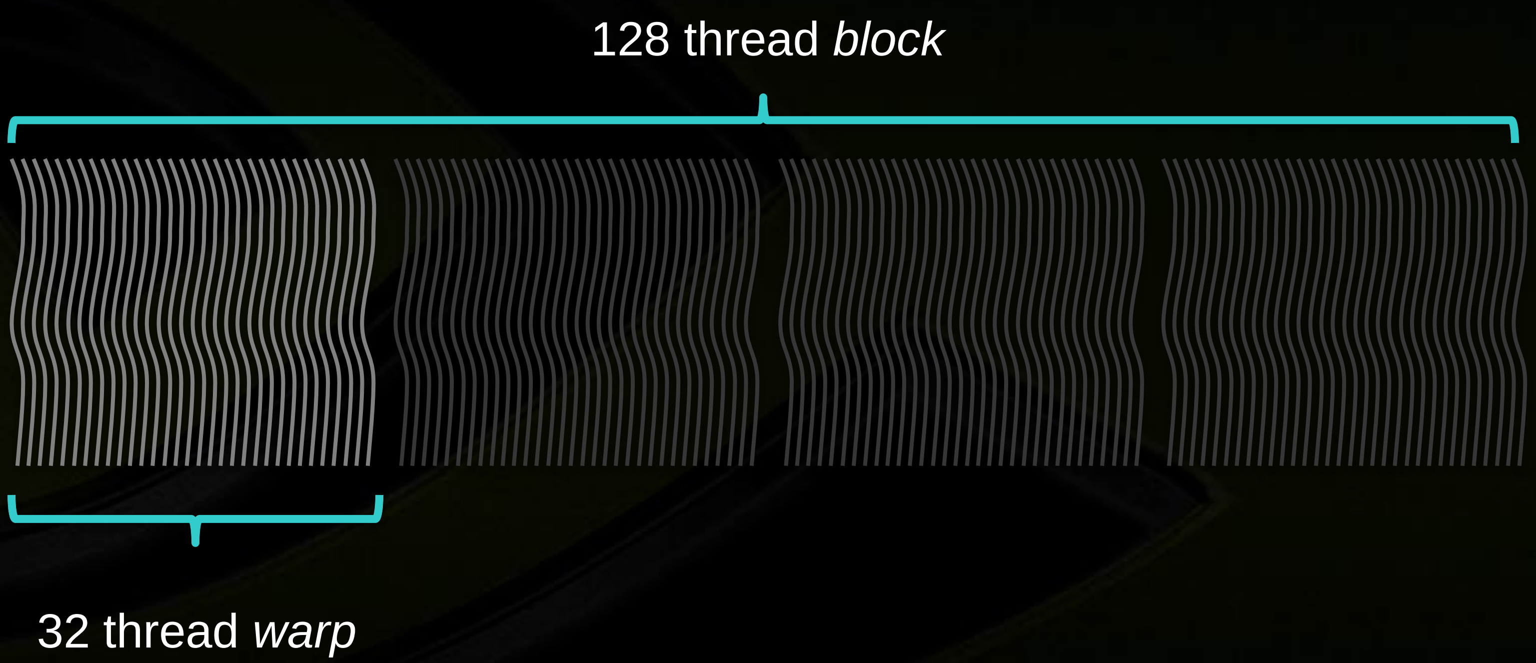
- Tesla C2050 threshold
  - 7.14 FLOP : Byte (single)
  - 3.57 FLOP : Byte (double)
- *Dense* Matrix-Vector Multiplication Intensity
  - 0.25 - 0.50 FLOP : Byte (single)
  - 0.12 - 0.25 FLOP : Byte (double)
- *Sparse* Matrix-Vector Multiplication Intensity
  - 0.12 - 0.50 FLOP : Byte (single)
  - 0.08 - 0.25 FLOP : Byte (double)

Performance Considerations

- **Use memory bandwidth efficiently**
  - **Memory coalescing**
- **Expose lots of fine-grained parallelism**
  - **Need thousands of threads**
- **Find opportunities for reuse**
  - **Make use of caching**

Memory Coalescing

- **Recall: *blocks* divided into physical *warps***

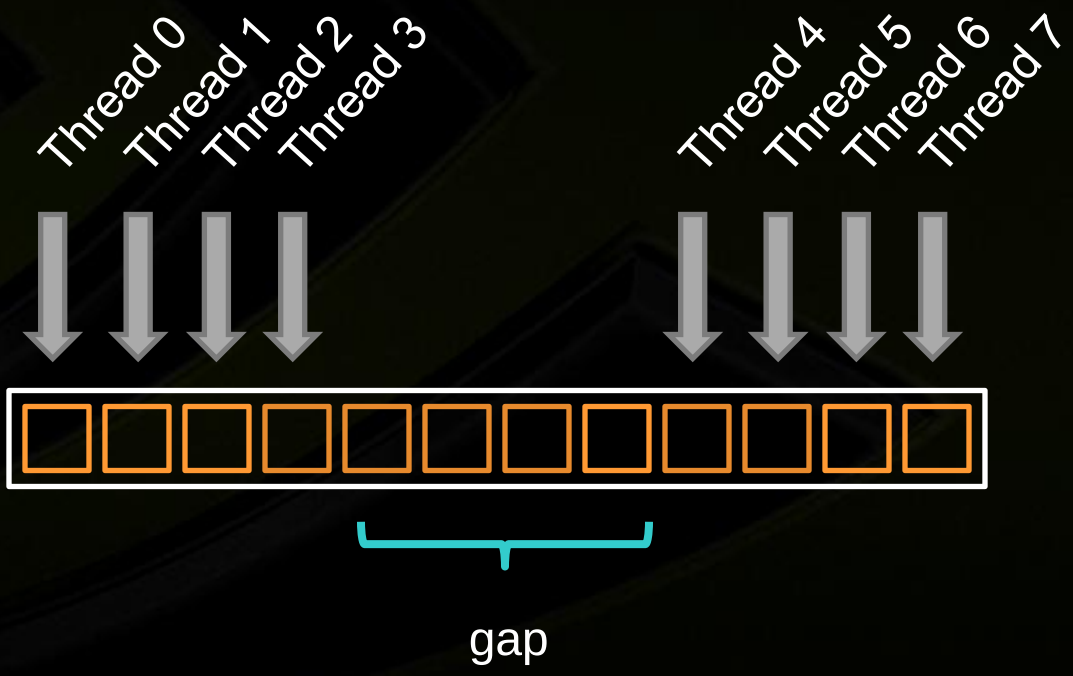


Memory Coalescing

- **Fully Coalesced Memory Access**

Memory Coalescing

- **Partially Coalesced Memory Access**

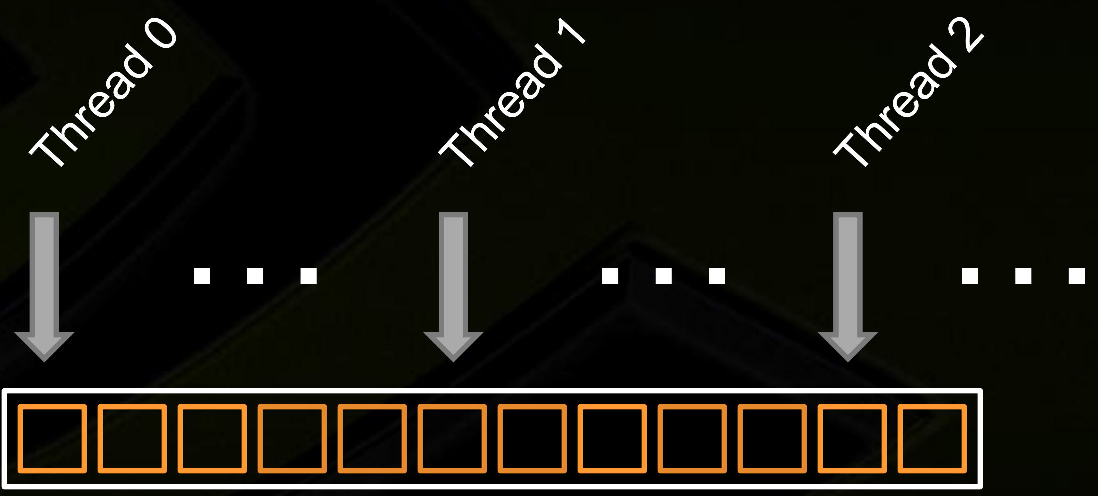


Memory Coalescing

- **Misaligned memory access**

Memory Coalescing

- **Uncoalesced Memory Access**
  - **Separated by 32+ words**

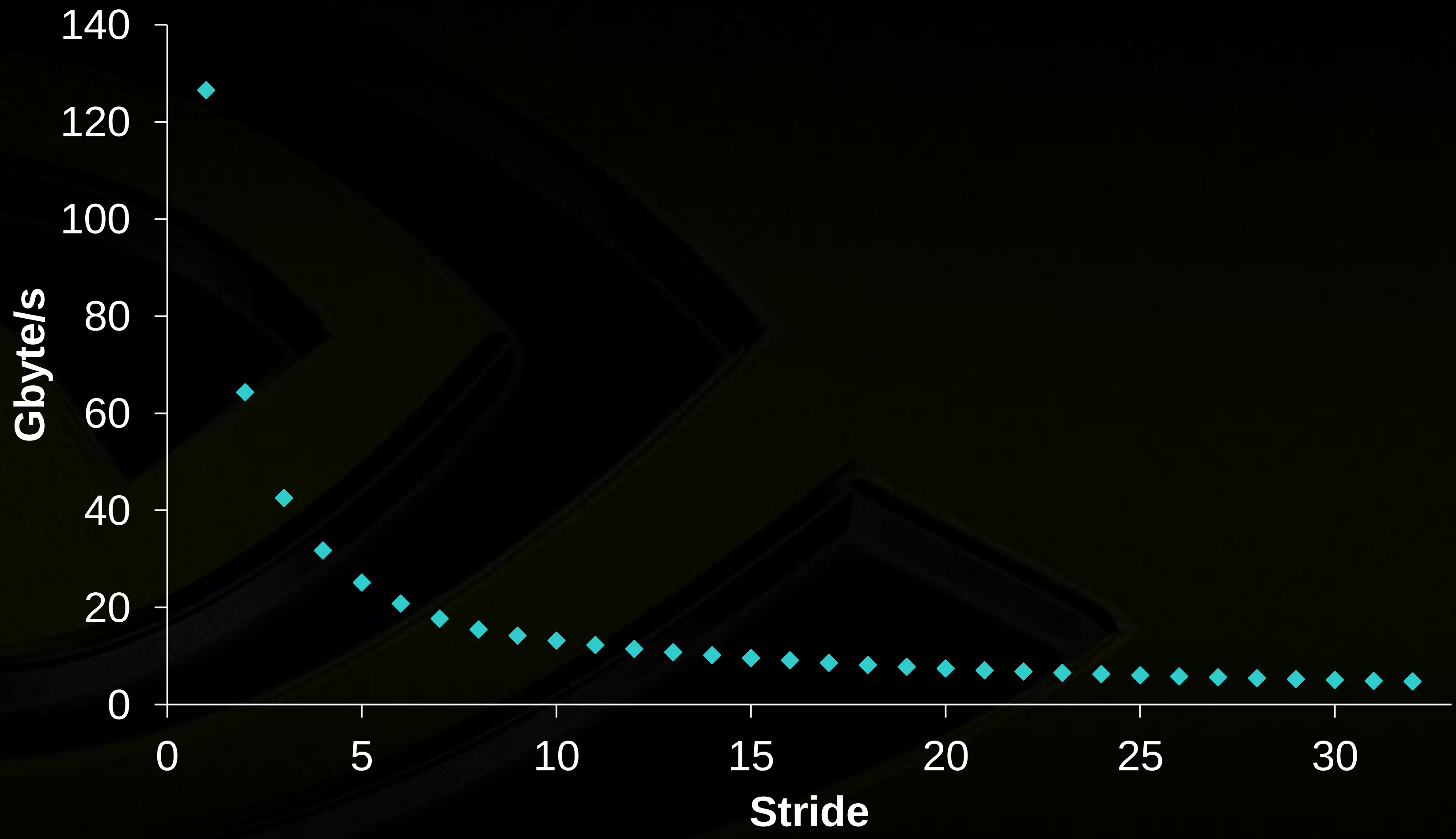


Memory Coalescing (SAXPY)

- **Example: SAXPY with stride**
  - **Fully Coalesced when `stride` is 1**

```
for (int i = 0; i < N; i++)
  y[stride * i] += a * x[stride * i];
```

Memory Coalescing (SAXPY)



Memory Coalescing (SAXPY)

- **Example: SAXPY with offset**
  - **Aligned when `offset` is 0**

```
for (int i = 0; i < N; i++)
  y[i + offset] += a * x[i + offset];
```

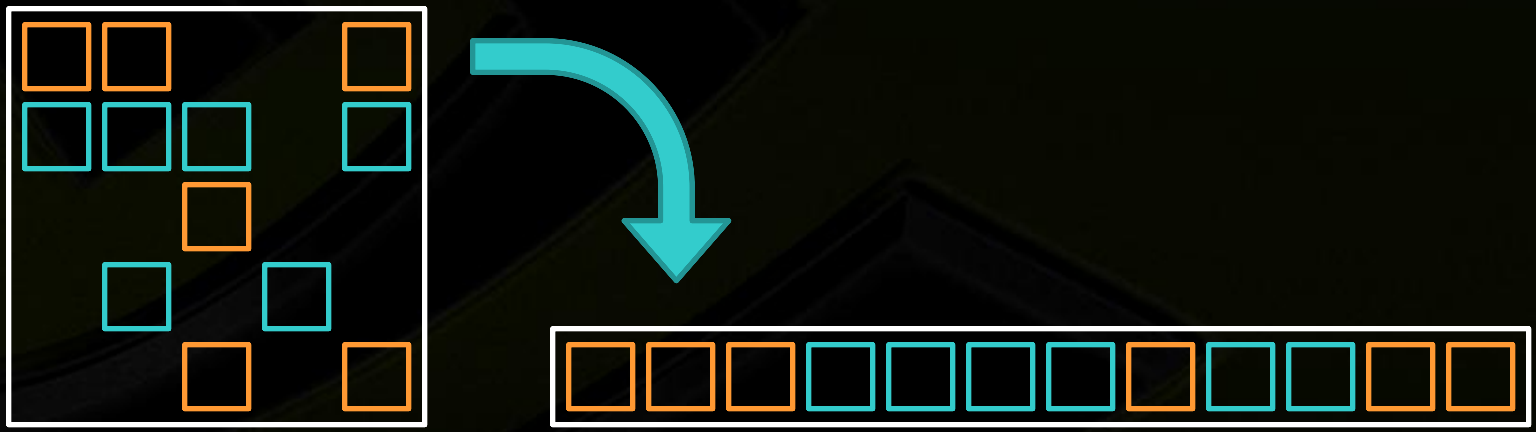
Memory Alignment (SAXPY)

Types of Memory Access

- Reading matrix structure
  - Determined by matrix format
  - Most bandwidth consumption
- Reading source vector (x)
  - Determined by matrix structure
  - Little control over access pattern
  - Potential reuse
- Writing destination vector (y)
  - Little bandwidth consumption

Compressed Sparse Row (CSR)

- Rows laid out in sequence
- Inconvenient for fine-grained parallelism



CSR SpMV (serial)

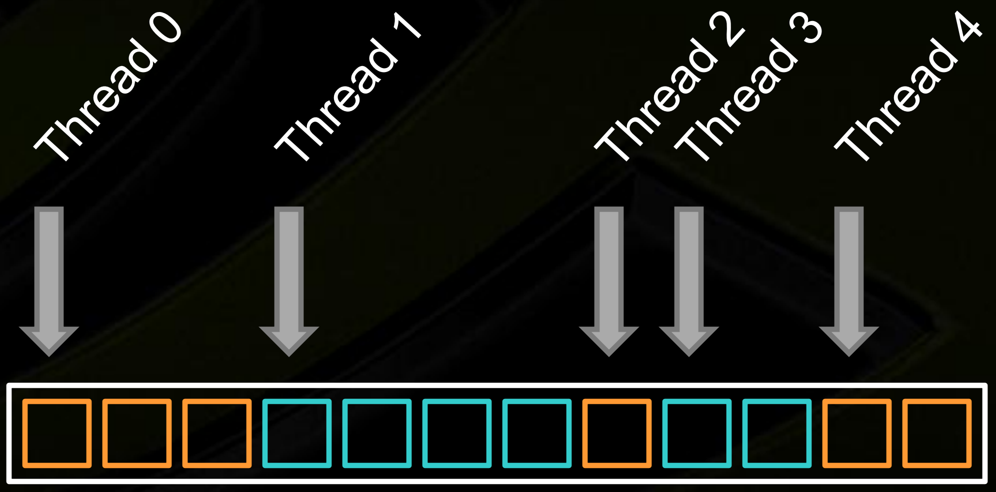
```
void csr_spmv(int     num_rows,
              int   * row_offsets,
              int   * column_indices,
              float * values,
              float * x,
              float * y)
{
  for (int row = 0; row < num_rows; row++)
  {
    int row_begin = row_offsets[row];
    int row_end   = row_offsets[row + 1];

    float sum = 0;

    for (int offset = row_begin; offset < row_end; offset++)
      sum += values[offset] * x[column_indices[offset]];

      y[row] = sum;
  }
}
```

CSR (scalar) kernel

```
__global__
void csr_spmv(int     num_rows,
              int   * row_offsets,
              int   * column_indices,
              float * values,
              float * x,
              float * y)
{
  int row = blockDim.x * blockIdx.x + threadIdx.x;

  if (row < num_rows)
  {
    int row_begin = row_offsets[row];
    int row_end   = row_offsets[row + 1];

    float sum = 0;

    for (int offset = row_begin; offset < row_end; offset++)
      sum += values[offset] * x[column_indices[offset]];

      y[row] = sum;
  }
}
```

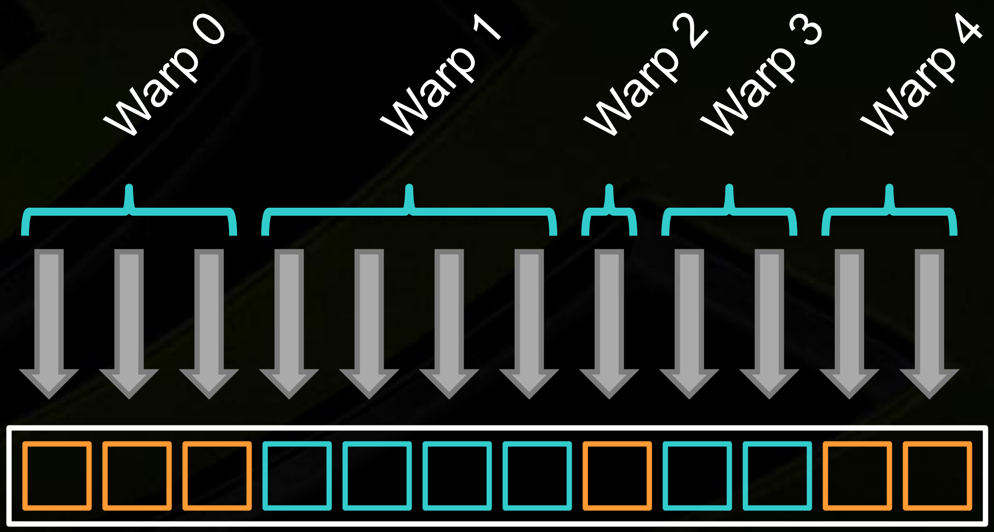
CSR (scalar) kernel

- One thread per row
  - Poor memory coalescing
  - Unaligned memory access



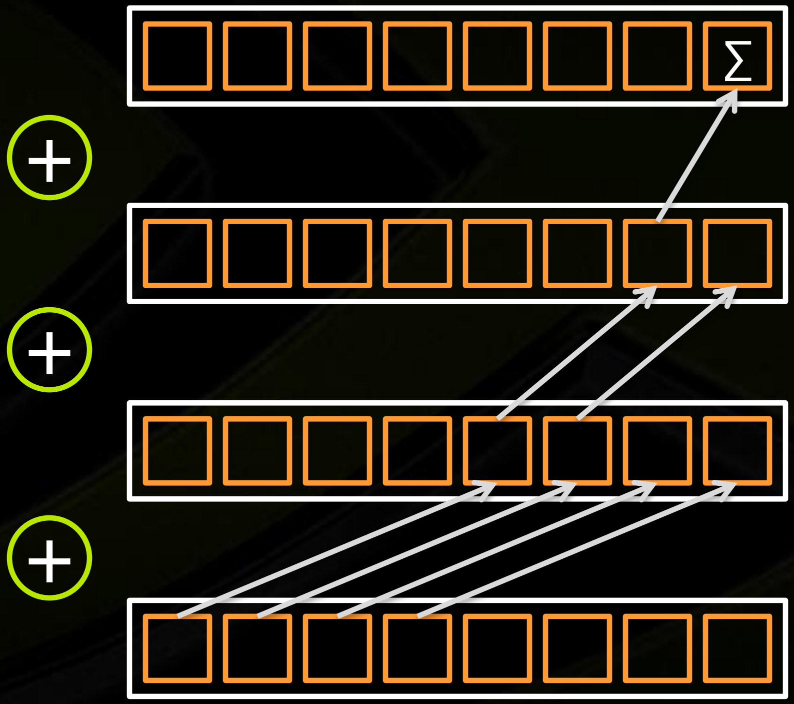
CSR (vector) kernel

- One SIMD vector or *warp* per row
  - Partial memory coalescing
  - Unaligned memory access



CSR (vector) kernel

- Reduce partial sums in shared memory
  - Example: warp of 8 threads



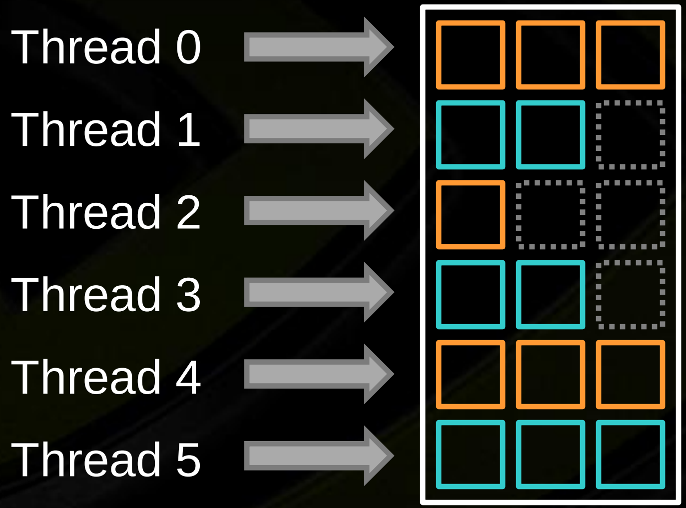
Shared memory

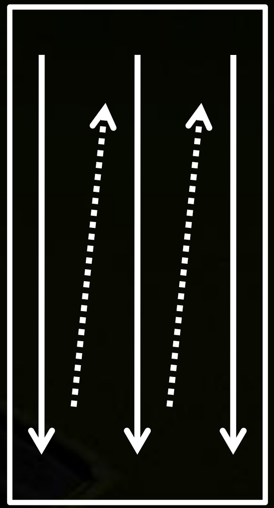
CSR (vector) kernel

```
__global__ void
spmv_csr_vector_kernel()
{
    __shared__ volatile ValueType sdata[VECTORS_PER_BLOCK * THREADS_PER_VECTOR];
    __shared__ volatile IndexType ptrs[VECTORS_PER_BLOCK][2];

        // use two threads to fetch Ap[row] and Ap[row+1]
        if(thread_lane < 2) {  ptrs[vector_lane][thread_lane] = Ap[row + thread_lane]; }

        // implicit synchronization here, no synchronization due to warp synchronous behavior assumed
        const IndexType row_start = ptrs[vector_lane][0];                   //same as: row_start = Ap[row];
        const IndexType row_end   = ptrs[vector_lane][1];                   //same as: row_end   = Ap[row+1];

        // initialize local sum
        ValueType sum = 0;

        // accumulate local sums
        for(IndexType jj = row_start + thread_lane; jj < row_end; jj += THREADS_PER_VECTOR)
             sum += Ax[jj] * x[Aj[jj]];

        // store local sum in shared memory
        sdata[threadIdx.x] = sum;

        // reduce local sums to row sum
       <norrmal warpscan here>

        // first thread writes the result
        if (thread_lane == 0)
            y[row] = sdata[threadIdx.x];
}
```

ELL kernel

- **Full coalescing**

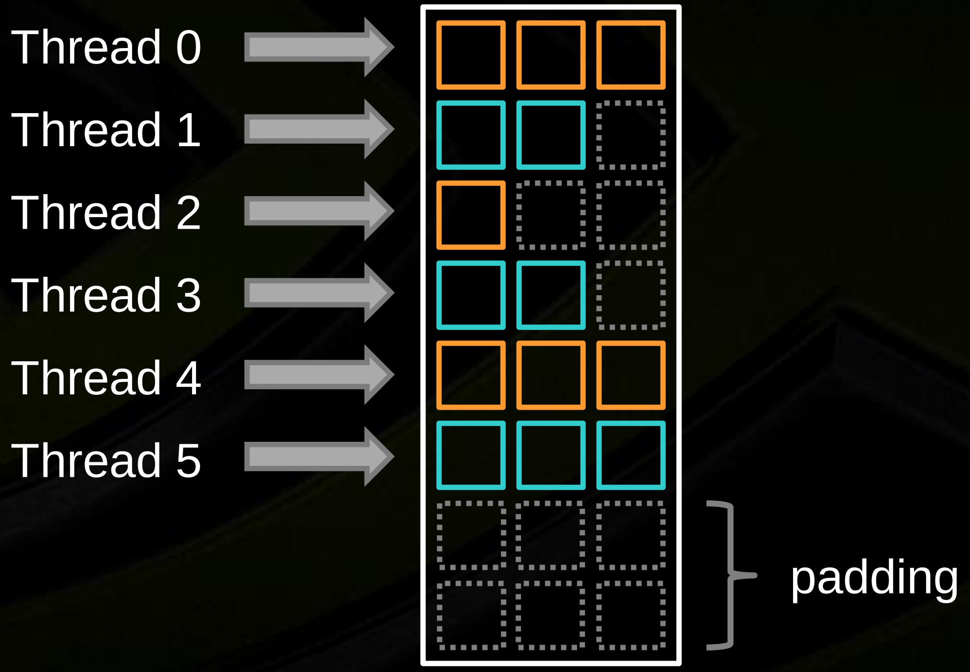




column-major ordering

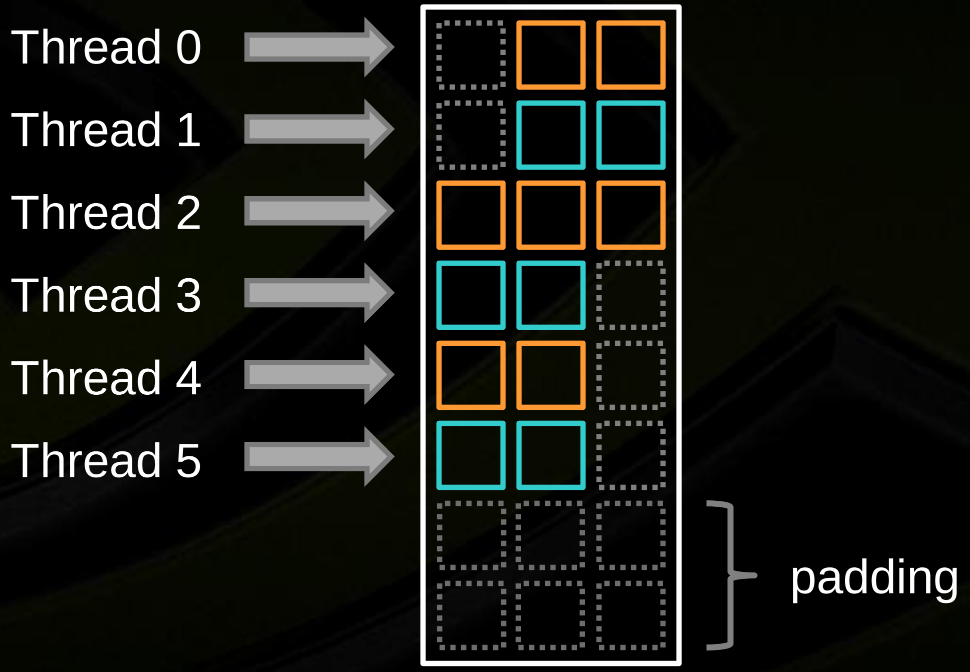
ELL kernel

- **Pad columns for alignment**



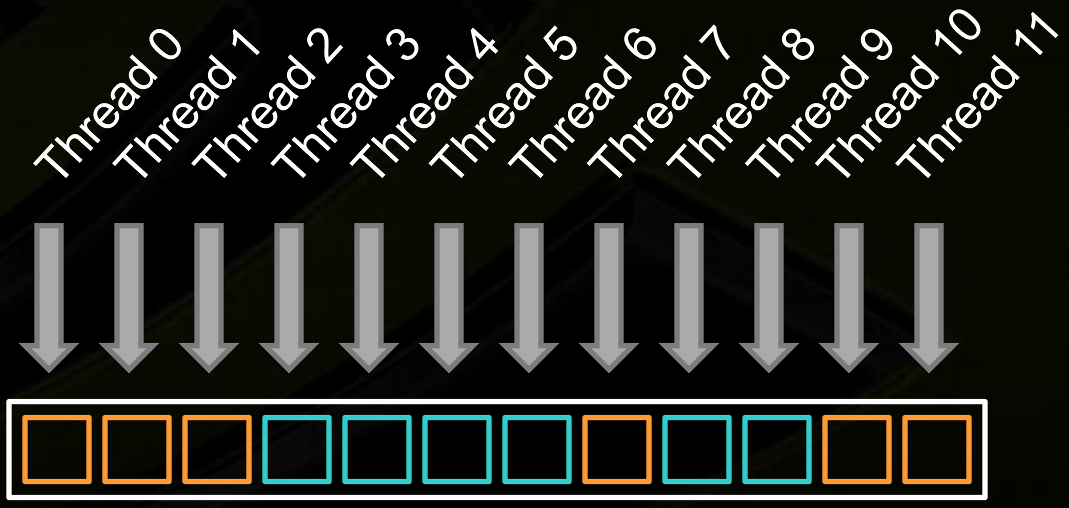
DIA kernel

- **Same as ELL**



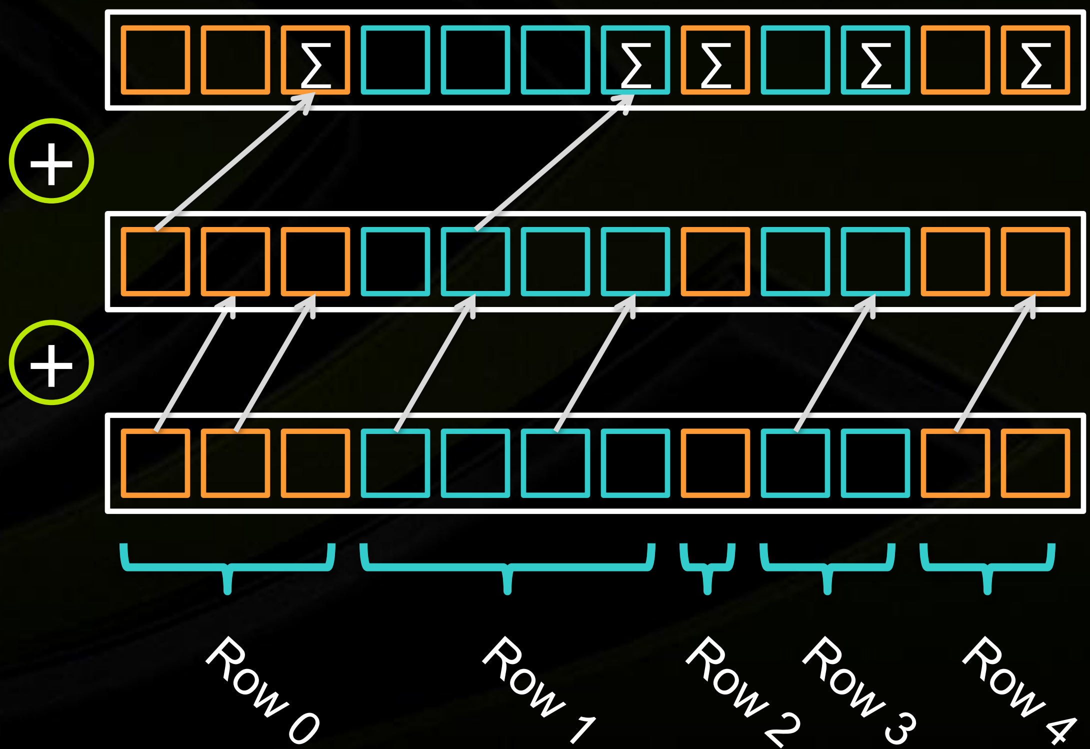
COO kernel

- One thread per nonzero element
  - Fully coalesced



COO kernel

- Store `i` and `A(i,j) * x(j)` in shared memory
  - Compute row sums using *segmented* reduction



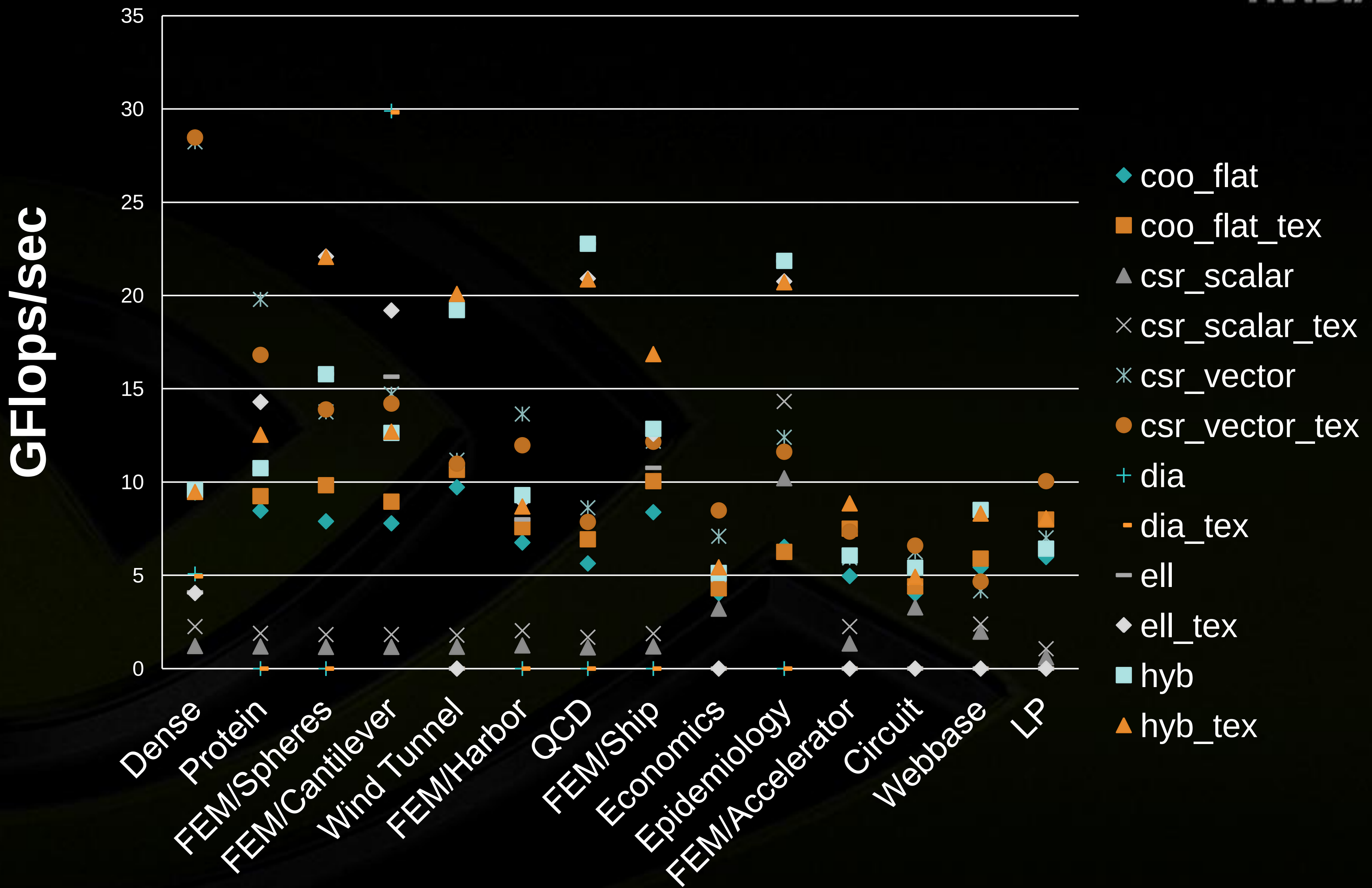
Memory Coalescing Summary

- Full Coalescing
  - DIA, ELL, and COO

- Partial Coalescing
  - CSR (vector)
  - Efficiency depends on row length

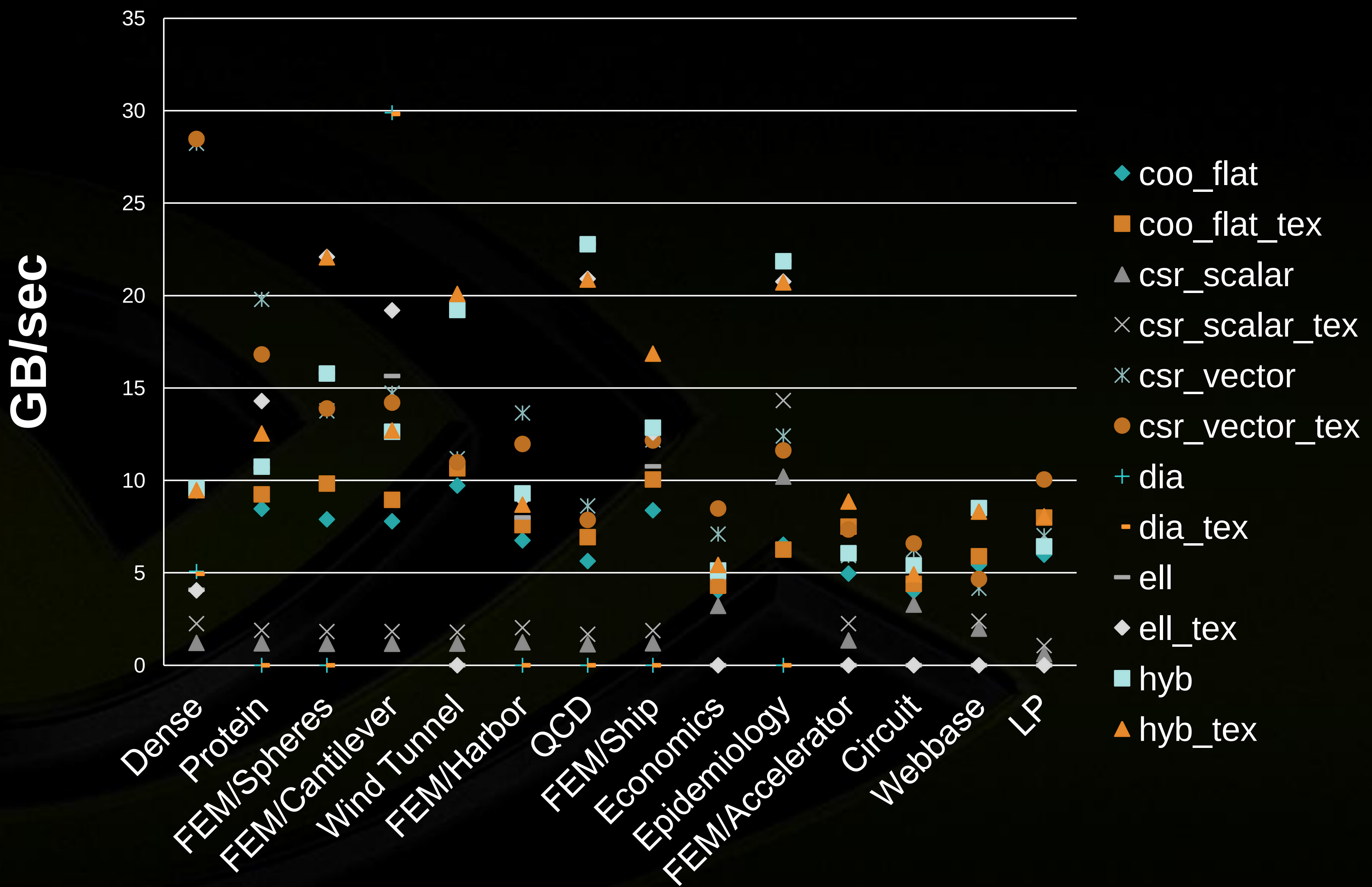
- Little Coalescing
  - CSR (scalar)

Performance Comparison of Formats



Data collected on a C2075 by Steve Dalton with latest CUSP

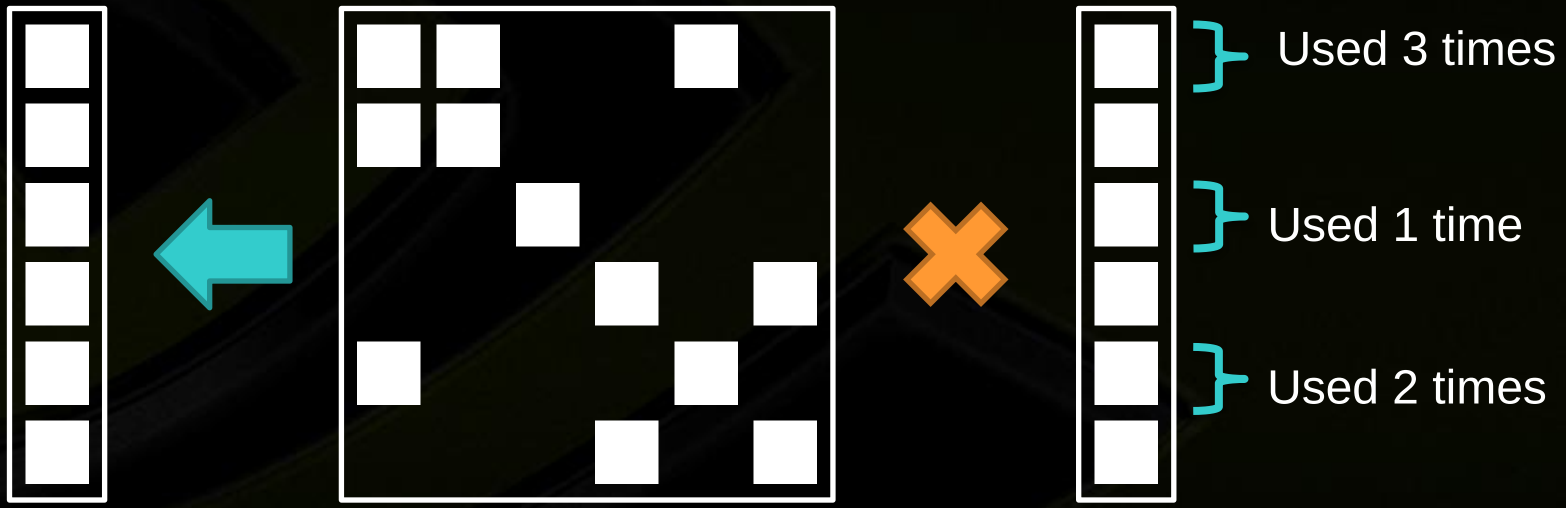
Performance Comparison of Formats



Data collected on a C2075 by Steve Dalton with latest CUSP

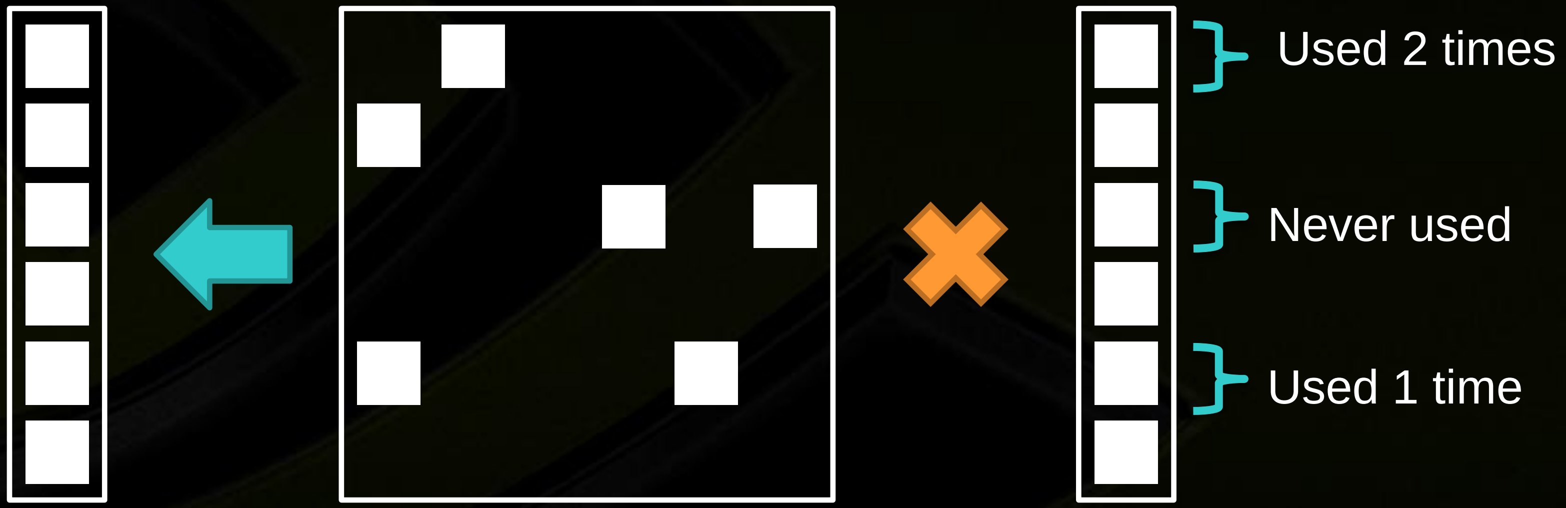
Caching

- **Repeated accesses to source vector**



Caching

- **Effectiveness depends on matrix structure**

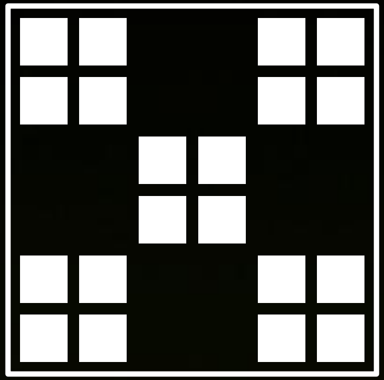


Caching

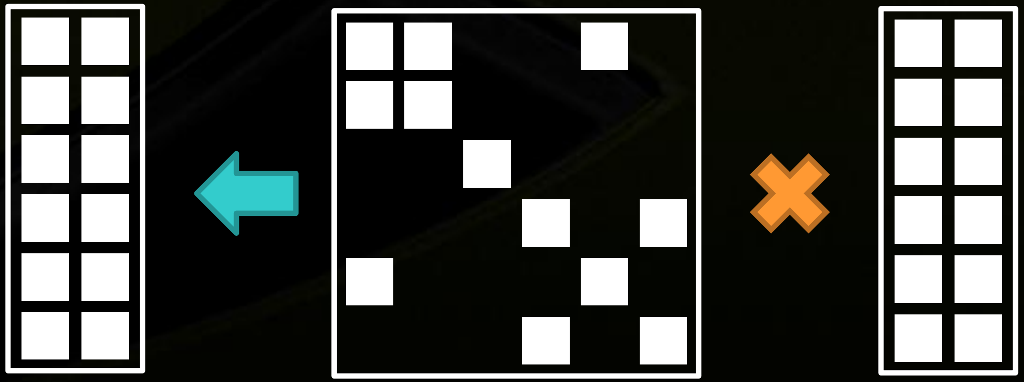
- Fermi architecture has L1 cache
  - No effort needed
- Earlier architectures have texture cache
  - Often worth ~30% improvement
- Software-managed cache
  - Preload into shared memory
- Effectiveness depends on matrix structure

Other Techniques

- Block Formats
  - Reduce index overhead



- Multiple Vectors
  - Reuse matrix data



Performance Considerations

- Use memory bandwidth efficiently
  - Memory coalescing
- Expose lots of fine-grained parallelism
  - Need thousands of threads
- Find opportunities for reuse
  - Make use of caching

Generic Parallel Algorithms for
Sparse Matrix and Graph Computations

CUSP

Example Usage

```
#include <cusp/coo_matrix.h>
#include <cusp/array1d.h>
#include <cusp/multiply.h>

int main(void)
{
    size_t M   = 10;
    size_t N   = 15;
    size_t NNZ = 43;

    // allocate 10x15 COO matrix and vectors
    cusp::coo_matrix<int, float, cusp::device_memory> A(M, N, NNZ);
    cusp::array1d<float, cusp::device_memory> x(N);
    cusp::array1d<float, cusp::device_memory> y(M);

    // initialize A and x
    ...

    // compute matrix-vector product y = A * x
    cusp::multiply(A, x, y);

    return 0;
}
```

Algorithms

- Multiply
  - Sparse Matrix * Vector
  - Sparse Matrix * Sparse Matrix
- Level 1 BLAS
- Transpose
- Maximal Independent Sets
- More to come

Sparse Matrix Containers

```
#include <cusp/coo_matrix.h>

int main(void)
{
    // allocate storage for (4,3) matrix with 6 nonzeros
    cusp::coo_matrix<int, float, cusp::host_memory> A(4,3,6);

    // initialize matrix entries on host
    A.row_indices[0] = 0; A.column_indices[0] = 0; A.values[0] = 10.0f;
    A.row_indices[1] = 0; A.column_indices[1] = 2; A.values[1] = 20.0f;
    A.row_indices[2] = 2; A.column_indices[2] = 2; A.values[2] = 30.0f;
    A.row_indices[3] = 3; A.column_indices[3] = 0; A.values[3] = 40.0f;
    A.row_indices[4] = 3; A.column_indices[4] = 1; A.values[4] = 50.0f;
    A.row_indices[5] = 3; A.column_indices[5] = 2; A.values[5] = 60.0f;

    // A now represents the following matrix
    //     [10  0 20]
    //     [ 0  0  0]
    //     [ 0  0 30]
    //     [40 50 60]

    return 0;
}
```

Sparse Matrix Containers

- COO – Coordinate format
- CSR – Compressed Sparse Row Format
- DIA – Diagonal Format
- ELL – ELLPACK Format
- HYB – Hybrid ELL + COO Format

TEXTURE

Texture Use

```
texture<float, 1, cudaReadModeElementType> t_foo;

__global__ bar_kernel(float * d_bar)
{
      int index = …
      float fromTex = tex1D(t_foo,
index);
      float fromArray = d_bar[index];
}
```

Texture Binding

```
//create cuda array
    cudaChannelFormatDesc channelDesc =
cudaCreateChannelDesc(32, 0, 0, 0, cudaChannelFormatKindFloat);
     cudaMallocArray( &d_foo_array, &channelDesc, size, 1 );
    cudaMemcpyToArray( d_foo_array, 0, 0, d_foo, size * sizeof
(float), cudaMemcpyDeviceToDevice);

    // Bind the array to the texture
    cudaBindTextureToArray( t_foo, d_some_array, channelDesc);
```

CSR (vector) kernel with texture

```
texture<float, 1, cudaReadModeElementType> t_x;



    __global__ void
    spmv_csr_vector_tex_kernel()
    {
      …
         // initialize local sum
         ValueType sum = 0;

         // accumulate local sums
         for(IndexType jj = row_start + thread_lane; jj < row_end; jj +=
    THREADS_PER_VECTOR)
             sum += Ax[jj] * tex1D(t_x, Aj[jj]); // access to x is a sparse gather

         // store local sum in shared memory
         sdata[threadIdx.x] = sum;


        …
    }
```

References

Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors
Nathan Bell and Michael Garland
Proceedings of Supercomputing '09

Efficient Sparse Matrix-Vector Multiplication on CUDA
Nathan Bell and Michael Garland
NVIDIA Technical Report NVR-2008-004, December 2008

Model-driven Autotuning of Sparse Matrix-Vector Multiply on GPUs
Jee Whan Choi, Amik Singh and Richard W. Vuduc
Proceedings of Principles and Practice of Parallel Programming (PPoPP) 2010

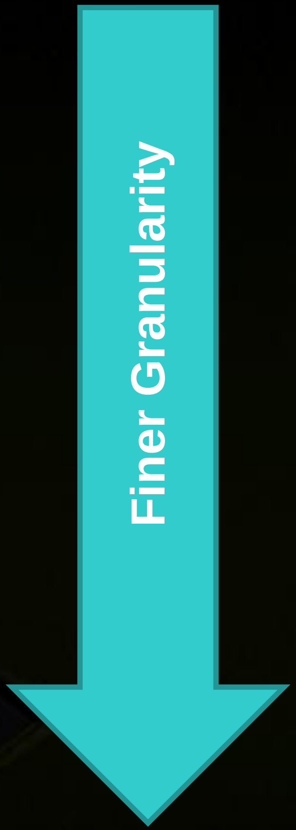
Backup Slides

Performance Considerations

- Use memory bandwidth efficiently
  - Memory coalescing
- Expose lots of fine-grained parallelism
  - Need thousands of threads
- Find opportunities for reuse
  - Make use of caching

Exposing Parallelism

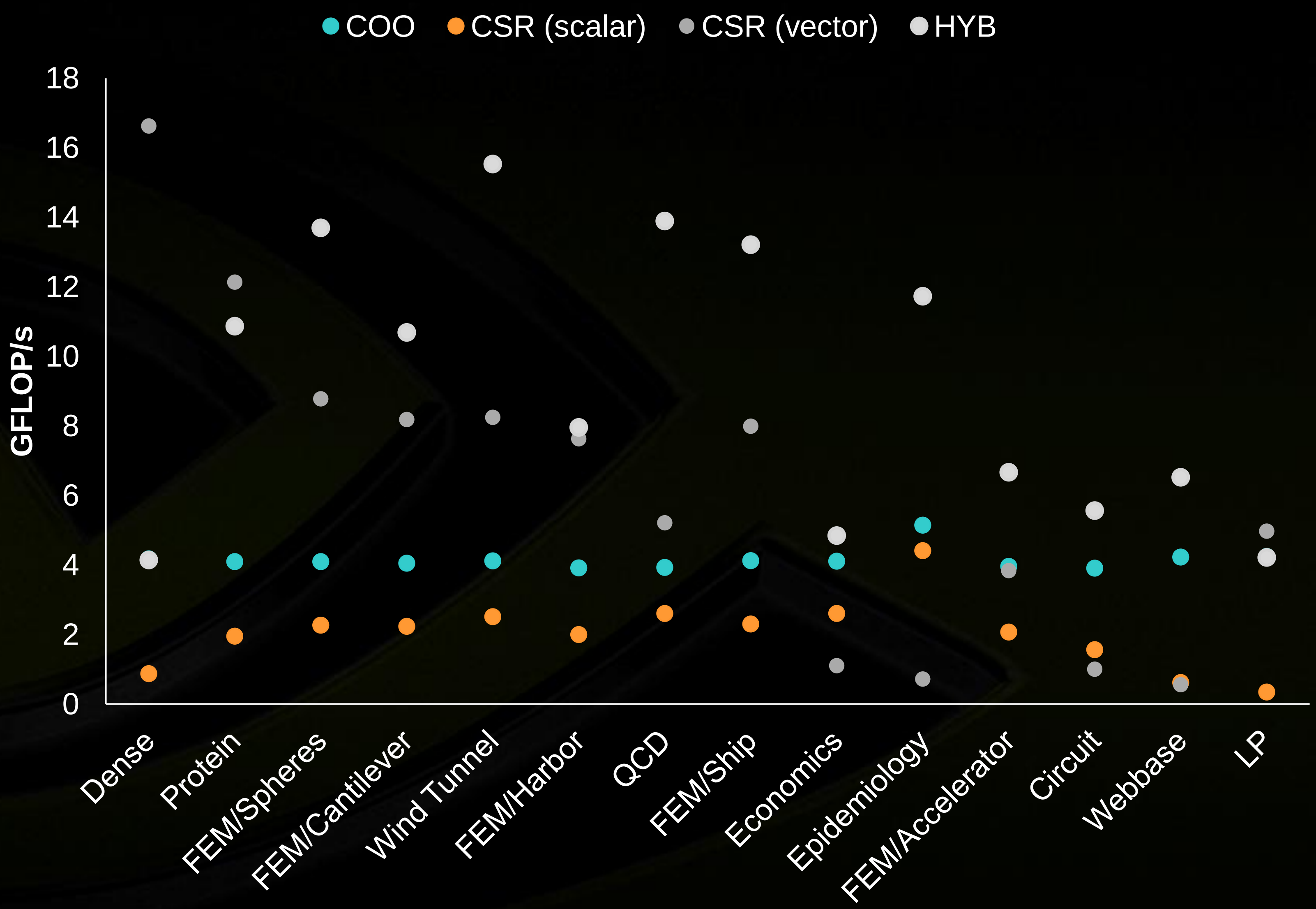
- DIA, ELL & CSR (scalar)
    - One thread per row
- CSR (vector)
    - One warp per row
- COO
    - One thread per nonzero



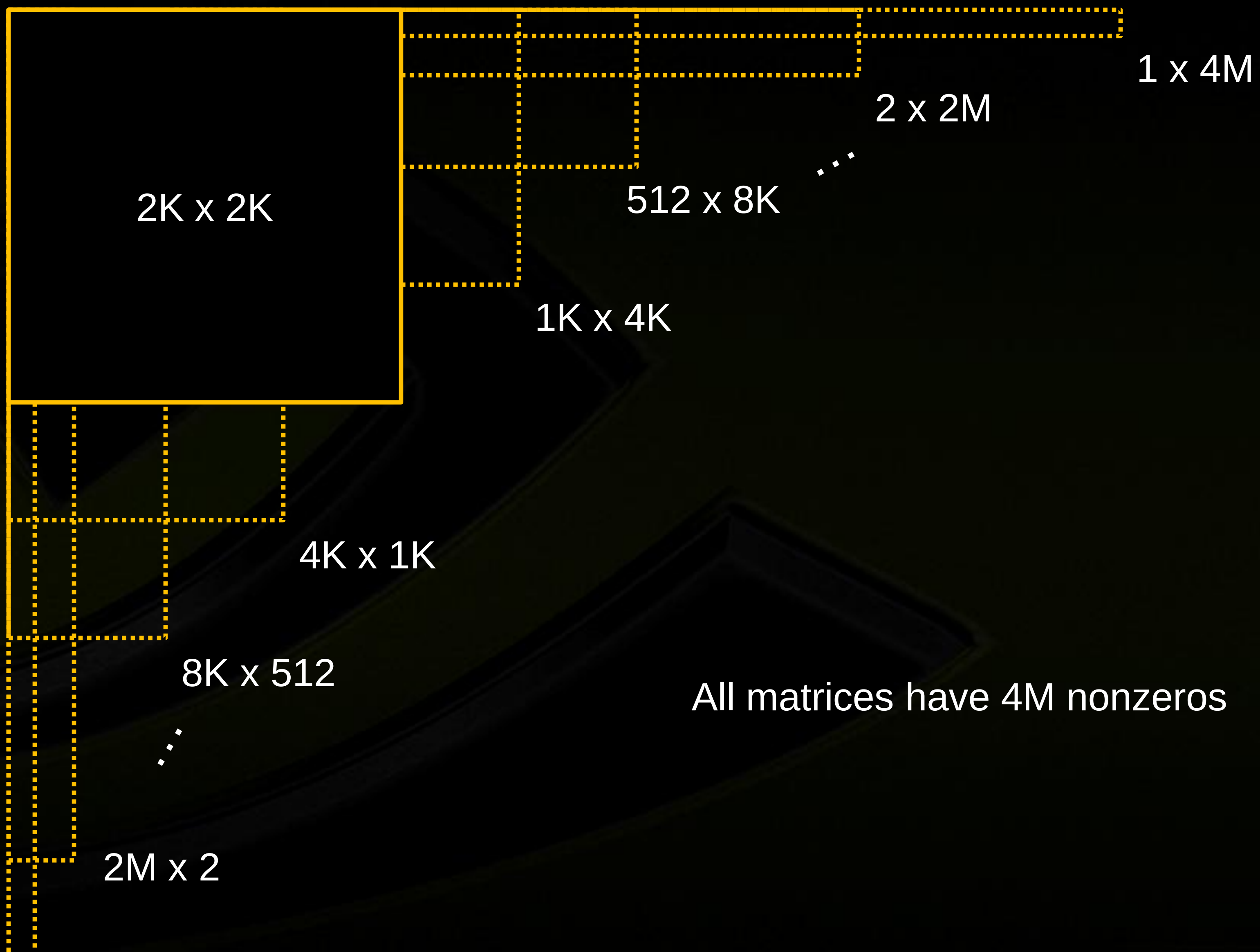
Performance Considerations

- **Use memory bandwidth efficiently**
  - **Memory coalescing**
- **Expose lots of fine-grained parallelism**
  - **Need thousands of threads**
- **Find opportunities for reuse**
  - **Make use of caching**

Performance Comparison of Formats

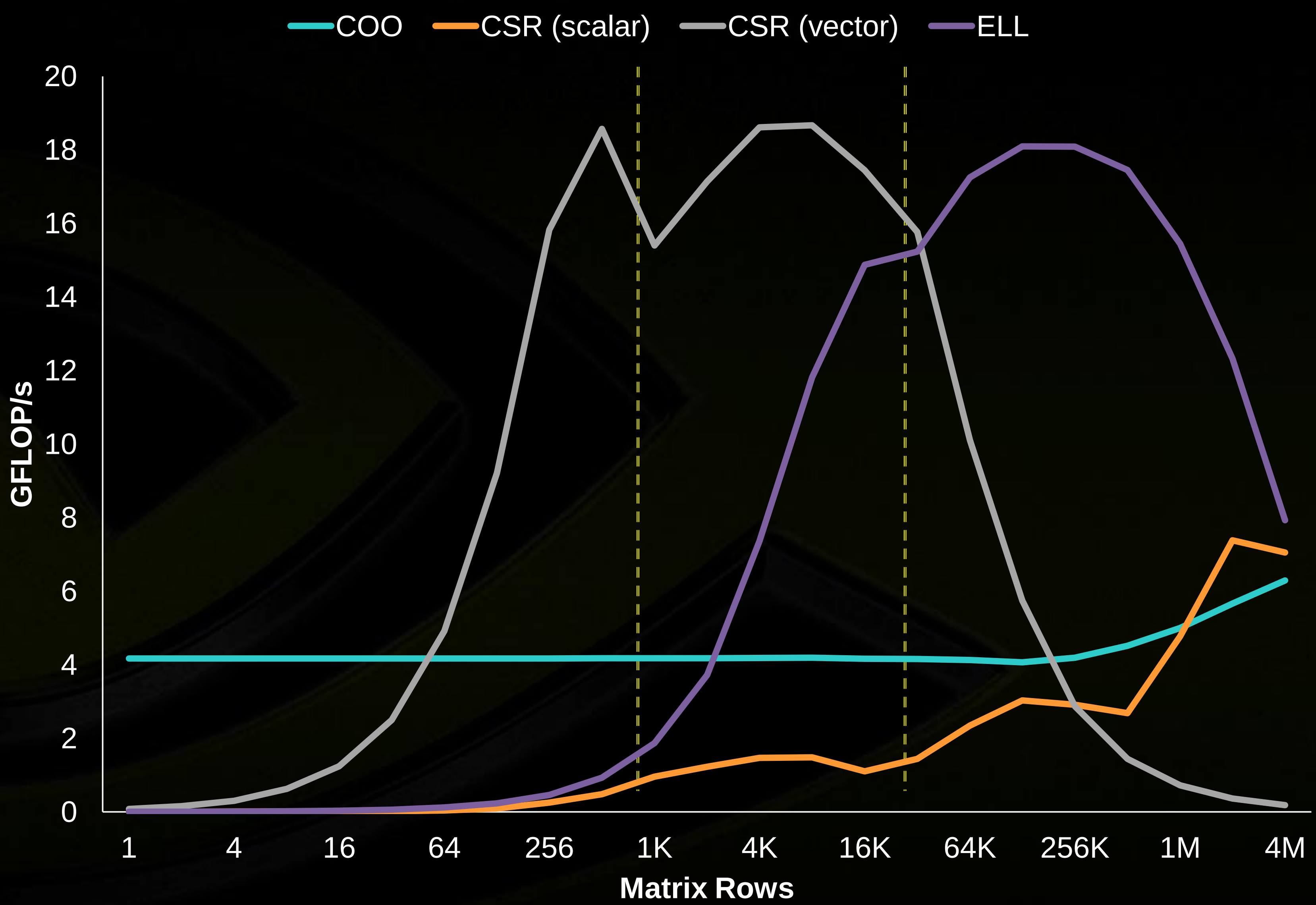


Exposing Parallelism



All matrices have 4M nonzeros

Exposing Parallelism



GPU: GeForce GTX 285

Exposing Parallelism

- One thread per row
  - ELL, DIA, and CSR (scalar) kernel
  - Generally good enough (>20K rows is common)
- One warp per row
  - CSR (vector) kernel
  - Fewer rows is sufficient (>256)
- One thread per entry
  - COO kernel
  - Insensitive to matrix shape