Leveraging Matrix Block Structure In Sparse Matrix-Vector Multiplication

Steve Rennich
Nvidia
Developer Technology - Compute
Block Sparse Matrix Vector Multiplication

- Sparse Matrix-Vector Multiplication (SpMV)
  - \( y = A \times x \)
  - Iterative methods: 50% - 75% of solution time
  - Bandwidth bound

- Many matrices have a 'natural' block structure
  - those arising in FEA

- Block SpMV (BSpMV) algorithm for the GPU
  - Leverage block structure to reduce communications
Block Matrix Structure

- As would arise in FEA
  - $A$ is ‘naturally’ blocked
  - Variable block sizes, aspect ratios
    - Determined by # of unknowns, connectivity
    - Typical sizes 1 -> 8
  - Blocks are dense

- Not ‘prescribed’ blocks
Block Matrix Structure

- As would arise in FEA
  - A is ‘naturally’ blocked
  - Variable block sizes, aspect ratios
    - Determined by # of unknowns, connectivity
    - Typical sizes 1 -> 8
  - Blocks are dense

- Not ‘prescribed’ blocks
Block Matrix Structure

- As would arise in FEA
  - \( A \) is ‘naturally’ blocked
  - Variable block sizes, aspect ratios
    - Determined by # of unknowns, connectivity
    - Typical sizes 1 -> 8
  - Blocks are dense

- Not ‘prescribed’ blocks
BSpMV Bandwidth Analysis

- M2090 ECC OFF: (177 GB/s) 142 GB/s
  - Double Precision

- Standard approach
  \[
  8 + 4 + 8 = 20 \text{ bytes} \rightarrow 14.2 \text{ Gflops/s}
  \]

- Block-based
  \[
  8 + \frac{4}{nr*nc} + \frac{8}{nr} = 11.3 \text{ bytes} \rightarrow 25.1 \text{ Gflops/s}
  \]
  - 6 x 6 blocks
  - Limit M2090: 30 Gflops/s
  - Limit Westmere socket: 25 GB/s 6.25 Gflops/s

\[ \text{Gflops} = 2 \times \text{BW} / \text{bytes} \]
BSpMV Algorithm Features

- **Coherent Execution & Memory Access**
  - ELLPACK-like data structure
  - Reorder rows - Load balancing
  - Decompose A matrix & use separate kernels for each block column size

- **Sufficient Parallelism**
  - One thread per row

- **Minimal Data Transfer**
  - Leverage block structure to minimize column index data
  - Use texture cache for x data
  - Row and block column size are implicit
Data Structure

- Separate array for each block column size
  - Eliminates divergence
Data Structure

- For each block column size:

  - Original
  - Sort
  - ELLPACK: 64 rows, <3% waste
  - Warps: load balanced warps
Standard Caching of x in Texture

thread 0

\[
\begin{align*}
\text{ld } x_0 \\
y_0 &\leftarrow A_{00} \times x_0 \\
\text{ld } x_1 \\
y_0 &\leftarrow A_{01} \times x_1 \\
\text{ld } x_2 \\
y_0 &\leftarrow A_{02} \times x_2
\end{align*}
\]

thread 1

\[
\begin{align*}
\text{ld } x_0 \\
y_1 &\leftarrow A_{10} \times x_0 \\
\text{ld } x_1 \\
y_1 &\leftarrow A_{11} \times x_1 \\
\text{ld } x_2 \\
y_1 &\leftarrow A_{12} \times x_2
\end{align*}
\]

thread 2

\[
\begin{align*}
\text{ld } x_0 \\
y_2 &\leftarrow A_{20} \times x_0 \\
\text{ld } x_1 \\
y_2 &\leftarrow A_{21} \times x_1 \\
\text{ld } x_2 \\
y_2 &\leftarrow A_{22} \times x_2
\end{align*}
\]
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`

Threads idle waiting for $x_0$
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
```

$x_0$ broadcast to all threads

```
y = A * x
```

thread 0

```
thread 1

thread 2
```

TEX

```
x0
x1
x2
x3
```

L2

```
x0
x1
x2
x3
```

x in GMEM

```
x0
x1
x2
x3
```
Standard Caching of $x$ in Texture

**Thread 0**
- $ld\ x0$
- $y0 += A00 \times x0$
- $ld\ x1$
- $y0 += A01 \times x1$
- $ld\ x2$
- $y0 += A02 \times x2$

**Thread 1**
- $ld\ x0$
- $y1 += A10 \times x0$
- $ld\ x1$
- $y1 += A11 \times x1$
- $ld\ x2$
- $y1 += A12 \times x2$

**Thread 2**
- $ld\ x0$
- $y2 += A20 \times x0$
- $ld\ x1$
- $y2 += A21 \times x1$
- $ld\ x2$
- $y2 += A22 \times x2$

**Texture Access**
- Single ‘quad’ texture request for $x_1$
- Potentially evicted!

**Cache Access**
- $x_0, x_1, x_2, x_3$
- Loaded multiple times from GMEM

**Equation**
- $y = A \times x$

**GPU Technology Conference**
Improved Caching of $x$ in Texture

- Thread 0:
  - `ld x0`
  - `ld x1`
  - `ld x2`
  - `y0 += A00 * x0`
  - `y0 += A01 * x1`
  - `y0 += A02 * x2`

- Thread 1:
  - `ld x0`
  - `ld x1`
  - `ld x2`
  - `y1 += A10 * x0`
  - `y1 += A11 * x1`
  - `y1 += A12 * x2`

- Thread 2:
  - `ld x0`
  - `ld x1`
  - `ld x2`
  - `y2 += A20 * x0`
  - `y2 += A21 * x1`
  - `y2 += A22 * x2`

Single "quad" texture request for $x_0$
Improved Caching of x in Texture

- **Thread 0**
  - `ld x0`
  - `ld x1`
  - `ld x2`
  - `y0 += A00 * x0`
  - `y0 += A01 * x1`
  - `y0 += A02 * x2`

- **Thread 1**
  - `ld x0`
  - `ld x1`
  - `ld x2`
  - `y1 += A10 * x0`
  - `y1 += A11 * x1`
  - `y1 += A12 * x2`

- **Thread 2**
  - `ld x0`
  - `ld x1`
  - `ld x2`
  - `y2 += A20 * x0`
  - `y2 += A21 * x1`
  - `y2 += A22 * x2`

**Texture Requests for x_1, x_2**

**L2 Miss in L2**

**y = A * x**

**Threads**
- Thread 0
- Thread 1
- Thread 2
Improved Caching of x in Texture

Thread 0
- ld x0
- ld x1
- ld x2
- y0 += A00 * x0
- y0 += A01 * x1
- y0 += A02 * x2

Thread 1
- ld x0
- ld x1
- ld x2
- y1 += A10 * x0
- y1 += A11 * x1
- y1 += A12 * x2

Thread 2
- ld x0
- ld x1
- ld x2
- y2 += A20 * x0
- y2 += A21 * x1
- y2 += A22 * x2

x0, x1, x2 from single GMEM access

y = A * x

TEX
  x0
  x1
  x2
  x3

L2
  x0
  x1
  x2
  x3

x in GMEM

x0, x1, x2 from single GMEM access
Improved 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
- Leveraging block structure
- No use of SMEM required
Implementation Details

- X vector is 'padded' to block column size by 'block row'
  - 'Blocks' padded to multiple of 32 Bytes
  - Constant - indexable by block ID
  - Requires a ‘swizzle’ for every block column size

- Reverse permutation of rows and summation of intermediate results are all done on the GPU
  - Integrated with BSpMV kernel
template <unsigned char colSize>
__global__ void BSpMV ( ... )
{
    // Initializations
    ...

    // Loop over nonzero blocks in this row
    for ( uint iblock=0; iblock<nBlocks; ++iblock ) {
        // Get column start
        col = padding[colSize] * nzBlocks[blockStride*iblock+ibRow ];

        // Loop over block column
        for ( int i=0; i<colSize; i++ ) {
            texval[i] = tex1Dfetch (tex_x, col++);
            y += A[ap] * __hiloint2double ( texval[i].y, texval[i].x );
            ap += stride;
        }
    }

    ...
}
Kernel

template <unsigned char colSize>
__global__ void BSpmV ( ... )
{
    // Initializations
    ...

    // Loop over nonzero blocks in this row
    for ( uint iblock=0; iblock<nBlocks; ++iblock ) {

        // Get column start
        col = padding[colSize] * nzBlocks[ blockStride*iblock+ibRow ];

        // Loop over block column
        for ( int i=0; i<colSize; i++ ) {
            texval[i] = tex1Dfetch ( tex_x, col++);
            y += A[ap] * __hiloint2double ( texval[i].y, texval[i].x );
            ap+= stride;
        }
    }
    ...
}

nvcc does the unrolling and reordering
Performance

  - Chapter 4: Williams, Bell, Choi, Garland, Oliker, Vuduc, “Sparse Matrix-Vector Multiplication on Multicore and Accelerators”

- Florida Sparse Matrix Collection
  - Williams Group

- are the BSpMV results
  - Computed on M2090 and scaled by 159 GBs /177 GBs

- Many of the Williams matrices are not blocked
Performance for BSpMV

FSMC / Williams

FSMC / real, square, >250k

“structural problem”

Industry

Gflops/s (M2090)

0 5 10 15 20 25 30 35

Protein  FEM/sphere  Wind Turbine  FEM/Ship  EconomicS  FEM/accelerator... al_shell/0  Geo_1438  Serena  lidor  alldkw_1  Emillia_923  Apache2  Fault_619  al_shell/7  Inline_1  al_0_k101  msdor  F1  Lin
industry 1  industry 2  industry 3  industry 4  industry 5  industry 6  industry 7  industry 8  industry 9  industry 10

tesla M2090
Performance for BSpMV

FSMC / Williams

FSMC / real, square, >250k
“structural problem”

Industry

Tesla M2090
Practical Considerations

- GPU performance is dependent on block size
  - Larger is better - Prefer multiples of 4
  - Performance is reduced for 1x1 blocks (thermal analysis) (~10 Gflops/s)

- GPU-friendly data structure is very important for performance

- Datastructure translation costs ~40 iterations on CPU
  - Very roughly 130 CPU iterations for 2x speedup
Summary

- Block structure of sparse matrices can be effectively leveraged to improve SpMV performance
  - Demonstrated for structural analysis / FEA matrices
  - Performance approaches limit of A data transfer

- Limitations
  - Data structure translation ~ 130 iterations

- Future
  - Faster data structure translation, multi-gpu, hybrid-computing, ...
Thank you