L11: Sparse Linear Algebra on GPUs

Sparse Linear Algebra

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i,$$

where $a_1 = 0$ and $c_n = 0$. In matrix form, this system is written as

$$\begin{bmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & \ddots & \\ & & \ddots & \ddots & c_{n-1} \\ 0 & & & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_n \end{bmatrix}.$$

http://en.wikipedia.org/wiki/Tridiagonal_matrix_algorithm

GPU Challenges

- Computation partitioning?
- Memory access patterns?
- Parallel reduction

BUT, good news is that sparse linear algebra performs TERRIBLY on conventional architectures, so poor baseline leads to improvements!

Some common representations

DIA

$$A = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

$$\text{data} = \begin{bmatrix} * & 1 & 7 \\ * & 2 & 8 \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix}$$

offsets = [-2 0 1]

Stores elements along a set of diagonals.

data	1	2	5	6	7	8	3	4	*	*	9	*
indices	0	1	0	1	1	2	2	3	*	*	3	*
Iteration 0	0	1	2	3								
Iteration 1					0	1	2	3				
Iteration 2									0	1	2	3

. Each thread iterates over the diagonals
+ Avoids the need to store row/column indices
+ Guarantees coalesced access
- Can be wasteful if lots of padding required

ELL

$$\text{data} = \begin{bmatrix} 1 & 7 & * \\ 2 & 8 & * \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix} \quad \text{indices} = \begin{bmatrix} 0 & 1 & * \\ 1 & 2 & * \\ 0 & 2 & 3 \\ 1 & 3 & * \end{bmatrix}$$

Stores a set of K elements per row and pad as needed. Best suited when number non-zeros roughly consistent across rows.

-Efficiency rapidly degrades when the number of nonzeros per matrix row varies

data	*	*	5	6	1	2	3	4	7	8	9	*
Iteration 0			2	3								
Iteration 1					0	1	2	3				
Iteration 2									0	1	2	

Some common representations

$$A = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

CSR

ptr = [0 2 4 7 9]
indices = [0 1 1 2 0 2 3 1 3]
data = [1 7 2 8 5 3 9 6 4]

Compressed Sparse Row (CSR): Store only nonzero elements, with "ptr" to beginning of each row and "indices" representing column.

COO

row = [0 0 1 1 2 2 2 3 3]
indices = [0 1 1 2 0 2 3 1 3]
data = [1 7 2 8 5 3 9 6 4]

Store nonzero elements and their corresponding "coordinates".

+ Performance is largely insensitive to irregularity in the underlying data structure

row	0	0	1	1	2	2	2	3	3
indices	0	1	1	2	0	2	3	1	3
data	1	7	2	8	5	3	9	6	4

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

CSR Example

```
for (j=0; j<nr; j++) {
   for (k = ptr[j]; k<ptr[j+1]-1; k++)
      t[j] = t[j] + data[k] * x[indices[k]];
```


Summary of Representation and Implementation

Kernel	Granularity	Coalescing	Bytes/Flop 32-bit	Bytes/Flop 64-bit
DIA	thread : row	full	4	8
ELL	thread : row	full	6	10
CSR(s)	thread : row	rare	6	10
CSR(v)	warp : row	partial	6	10
COO	thread : nonz	full	8	12
HYB	thread : row	full	6	10

Table 1 from Bell/Garland: Summary of SpMV kernel properties.

Other Representation Examples

- Blocked CSR
  - Represent non-zeros as a set of blocks, usually of fixed size
  - Within each block, treat as dense and pad block with zeros
  - Block looks like standard matvec
  - So performs well for blocks of decent size

$$A = \begin{pmatrix} a_{00} & a_{01} & 0 & 0 & a_{04} & a_{05} \\ a_{10} & a_{11} & 0 & 0 & a_{14} & a_{15} \\ 0 & 0 & a_{22} & 0 & a_{24} & a_{25} \\ 0 & 0 & a_{32} & a_{33} & a_{34} & a_{35} \end{pmatrix}$$

$$\texttt{b\_row\_ptr} = \begin{pmatrix} 0 & 2 & 4 \end{pmatrix}, \texttt{b\_col\_idx} = \begin{pmatrix} 0 & 4 & 2 & 4 \end{pmatrix}$$

$$\texttt{b\_value} = \begin{pmatrix} a_{00} & a_{01} & a_{10} & a_{11} & a_{04} & a_{05} & a_{14} & a_{15} & a_{22} & 0 & a_{32} & a_{33} & a_{24} & a_{25} & a_{34} & a_{35} \end{pmatrix}$$

Richard Vuduc, Attila Gyulassy, James W. Demmel, and Katherine A. Yelick. 2003. Memory hie optimizations and performance bounds for sparse A^TAx. In Proceedings of the 2003 internation conference on Computational science: PartIII (ICCS'03)

- Hybrid ELL/COO
  - Exploits these:
    - ELLPACK format is well-suited to vector and SIMD, its efficiency rapidly degrades when the number of nonzeros per matrix row varies
    - Storage efficiency of the COO format is invariant to the distribution of nonzeros per row
  - Find a "K" value that works for most of matrix
  - Use COO for rows with more nonzeros (or even significantly fewer)

Stencil Example

What is a 3-point stencil? 5-point stencil? 7-point? 9-point? 27-point?

Examples:

a[i] = 2*b[i] – (b[i-1] + b[i+1]);
[-1 2 -1]

a[i][j] = 4*b[i][j] -(b[i-1][j] + b[i+1][j] + b[i][j-1] + b[i][j+1]);
[0 -1 0]
[-1 4 -1]
[0 -1 0]

Stencil Result (structured matrices)

See Figures 11 and 12, Bell and Garland

Unstructured Matrices

See Figures 13 and 14

Note that graphs can also be represented as sparse matrices.
What is an adjacency matrix?

PPoPP paper

- What if you customize the representation to the problem?
- Additional global data structure modifications (like blocked representation)?
- Strategy
  - Apply models and autotuning to identify best solution for each application

Summary of Results

BELLPACK (blocked ELLPACK) achieves up to 29 Gflop/s in SP and 15.7 Gflop/s in DP

Up to 1.8x and 1.5x improvement over Bell and Garland.

This Lecture

- Exposure to the issues in a sparse matrix vector computation on GPUs
- A set of implementations and their expected performance
- A little on how to improve performance through application-specific knowledge and customization of sparse matrix representation