L11: Sparse Linear Algebra on GPUs

Sparse Linear Algebra

\[ \begin{bmatrix} a_{11} & a_{12} & 0 & \cdots & 0 \\ a_{21} & a_{22} & a_{23} & \cdots & 0 \\ 0 & a_{32} & a_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_{n,n-1} \\ \end{bmatrix} \]


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**

Stores elements along a set of diagonals.

- Efficiency rapidly degrades when the number of nonzeros per matrix row varies
- Can be wasteful if lots of padding required

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

Data: [1 7 0 0 2 8 0 5 0 3 9 0 6 0 4]

Indices: [1 3 2 4]

Offsets: [-2 0 1]

**ELL**

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

\[
\begin{bmatrix} 
17 & 28 & 539 & 64 \\
12 & 23 & 34 & 56 \\
13 & 24 & 35 & 56 \\
\end{bmatrix}
\]

Data: [1 2 3 1 2 3 4 5 6 7 8 9 4 5 6 7 8 9]

Indices: [0 1 2 1 2 3]

Horizons: [0 0 2 0 2 0]
Some common representations

\[
A = \begin{bmatrix}
1700 & 0 & 0 & 0 \\
0 & 280 & 0 & 0 \\
0 & 0 & 5039 & 0 \\
0 & 0 & 0 & 604
\end{bmatrix}
\]

CSR

- \(\text{ptr} = [0 \ 2 \ 4 \ 7 \ 9]\)
- \(\text{indices} = [0 \ 1 \ 1 \ 2 \ 0 \ 2 \ 3 \ 1 \ 3]\)
- \(\text{data} = [1 \ 7 \ 2 \ 8 \ 5 \ 3 \ 9 \ 6 \ 4]\)

Compressed Sparse Row (CSR):
Store only nonzero elements, with “\(\text{ptr}\)” to beginning of each row and “\(\text{indices}\)” representing column.

COO

- \(\text{row} = [0 \ 0 \ 1 \ 2 \ 3 \ 3 \ 3]\)
- \(\text{indices} = [0 \ 1 \ 1 \ 2 \ 0 \ 2 \ 3 \ 1 \ 3]\)
- \(\text{data} = [1 \ 7 \ 2 \ 8 \ 5 \ 3 \ 9 \ 6 \ 4]\)

Store nonzero elements and their corresponding “coordinates”.

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

<table>
<thead>
<tr>
<th>Bytes/Flop</th>
<th>Kernel</th>
<th>Granularity</th>
<th>Coalescing</th>
<th>32-bit</th>
<th>64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIA</td>
<td>thread : row</td>
<td>full</td>
<td>4</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>ELL</td>
<td>thread : row</td>
<td>full</td>
<td>6</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>CSR(s)</td>
<td>thread : row</td>
<td>rare</td>
<td>6</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>CSR(v)</td>
<td>warp : row</td>
<td>partial</td>
<td>6</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>COO</td>
<td>thread : nonz</td>
<td>full</td>
<td>8</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>HVB</td>
<td>thread : row</td>
<td>full</td>
<td>6</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

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 matrix
  - So performs well for blocks of decent size

- **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 if 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:
\[ a[i] = 2 \times b[i] - (b[i-1] + b[i+1]); \]
\[ [-1 \quad 2 \quad -1] \]
\[ a[i][j] = 4 \times b[i][j] - (b[i-1][j] + b[i+1][j] + b[i][j-1] + b[i][j+1]); \]
\[ [\begin{array}{ccc} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{array}] \]

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