Sparse Compilation
Motivation for Sparse Codes

- Consider flux, heat, or stresses – interactions are between neighbors.
- Linear equations are sparse. Therefore, matrices are sparse.
Three Sparse Matrix Representations

**CRS**

Indexed access to a row

**CCS**

Indexed access to a column

**Co-ordinate Storage**

Indexed access to neither rows nor columns
### Jagged Diagonal (JDiag)

\[
\begin{pmatrix}
1 & 2 & 3 & 4 \\
1 & 2 & 3 & 4 \\
2 & 3 & 4 & 2 \\
3 & 4 & 2 & 3 \\
4 & 2 & 3 & 4 \\
\end{pmatrix}
\rightarrow
\begin{pmatrix}
1 & 2 & 3 & 4 \\
2 & 3 & 4 & 2 \\
1 & 2 & 3 & 4 \\
3 & 4 & 2 & 3 \\
4 & 2 & 3 & 4 \\
\end{pmatrix}
\rightarrow
\begin{bmatrix}
c & d & e \\
a & b \\
f & g \\
h \\
\end{bmatrix}
\]

- Long “vectors”
- Direction of access is not row or column
BlockSolve (BS)

- Dense submatrices
- Colors $\rightarrow$ cliques $\rightarrow$ inodes.
- Composition of two storage formats.
The effect of sparse storage

<table>
<thead>
<tr>
<th>Name</th>
<th>N</th>
<th>Nzs</th>
<th>Diag.</th>
<th>Coor.</th>
<th>CRS</th>
<th>JDiag</th>
<th>B.S.</th>
</tr>
</thead>
<tbody>
<tr>
<td>nos6</td>
<td>675</td>
<td>3255</td>
<td>38.658</td>
<td>5.441</td>
<td>20.634</td>
<td>32.945</td>
<td>2.570</td>
</tr>
<tr>
<td>2 × 25 × 1</td>
<td>625</td>
<td>3025</td>
<td>37.907</td>
<td>5.650</td>
<td>21.416</td>
<td>32.952</td>
<td>2.593</td>
</tr>
<tr>
<td>nos7</td>
<td>729</td>
<td>4617</td>
<td>35.749</td>
<td>4.836</td>
<td>20.000</td>
<td>27.830</td>
<td>3.259</td>
</tr>
<tr>
<td>2 × 10 × 3</td>
<td>300</td>
<td>4140</td>
<td>27.006</td>
<td>9.359</td>
<td>29.881</td>
<td>33.727</td>
<td>17.457</td>
</tr>
<tr>
<td>medium</td>
<td>181</td>
<td>2245</td>
<td>23.192</td>
<td>7.888</td>
<td>29.874</td>
<td>32.583</td>
<td>19.633</td>
</tr>
<tr>
<td>bcsstm27</td>
<td>1224</td>
<td>56k</td>
<td>15.130</td>
<td>4.807</td>
<td>23.677</td>
<td>21.604</td>
<td>28.907</td>
</tr>
<tr>
<td>e05r0000</td>
<td>236</td>
<td>5856</td>
<td>8.534</td>
<td>4.841</td>
<td>26.642</td>
<td>25.085</td>
<td>SEGV</td>
</tr>
<tr>
<td>3 × 17 × 7</td>
<td>34.4k</td>
<td>1.6M</td>
<td>8.478</td>
<td>4.752</td>
<td>23.499</td>
<td>11.805</td>
<td>27.615</td>
</tr>
</tbody>
</table>
NIST Sparse BLAS

• Algorithms

1. Matrix-matrix products (MM),

\[ C \leftarrow \alpha AB + \beta C \quad C \leftarrow \alpha A^T B + \beta C, \]

where \( A \) is sparse, \( B \) and \( C \) are dense, and \( \alpha \) and \( \beta \) are scalars.

2. Triangular solves,

\[ C \leftarrow \alpha DA^{-1} B + \beta C \quad C \leftarrow \alpha DA^{-T} B + \beta C \]
\[ C \leftarrow \alpha A^{-1} DB + \beta C \quad C \leftarrow \alpha A^{-T} DB + \beta C \]

where \( D \) is a “(block) diagonal” matrix.

3. Right permutation of a sparse matrix in Jagged Diagonal format,

\[ A \rightarrow AP \quad A \rightarrow AP^T \]

4. Integrity check of sparse \( A \).
NIST Sparse BLAS (cont).

- Storage formats
  - Point entry – each entry of the storage format is a single matrix element.
    - Coordinate
    - CCS
    - CRS
    - Sparse diagonal
    - ITPACK/ELLPACK
    - Jagged diagonal
    - Skyline
  - Block entry – each “entry” is a dense block of matrix elements.
    - Block coordinate
    - Block CCS
    - Block CRS
    - Block sparse diagonal
    - Block ITPACK/ELLPACK
    - Variable Block compressed Row storage (VBR)
NIST Sparse BLAS (cont).

- Limitations
  - Huge number of routines to implement.
    - **User-level** Only 4 routines.
    - **Toolkit-level** 52 (= 4 routines * 13 formats) routines.
    - **Lite-level** 2,964 (= 228 routines * 13 formats) routines.
  - Algorithms are not complete.
    - E.g., Matrix assembly, Incomplete and complete factorizations.
  - Data structures are not complete.
    - E.g., BlockSolve
  - Only one operand can be sparse.
    - No sparse $C = A \times B$. 
A Sparse Compiler

- Still need to develop sparse applications.
- Want to automate the task.

Design goals:

- Programmer selects the sparse storage formats.
- Programmer can specify novel storage formats.
- Sparse implementations as efficient as possible.
Challenges for sparse compilation

- Describing sparse matrix formats to the compiler.
- Transforming loops for efficient access of sparse matrices.
- Dealing with redundant dimensions.
- Accessing only Non-zeros.
Describing Storage Formats – Random Access

- Sparse matrices as objects with `get` and `set` methods.
- Dependencies are preserved.

```plaintext
for i = 1, n
  for j = 1, n
    y[i] += A.get(i, j) * x[j]
```

- Inefficient
  - Searching is inefficient.
  - Useless computation when $A[i, j] = 0$. 
Describing Storage Formats – Sequential Access

- Stream of non-zeros, \(<i,j,v>\).
- Sparse matrices as containers with iterations.

```python
for <i,j,v> in A.nzs():
    y[i] += v * x[j]
```

- What about dependencies? Must know order of iteration.
- Simultaneous enumeration...
• Consider $C = A \times B$, where $A$ and $B$ are sparse.

• With sequential access,

  
  for <i,k,Av> in A.nzs()
  for <k',j,Bv> in B.nzs()
  if k = k' then
    C[i,j] += Av * Bv

• a better solution,

  
  for <i,k,Av> in A.nzs()
  for <j,Bv> in B[k,:].nzs()
  C[i,j] += Av * Bv

• CRS gives us this type of access.
Indexed-sequential access

- Storage formats have hierarchical structure.
- Algebraic description of this structure.

- **Nesting**
  
  $c \rightarrow r \rightarrow v$

  ![Compressed Column Storage](image)

- **Aggregation**
  
  $(r \rightarrow c \rightarrow v) \cup (c \rightarrow r \rightarrow v)$

  ![CCS](image)

- **Linear Maps**
  
  map{$br*B+or \leftrightarrow r$, $bc*B+oc \leftrightarrow c$ :
  $bc \rightarrow br \rightarrow <or \quad oc> \rightarrow v$}

  ![Block Sparse Column](image)

- **Perspective**
  
  $(r \rightarrow c \rightarrow v) \oplus (c \rightarrow r \rightarrow v)$

  ![CRS](image)
Conveying the structure to the compiler

- Annotations

  \[ \text{\$SPARSE CRS: } r \rightarrow c \rightarrow v \]
  \[
  \text{real } A(0:99,0:99)
  \]

- Each production implemented as an abstract interface class

  ```cpp
  class CRS : public Indexed<int,
                               Indexed<int,
                               Value<double> > >
  ```
Challenges for sparse compilation

✓ Describing sparse matrix formats to the compiler.

\[ r \rightarrow c \rightarrow v \]

- Transforming loops for efficient access of sparse matrices.
- Dealing with redundant dimensions.
- Accessing only Non-zeros.
Loop transformation framework

- Extend our framework for imperfectly nested loop
- For each statement – Statement space = (Iteration space, Data space)

for i =
    for j = ...
        $S_1$: ... $A[F_1(i,j), F_2(i,j)]$
        $+ B[G(i,j)]$ ...

- Iteration space - as before
- Data space - Product of sparse array dimensions,

$$S_1: <i, j, a_1, a_2, b>$$

- Product space - Cartesian product of statement spaces
Loop transformation framework (cont.)

- Finding embedding functions
  - Add constraints for array refs $a = Fi$.
  - Use Farkas Lemma, as before.
- Account for the structure of sparse matrices,
  - map – change of variables, $P' = TP$
  - perspective – choice
  - aggregation – make two copies of indices, $a \rightarrow a', a''$
- Transformations - not tiling, instead Data-centric
  - order the array indices, $a_1, a_2, b_1, b_2, \ldots$
  - Partial transformation, bring array indices outermost
  - Complete transformation.
Challenges for sparse compilation

√ Describing sparse matrix formats to the compiler.

\[ r \rightarrow c \rightarrow v \]

√ Transforming loops for efficient access of sparse matrices.

- Augmented product space.
- Data-centric transformations.
- Dealing with redundant dimensions.
- Accessing only Non-zeros.
Redundent dimensions

• Dot product of two sparse vectors, $j \rightarrow v$.
  
  \[
  \text{for } i = 1, n \\
  \text{sum } += R[i] \times S[i]
  \]

• Statement and product space: $(i, r, s)^T$.

• Transform: $(r, s, i)^T$

• Constraints: $i = r = s$
  
  \[
  \text{for } <ir, a> \text{ in } R \\
  \text{for } <is, b> \text{ in } S \\
  \quad \text{if } ir = is \text{ then} \\
  \quad \quad \text{sum } += a \times b
  \]

• Two dimensions are redundant.
  
  • Dense code, random access:
    Replace $s$ and $i$ with $r$.
  
  • Sparse code, sequential access:
    simultaneous enumeration
    
    \[
    \text{for } <ir, a> \text{ in } R, \\
    <is, b> \text{ in } S, \text{ when } ir=is \\
    \text{sum } += a \times b
    \]
Connection with relational databases

- Relations – sets of tuples
- Join, $\Join$ – Constrained cross product.

\[ R \Join S = \{ <i, a, b> | <i, a> \in R, <i, b> \in S \} \]

- Connection:
  - Sparse matrices as relations.
  - Simultaneous enumeration as $\Join$. 
Join implementations

Implementations of $R \Join S$:

- Nested loop join
  \[ R \]
  \[ S \]

- Index join
  \[ R \]
  \[ S \]

- Hash join
  \[ S \]
  \[ R \]

- Sort-merge join
  \[ R \]
  \[ S \]
Simultaneous enumeration

- Identifying the joins –
  - Let $x$ be a vector of the transformed product space indices,
  - Let $Fx = f_0$ be the constraints on the indices (array access, ...),
  - Hermite Normal Form, $L = PFU$.
  - One join for each non-zero column of $L$.

- Affine constraints – more general join operation.
- Dependencies – constrain order of enumeration
- Checks for original loop bounds – use Fourier-Motzkin to simplify.
Results

Triangular solve - NIST C vs. NIST F77 vs. Bernoulli

SGI Octane, 300Mhz

Pentium II, 300Mhz