Bernoulli
A relational approach to the compilation of sparse matrix programs

The Problem
- Matrix Algorithms tend to be relatively simple
  - Matrix Vector multiplication
  - Matrix-Matrix multiplication
- Coding them for sparse matrices is hard
  - sparse matrix formats can be hard to work with
    - involve a lot of indirection
  - hence iteration space is very complex
- there are lots of matrix formats
  - Makes library-based approaches infeasible

Example: Matrix Vector Multiply
- In Dense Form
  - In Sparse form

The Solution
- Describe the sparse matrix format
  - opt for a declarative specification
  - we want to avoid pointer analysis
- Define the algorithm on a dense matrix
  - imperative specification (like in SKETCH)
  - but on a restricted language
    - we want to avoid dependence analysis
    - Convert it to a relational query
- Generate the sparse implementation
  - must iterate over the matrices in an efficient way
  - relational calculus helps us with this

An Example
- CRS Format
  - Compressed Row Storage
  - What we know
    - we can do random access of rows
    - within a row we can iterate over the columns

Another Example
- CCS Format
  - Compressed Column Storage
  - What we know
    - we can do random access of columns
    - within a column we can iterate over the rows
Another Example

- INODE Format

- What we know
  - we can do random access of inodes
  - within an inode I can iterate over row and column id

- What should we convey to the compiler?

Description of Sparse Format

- What the compiler needs to know
  - what is the hierarchy of indices
  - what can I do at each level of the hierarchy
    - can I do random access?
    - do I have to search sequentially?
  - how do I access the next level of the hierarchy

Specifying the hierarchy of indices

\[ T := V \mid F \mid F_{>op} \mid F \times F \times F \times \ldots > T \mid (F, F, \ldots) > T \mid T \mid U \]

- The > operator indicates the nesting of fields
  - Example: in CCS, we have i>j because we have to get the column before we can access an element in the row

Specifying the hierarchy of indices

\[ T := V \mid F \mid F_{>op} \mid F \times F \times F \times \ldots > T \mid (F, F, F, \ldots) > T \mid T \mid U \]

- The \((F_1 \times \ldots \times F_n)\) operator indicates that the indices corresponding to the \(F_i\) can be enumerated independently
  - Example: Dense storage
    \[(i \times j) > V\]

Specifying the hierarchy of indices

\[ T := V \mid F \mid F_{>op} \mid F \times F \times F \times \ldots > T \mid (F, F, F, \ldots) > T \mid T \mid U \]

- The \(U\) operator indicates alternative index hierarchies
  - Example: A combined CSS/CRS format
    \[(i \times j) \cup (j \times i) > V\]
Hidden fields

- Sometimes hidden fields are needed to specify additional structure
  - Example INODE:
    \[ \text{INODE} > (i \times j) > V \]
- Note: can’t handle unbounded recursion

An Example

- CRS Format
  - Compressed Row Storage
  - What we know
    - we can do random access of columns
    - within a row we can iterate over the rows
    \[ j > i > v \]

Another Example

- CCS Format
  - Compressed Column Storage
  - What we know
    - we can do random access of rows
    - within a column we can iterate over the columns
    \[ i > j > v \]

Another Example

- INODE Format
  - What we know
    - we can do random access of inodes
    - within an inode I can iterate over row and column id
    \[ \text{INODE} > (i \times j) > v \]

Access methods

- Must tell the compiler how to get to the actual entries in the matrix
- Three basic methods
  - Search(x)
    - return a pointer to entry containing x
  - Enum()
    - enumerate each value together with a pointer to it
  - Deref(p)
    - get the value referenced by p
- The details vary depending on the level of the hierarchy

Access Method example

- For CRS, for the column
  - Search(col) = \{true, col\}
    - Because col storage is dense
  - Enum() = \{(1,1), (2,2), ..., (n,n)\}
  - Deref(col) = rowstart
- For the row on a given column
  - Search(row) = \{b, i\}
    - b says whether that row exists, and I is it’s position in row
  - Enum()= the row array
  - Deref(i) = val
What about performance

- So far we haven’t said anything about performance
  - the following two formats are represented by the same index hierarchy \((i \times j) > V\)
  - one requires searching to find entries, the other one does not!

Performance information

- Each level of the hierarchy we need to provide the following info
  - Cost of searching
  - Ordering of indices
  - Range of indices
  - Type and range of handles
  - Arity of dereference
  - Kind of dereference

Access Method example

- For CRS, for the column
  - Search(col) = \(<true, col>\) \(O(1)\)
    - Because col storage is dense
  - Enum() = \(<1,1>, <2,2>, \ldots, <n,n>\) \(O(n)\)
  - Deref(col) = rowstart \(O(1)\)

- For the row on a given column
  - Search(row) = \(<b, i>\) \(O(n)\)
    - \(b\) says whether that row exists, and \(i\) is its position in row
  - Enum() = the row array \(O(1)\)
  - Deref(row) = \(val\) \(O(1)\)

The algorithm as relational query

- Matrices can be seen as relations \((i, j, \text{val})\)
- The traversal performed by the algorithm is just a join over different relations
  - The \(\star \), makes \(\text{Iter} \cdot i = Y.i = A.i\) and \(\text{Iter} \cdot j = A.j = X.j\)
  - Database people could solve this problem

\[
\begin{align*}
\text{Do } i = 1, N \\
\text{Do } j = 1, N \\
Y[i] = Y[i] + A[i, j] \times X[j];
\end{align*}
\]

- Only do-all loops are allowed in the spec
  - loop iteration order is arbitrary
  - eliminates need for dependence analysis

The compiler’s view

- The algorithm as relational query
  - how do we get equijoins?
  - how do we order them?
  - how do we implement joins?
- Implemented in a two level plan
  - high-level planning decides how to order the joins
  - Low-level planning decides how the joins are to be implemented

High-level planning: Ordering of joins

- Key idea:
  - ordering of joins should respect hierarchy of indices when possible
  - equijoins preferred whenever possible
Ordering of loops, Example

DO i = 1, n
DO j = 1, n
Y(i) = Y(i) + A(i-j, j)*X(j)

Relational Query
- We can define an affine equation to describe the selection
  \[ a = H \cdot i \]
  \[ a = \langle i, j, s, t, y, x \rangle \]
- vector a corresponds to actual indices to the matrices

Low-Level planning
- For this, we can use standard techniques from databases
- Tradeoffs between space and time
  - For example, when you do scatter/gather
- Also guided by the complexity information

Ordering of loops, Example

DO i = 1, n
DO j = 1, n
Y(i) = Y(i) + A(i-j, j)*X(j)

Relational Query
- We can rearrange the Matrix using Column operations
  \[ a = H' \cdot i \]
  \[ a = \langle s, i, j, t, y, x \rangle \]
- Now we can do equijoins easily

Conclusion
- Key Ideas
  - We need to provide the compiler with all the necessary information about the matrix format
  - The problem becomes tractable when we push it to a higher level
    - Relational queries in this case
    - Just like last week with garbage collection
  - Why is nobody using it?