Supporting Structured Data

Saman Amarasinghe

Fredrik Kjolstad (Stanford)
Stephen Chou (MIT)
Rawn Henry (MIT)
Olivia Hsu (Stanford)
Rohan Yadav (Stanford)
Changwan Hong (MIT)
Ryan Senanayake (MIT)
Daniel Donenfeld (MIT)
Peter Ahrens (MIT)
Shoaib Kamil (Adobe)
David Lugato (CEA)
World Is Built For Dense

Hardware Utilization
• Peak Performance (GEMM)
  • 70-80% of CPU
  • 80-90% of GPU
• Optimizations
  • Prefetching, Branch Predictions, TLB, cache, ..

Programming Systems
• Abstractions that Work across Different Algorithms (Dense Linear Algebra, Image Processing, Deep Learning, ..)
  • BLAS, Halide, TensorFlow, ...
• Optimizing Compilers
  • Tiling, Vectorization, Unrolling, ..
But...Most Data Has Structure

Sparsity

Replicated

Symmetry
For Example, Sparse Tensors Are Everywhere

Data Analytics
- Movies
- Social Networks
- Product Reviews

Machine Learning
- Sparse Convolutional Networks
- Sparse Networks
- Graph Convolutional Network

Science and Engineering
- Robotics
- Simulations

Extremely sparse
- Dense storage: 107 Exabytes
- Sparse storage: 13 Gigabytes

Products
- Amazon

Words
- Amazing
- Great
- Peter
- Lily
- Paul
- Billy
- Hide
- Bob
- Sam
- Mary

Customers
- Peter
- Lily
- Paul
- Billy
- Hide
- Bob
- Sam
- Mary

Dense storage: 107 Exabytes
Sparse storage: 13 Gigabytes
Ignoring Sparsity Is Throwing Away Performance

Sparse Matrix Vector Multiplication (SpMV)

8K x 8K double precision matrix in CSR
Ignoring Sparsity Is Throwing Away Performance

Sparse Matrix Matrix Multiplication (SpMV)

4K x 4K double precision matrix in CSR
Sparse Problems Are Everywhere

Density

0%  10%  Sparse Neural Networks  50%  100%

10^{-6} %  10^{-5} %  10^{-4} %  10^{-3} %  0.01 %  0.1 %  1 %

Dense Array programs

Density Log scale

Internet & Social Media  Recommendation Systems  Circuit Simulation  Computational Chemistry  Finite Element Methods  Electromagnetics Proteins  Fluid Dynamics  Problems in Statistics

[Hegde, et al., MICRO 2019]

Sparse Neural Networks

SpWM

SpMV
Complexity Of Sparse Code

\[ A_{ijk} = B_{ijk} + C_{ijk} \]

CSF

COO

```c
int kB0 = B2_crd[B2_pos];
int kB1 = B1_crd[B1_pos];
int kB2 = B2_crd[B2_pos1];
int kB3 = B2_crd[B2_pos2];
```

```c
int A2_pos0 = (A1_pos * A2_size) + kB0;
int A2_pos1 = (A1_pos * A2_size) + kB1;
int A2_pos2 = (A1_pos * A2_size) + kB2;
int A2_pos3 = (A1_pos * A2_size) + kB3;
int A2_pos4 = (A1_pos * A2_size) + kB4;
int A2_pos5 = (A1_pos * A2_size) + kB5;
int A2_pos6 = (A1_pos * A2_size) + kB6;
int A2_pos7 = (A1_pos * A2_size) + kB7;
```

```c
if (C1_pos < C1_end) { 
  while (C1_pos < C1_end && (C1_crd[C1_pos] != k)) { 
    C1_pos = C1_end; 
  } 
  while (C1_end < C1_crd[C1_end]) { 
    int C1_pos = C1_end; 
    C1_end++; 
  } 
  for (kB0 = B2_crd[B2_pos]; kB0 < B2_pos[B1_pos + 1]; kB0++) { 
    A2_pos0 = kB0;
  } 
}
```

```c
while ((C1_end < C1_crd[C1_end]) && (C1_crd[C1_end] == k)) { 
  C1_end++; 
}
```

```c
int A2_pos3 = (A1_pos * A2_size) + kB2;
int A2_pos4 = (A1_pos * A2_size) + kB3;
int A2_pos5 = (A1_pos * A2_size) + kB4;
int A2_pos6 = (A1_pos * A2_size) + kB5;
int A2_pos7 = (A1_pos * A2_size) + kB6;
```
Sparsity Is Currently Addressed One-Problem-At-A-Time

\[ a = Bc + a \]
\[ a = Bc \]
\[ a = Bc + b \]
\[ A = B + C \]
\[ a = \alpha Bc + \beta a \]
\[ a = B(c + d) \]
\[ A = B \]
\[ a = B^T Bc \]
\[ a = b + c \]
\[ A = B \]
\[ K = A^T C A \]

\[ A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{kj} \]
\[ A_{kj} = \sum_{il} B_{ikl} C_{lj} D_{kj} \]
\[ A_{lk} = \sum_{il} B_{ikl} C_{ij} D_{kj} \]
\[ A_{ik} = \sum_{il} B_{ikl} C_{lj} D_{kj} \]
\[ A_{ijl} = \sum_{k} B_{ijkl} C_{lkj} \]
\[ \tau = \sum_i z_i (\sum_j z_j \theta_{ij}) (\sum_k z_k \theta_{ik}) \]

\[ C = \sum_{ijkl} M_{ij} P_{jk} M_{kl} P_{il} \]
\[ a = \sum_{ijklmnop} M_{ij} P_{jk} M_{kl} P_{lm} M_{nm} P_{no} M_{po} P_{ip} \]
World Is Built For Dense...What About Sparse?

Hardware Utilization

- Peak Performance (GEMM)
  - 70-80% of CPU
  - 80-90% of GPU
- Optimizations
  - Prefetching, Branch Predictions, TLB, cache, ..

Programming Systems

- Abstractions that Work across Different Algorithms
  - BLAS, Halide, TensorFlow, ...
- Optimizing Compilers
  - Tiling, Vectorization, Unrolling, ..

- Peak Performance (PageRank, SpMv)
- < 10% Peak of CPU and GPU

Optimized PageRank for Multi-Core CPU
### Expression Language

- $A = Bc + a$
- $A = B\odot C$
- $A = B + C$
- $A = BCd$
- $A = Bc + a$
- $A = B\odot C$

### Format Language

- Dense Matrix
- DCSR
- CSR
- BCSR
- COO
- CSF
- DIA
- ELLPACK
- CSB
- Hash Maps
- Blocked COO
- CSC
- DCSC
- Sparse vector
- Blocked DIA
- Dense Tensors
- Blocked Tensors

### Schedule Language

- `pos`
- `reorder`
- `precompute`
- `divide`
- `vectorize`
- `split`
- `parallelize`

---

**Sparse Tensor Compiler (Taco)**

The Sparse Tensor Compiler (Taco) is a tool designed to efficiently handle sparse tensor operations. It supports various tensor formats, including Blocked Tensors (BCSR), COO, CSF, DIA, ELLPACK, and CSB. The compiler translates high-level expressions into optimized tensor computations, supporting languages such as CUDA for efficient execution on GPUs.

**THE C**

**PROGRAMMING LANGUAGE**

**CUDA**

**NVIDIA**
Structured Data Tensor Compiler

Expression Language

\[ A = Bc + a \]  \[ a = Be \]
\[ A = B \odot C \]  \[ A = B + C \]  \[ a = \alpha Be + \beta a \]
\[ A = BCD \]  \[ A = \alpha B \]  \[ A = 0 \]  \[ A = BC \]
\[ a = b \odot c \]  \[ A = B \odot (CD) \]
\[ A_{ij} = \sum_{kl} B_{iak} C_{kj} D_{kj} \]
\[ A_{i} = \sum_{j} B_{ij} C_{ij} D_{ij} \]
\[ A_{ij} = \sum_{k} B_{ijk} C_{ij} \]
\[ A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{kj} \]
\[ C = \sum_{i} M_{ij} P_{ik} P_{ij} \]
\[ \tau = \sum_{i} z_{ii} \sum_{j} z_{ij} \theta_{ij} (\sum_{k} z_{ik} \theta_{ik}) \]
\[ a = \sum_{ijklmnop} M_{ij} P_{ik} M_{kl} P_{lm} M_{mn} P_{np} P_{p} \]

Format Language

Dense Matrix  DCSR  CSR  BCSR
COO  CSF  DIA  ELLPACK  CSB
Hash Maps  Blocked COO  CSC
DCSC  Sparse vector  Blocked DIA
Dense Tensors  Blocked Tensors
PackBITs  Banded  VBR  Ragged
LZ77  RLE  Definite Symmetric

Schedule Language

pos  reorder  vectorize  parallelize
precompute  divide  split

Looplet Language

Lookup  Run  Spike
Pipeline  Stepper  Jumper
Switch  Shift
Generated Sparse Code Performance Matches Hand-Optimized Libraries

\[ a = Bc + a \quad a = Bc \]
\[ a = Bc + b \quad A = B + C \quad a = \alpha Bc + \beta a \]
\[ a = B^Tc \quad A = \alpha B \quad a = B(c + d) \]
\[ a = B^Tc + d \quad A = B + C + D \quad A = BC \]
\[ A = B \odot C \quad a = b \odot c \]
\[ A = BCd \quad A = B^T \quad a = B^T Bc \]
\[ a = b + c \quad A = B \quad K = A^TCA \]

\[ A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{kj} \quad A_{kj} = \sum_{il} B_{ikl} C_{lj} D_{ij} \]
\[ A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{kj} \quad A_{ij} = \sum_{k} B_{ijk} c_k \]
\[ A_{ijk} = \sum_{l} B_{ikl} C_{lj} \quad A_{ik} = \sum_{j} B_{ijk} c_j \]
\[ A_{ik} = \sum_{j} B_{ijk} c_i \quad A_{ijl} = \sum_{k} B_{ikl} C_{kj} \]
\[ A_{ijk} = \sum_{l} B_{ikl} C_{lj} \quad A_{ijl} = \sum_{k} B_{ikl} C_{kj} \]
\[ A_{ik} = \sum_{j} B_{ijk} c_j \quad A_{ijl} = \sum_{k} B_{ikl} C_{kj} \]
\[ \tau = \sum_{i} z_i \left( \sum_{j} z_j \theta_{ij} \right) \left( \sum_{k} z_k \theta_{ik} \right) \]
\[ C = \sum_{ijkl} M_{ij} P_{jk} M_{lk} P_{il} \]
\[ a = \sum_{ijklmnop} M_{ij} P_{jk} M_{kl} P_{lm} M_{nm} P_{no} M_{po} P_{ip} \]
Generated Sparse Code Performance Matches Hand-Optimized Libraries

Sampled Dense-Dense Matrix Multiplication

We will generate fused operations

\[ A = B \odot (CD) \]

Element-wise multiplication

64-inner product

10 inner product

This dot product need not be computed

Normalized time

<table>
<thead>
<tr>
<th>Library</th>
<th>Normalized time</th>
</tr>
</thead>
<tbody>
<tr>
<td>rma10</td>
<td>2412x</td>
</tr>
<tr>
<td>cop20k</td>
<td>24835x</td>
</tr>
<tr>
<td>scircuit</td>
<td>59496x</td>
</tr>
<tr>
<td>mac-econ</td>
<td>73405x</td>
</tr>
<tr>
<td>pwtk</td>
<td>22400x</td>
</tr>
</tbody>
</table>
Challenges Of Sparse Array Compilation

Irregular Data Structures

Sparse Iteration with limited $O(1)$ access

Avoid wasted work and iterations

Coiteration Over Complex Data

Optimize Parallelism and Locality
Challenges Of Sparse Array Compilation

Irregular Data Structures

Sparse Iteration with limited O(1) access

Avoid wasted work and iterations

Coiteration Over Complex Data

Optimize Parallelism and Locality

Irregular Data Structures

Format Language

<table>
<thead>
<tr>
<th>CSF</th>
<th>Dense</th>
<th>Compressed</th>
<th>Compressed</th>
</tr>
</thead>
</table>

Sparse iteration space with limited O(1) access helps in optimizing parallelism and locality.
Dense Tensors Are Flexible But Can Waste Memory

locate(1,2) = 1\times4 + 2
= 6
Sparse Tensors Can Be Compressed By Adding Metadata

Compress Coordinate rows (CSR)

Duplicates

cols

0 2 1 2 3 3

rrows

0 0 1 1 1 2

col(3) = ???

A B C D E F

0 1 2 3

0 1 2 3 4 5 6 7 8 9 10 11
We Model Tensor Formats As A Hierarchy Of Per-Dimension Formats
Per-Dimension Formats Can Be Composed In Many Ways

Dense

Compressed

Singleton

CSR

Per-Dimension Formats Can Be Composed In Many Ways

Dense

Compressed

Singleton

CSR

Axes: A B C D E F

CSR

Axes: A B C D E F
Per-Dimension Formats Can Be Composed In Many Ways

Coordinates

Dense

Compressed

Singleton

Per-Dimension Formats Can Be Composed In Many Ways
Level Formats Can Be Composed In Many Ways

<table>
<thead>
<tr>
<th>Tensor formats</th>
<th>Dense formats</th>
<th>Compressed formats</th>
<th>Singleton formats</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coordinate matrix</td>
<td>Compressed</td>
<td>Dense</td>
<td>Singleton</td>
</tr>
<tr>
<td></td>
<td>Singleton</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mode-generic tensor</td>
<td>Compressed</td>
<td>Dense</td>
<td>Compressed</td>
</tr>
<tr>
<td></td>
<td>Singleton</td>
<td>Dense</td>
<td>Dense</td>
</tr>
<tr>
<td></td>
<td>Dense</td>
<td>Dense</td>
<td>Dense</td>
</tr>
<tr>
<td>Hash map vector</td>
<td>Hashed</td>
<td>Hashed</td>
<td>Hashed</td>
</tr>
<tr>
<td></td>
<td>[Baskaran et al. 2012]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hash map matrix</td>
<td>Hashed</td>
<td>Hashed</td>
<td>Hashed</td>
</tr>
<tr>
<td></td>
<td>[Patwary et al. 2015]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CSR</td>
<td>Dense</td>
<td>Dense</td>
<td>Dense</td>
</tr>
<tr>
<td>BCSR</td>
<td>Dense</td>
<td>Dense</td>
<td>Dense</td>
</tr>
<tr>
<td>CSB</td>
<td>Dense</td>
<td>Dense</td>
<td>Dense</td>
</tr>
<tr>
<td>Dense array tensor</td>
<td>Dense</td>
<td>Dense</td>
<td>Dense</td>
</tr>
<tr>
<td>Coordinate tensor</td>
<td>Compressed</td>
<td>Dense</td>
<td>Dense</td>
</tr>
<tr>
<td></td>
<td>Singleton</td>
<td>Dense</td>
<td>Dense</td>
</tr>
<tr>
<td></td>
<td>Singleton</td>
<td>Dense</td>
<td>Dense</td>
</tr>
<tr>
<td>ELLPACK</td>
<td>Dense</td>
<td>Dense</td>
<td>Dense</td>
</tr>
<tr>
<td></td>
<td>Singleton</td>
<td>Dense</td>
<td>Dense</td>
</tr>
<tr>
<td></td>
<td>[Kincaid et al. 1989]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block DIA</td>
<td>Dense</td>
<td>Range</td>
<td>Offset</td>
</tr>
<tr>
<td></td>
<td>Dense</td>
<td>Dense</td>
<td>Dense</td>
</tr>
<tr>
<td></td>
<td>Offset</td>
<td>Dense</td>
<td>Dense</td>
</tr>
</tbody>
</table>

Level Formats Can Be Composed In Many Ways

[Tinney and Walker, 1967]
[Im and Yelick 1998]
[Buluç et al. 2009]
[Saad 2003]
[Patwary et al. 2015]
Sparsity Beyond Zero Fill Values

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>


## Compressed Level Format

<table>
<thead>
<tr>
<th>pos</th>
<th>coord</th>
<th>vals</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 1 2 6 7 0 1 2 3 4 5 6 7 0 1 2 0 1 2 3 4 5 7 7</td>
<td>1 1 1 5 5 6 6 6 6 1 1 1 1 3 3 3 1 1 1 8 8 8 2 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
### Compressed Level Format

<table>
<thead>
<tr>
<th>pos</th>
<th>0</th>
<th>5</th>
<th>13</th>
<th>16</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>coord</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>vals</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

### Compressed Level Format with a Fill Value

<table>
<thead>
<tr>
<th>pos</th>
<th>0</th>
<th>5</th>
<th>9</th>
<th>17</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>coord</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>vals</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Fill</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Sparsity Beyond Zero Fill Values

Compressed Level Format

<table>
<thead>
<tr>
<th>pos</th>
<th>0</th>
<th>5</th>
<th>13</th>
<th>16</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>coord</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>vals</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Compressed Level Format with a Fill Value

<table>
<thead>
<tr>
<th>pos</th>
<th>0</th>
<th>5</th>
<th>9</th>
<th>17</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>coord</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>vals</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Fill</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Run Length Encoding (RLE) Level Format

<table>
<thead>
<tr>
<th>pos</th>
<th>0</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>coord</td>
<td>0</td>
<td>3</td>
<td>6</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>vals</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>

- Extension of the Compressed Format
- Last value is the Fill Value
Performance Advantage in Lossless Compression

Edge Detection of MRI Image

Alpha Blending of Two Videos
Dynamic Sparse Tensors

- All formats so far (CSR, COO, DIA, ELLPACK, RLE etc.) are static
  - Computing on them can be very fast
  - But...inserting or deleting an element can be (asymptotically) slow

- Many real world Applications are dynamic

Dynamic Graph Processing

Sparse Neural Network Training

Dynamic Sparse Tensors

- Need pointer-based, recursive data structures
- Novel Node Schema Language
  - Automatically generate the data structures
  - Automatically Generate the code for iteration
Challenges Of Sparse Array Compilation

Irregular Data Structures

Sparse Iteration with limited O(1) access

Avoid wasted work and iterations

Coiteration Over Complex Data

Optimize Parallelism and Locality

Format Language

<table>
<thead>
<tr>
<th>CSF</th>
<th>Dense</th>
<th>Compressed</th>
<th>Compressed</th>
</tr>
</thead>
</table>

Irregular data structures: Sparse iteration space with limited O(1) access. Avoid wasted work and iterations. Coiteration over complex data. Optimize parallelism and locality.
Compressed Storage Formats

No $O(1)$ random access

$$\text{locate}(2, 0, 2)$$

Values

30 40 50 10 70 80 20 60
Sparse Iteration Spaces And Iteration Graphs

\[ A_{ij} = \sum_k B_{ijk} * c_k \]

Sparse

Intersection

Dense
Sparse Iteration Graph Examples

\[ a_i = \sum_j B_{ij} c_j \]

\[ a_i = \sum_j \alpha B_{ij} a_j + \beta d_i \]

\[ A_{ij} = \sum_k B_{ij} C_k \]

\[ A_{ik} = \sum_j \sum_l B_{ijkl} c_{jl} \]

\[ B_{3ij} = \sum_k B_{ij} (C_{ik} D_{kj}) \]

\[ A_{ijl} = \sum_k B_{ijk} C_{lk} \]

\[ A_{ij} = \sum_k \sum_l B_{ikl} C_{kj} D_{lj} \]
\[ A_{ij} = \sum_k B_{ijk} \times c_k \]
Challenges Of Sparse Array Compilation

Irregular Data Structures

Sparse Iteration with limited O(1) access

Avoid wasted work and iterations

Coiteration Over Complex Data

Optimize Parallelism and Locality

Irregular data structures

Sparse iteration space with limited O(1) access

Avoid wasted work and iterations

Format Language

Sparse Iteration Graphs

Coiteration Code Generation

Format Language

<table>
<thead>
<tr>
<th>Format Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSF</td>
</tr>
<tr>
<td>Dense</td>
</tr>
<tr>
<td>Compressed</td>
</tr>
<tr>
<td>Compressed</td>
</tr>
</tbody>
</table>
Merge Lattice For Multiplications

\[ a_i = b_i c_i \]

Multiplication requires intersection

```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int i = min(ib, ic);
    if (ib == i) pb1++;
    if (ic == i) pc1++;
}
if (ib == i && ic == i) {
    a[i] = b[pb1] * c[pc1];
}
```
while (pc1 < c1_pos[1]) {
  int i = c1_crd[pc1];
  a[i] = c[pc1++];
}

while (pb1 < b1_pos[1]) {
  int i = b1_crd[pb1];
  a[i] = b[pb1++];
}

else {
  a[i] = c[pc1];
}

if (ib == i && ic == i) {
  a[i] = b[pb1] + c[pc1];
}
else if (ib == i) {
  a[i] = b[pb1];
}
else {
  a[i] = c[pc1];
}

while (pb1 < b1_pos[1]) {
  int i = b1_crd[pb1];
  a[i] = b[pb1++];
}

while (pc1 < c1_pos[1]) {
  int i = c1_crd[pc1];
  a[i] = c[pc1++];
}

Addition requires union

\[ a_i = b_i + c_i \]
Merge Lattice For A Compound Expression

\[ a_i = b_i + c_i + d_i \]
Beyond Multiply And Add
Merge Lattice For Additions

\[ a_i = b_i + c_i \]

Addition requires union

```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int i = min(ib, ic);
    if (ib == i) pb1++;
    if (ic == i) pc1++;
}
else if (ib == i) {
    a[i] = b[pb1];
}
else if (ib == i) {
    a[i] = b[pb1] + c[pc1];
}
while (pb1 < b1_pos[1]) {
    int i = b1_crd[pb1];
    a[i] = b[pb1++];
}
while (pc1 < c1_pos[1]) {
    int i = c1_crd[pc1];
    a[i] = c[pc1++];
}
```
\[ a_i = b_i \oplus c_i \]

```c
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int i = min(ib, ic);
    if (ib == i && ic == i) {
        a[i] = b[pb1] + c[pc1];
    } else if (ib == i) {
        a[i] = b[pb1];
    } else {
        a[i] = c[pc1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
}
while (pb1 < b1_pos[1]) {
    int i = b1_crd[pb1];
    a[i] = b[pb1++];
}
while (pc1 < c1_pos[1]) {
    int i = c1_crd[pc1];
    a[i] = c[pc1++];
}
```
User Defined Functions

def bitwise_and(x, y):
    x, y => {
        return x & y;
    }

properties:
    commutative
    annihilator=0

def gcd(x, y):
    x, 0 => { return abs(x); }
    0, y => { return abs(y); }
    x, y => { x = abs(x);
              y = abs(y);
              while (x != 0) {
                  int t = x;
                  x = y % x;
                  y = t;
              }
              return y;
    }

iteration_space:
    \{x \neq 0\} \cup \{y \neq 0\}
Challenges Of Sparse Array Compilation

Irregular Data Structures

Sparse Iteration with limited O(1) access

Avoid wasted work and iterations

Coiteration Over Complex Data

Optimize Parallelism and Locality

Format Language

Sparse Iteration Graphs

Coiteration Code Generation

Looplet Language

<table>
<thead>
<tr>
<th>CSF</th>
<th>Dense</th>
<th>Compressed</th>
</tr>
</thead>
</table>

Irregular data structures

Sparse iteration space with limited O(1) access

Avoid wasted work and iterations

Coiteration over complex data

Optimize parallelism and locality

Format language

Sparse iteration graphs

Coiteration code generation

Looplet language
### Iteration And Coiteration

#### Dense

<table>
<thead>
<tr>
<th>Increment</th>
<th>Next</th>
<th>Run Length</th>
<th>Blocked</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increment</td>
<td>Next/lookup</td>
<td>.....</td>
<td>.....</td>
</tr>
</tbody>
</table>

#### Compressed

| Next/lookup | Next/next two-finger merge | ..... | ..... |

#### Run Length

| ..... | ..... | ..... | ..... |

#### Blocked

| ..... | ..... | ..... | ..... |
Loolept Language

- A general language to iterate over structured data
- Iterating over complex structured data expressed using a language of a few primitives
  - Lookup
  - Run
  - Spike
  - Pipeline
  - Stepper
  - Jumper
  - Shift
  - Switch

Pipeline

Stepper

Spike

Spike

Spike

Spike

Run | Scalar | Run | Scalar | Run | Scalar | Run | Scalar | Run
---|-------|---|-------|---|-------|---|-------|---
0 | 0 | 4.2 | 8.6 | 0 | 0 | 0.7 | 0 | 0 | 9.2 | 0 | 0
Looplet Language

• A general language to iterate over structured data
  • Iterating over complex structured data expressed using a language of a few primitives
    • Lookup
    • Run
    • Spike
    • Pipeline
    • Stepper
    • Jumper
    • Shift
    • Switch

• Code generation from the iteration protocols is simple and mechanical

```plaintext
for i = 1:y.start-1
  visit(i, 0)
for i = y.start:y.stop
  visit(i, y.val[i + 1 - start])
for i = y.stop + 1:end
  visit(i, 0)
```
Looplet Language

• A general language to iterate over structured data
  • Iterating over complex structured data expressed using a language of a few primitives
    • Lookup
    • Run
    • Spike
    • Pipeline
    • Stepper
    • Jumper
    • Shift
    • Switch

• Code generation from the iteration protocols is simple and mechanical

• To coiterate, merge the individual iteration protocols
  • Use rewrite rules to simplify
LoOplet Language Supports Many Types Of Structured Data

- Unifying what is currently done by multiple compilers
- Hybrid “have-it-all” formats
- Expanding into other types of structures
Challenges Of Sparse Array Compilation

Irregular Data Structures

Sparse Iteration with limited O(1) access

Avoid wasted work and iterations

Coiteration Over Complex Data

Optimize Parallelism and Locality

Format Language

Sparse Iteration Graphs

Coiteration Code Generation

Looplet Language

Scheduling Language

<table>
<thead>
<tr>
<th>Compressed</th>
<th>Dense</th>
<th>CSF</th>
</tr>
</thead>
</table>

We expose the sparse transformation primitives as a scheduler and atomics across warps.

We implement our transformation framework as an extension on top of the TACO system (commit 331188), Intel MKL [2017], and SPLATT [Steinberger et al. 2015] used for transforming different schedules where the best schedule depends on the specific hardware and software setup.

For the comparative studies we use the 17 matrices from [Karypis and Kumar 1999], but compose with our set of transformations that our technique produces code with good performance for the locality study in Section 8.6 where we use the SpMM algorithms where the best schedule differs based on the level of parallelism.

The transformations in this section are described in prior work [Spassov et al. 2019], but compose with our set of transformations where the best schedule differs based on the level of parallelism.

We precompute(IndexExpr e, vector<IndexVar> vars, int unrollFactor)

void precompute(IndexExpr e, vector<IndexVar> vars, int unrollFactor)

We add the following scheduling APIs to TACO, inspired by the Halide system for dense stencil computations [2017]:

optimize(1, 2, 3, 4)

In particular, the derived code reorders them.

Optimize(1, 2, 3, 4)

For the comparative studies we use the 17 matrices from [Karypis and Kumar 1999], but compose with our set of transformations where the best schedule differs based on the level of parallelism.

Optimize(1, 2, 3, 4)

To evaluate it, we compare the performance against the state-of-the-art library implementations of three important kernels.

Optimize(1, 2, 3, 4)

For most studies this is the point of our experiments is not to demonstrate that our technique produces code with good performance that is competitive with hand-optimized kernels.

Optimize(1, 2, 3, 4)
Sparse Iteration Space Transformations

Scheduling API

void reorder(IndexVar i1, IndexVar i2);
void fuse(IndexVar i, IndexVar j, IndexVar f)
void split(IndexVar i, IndexVar i1, IndexVar i2, int size)
void divide(IndexVar i, IndexVar i1, IndexVar i2, int size)
void pos(IndexVar i, IndexVar i1, IndexVar i2, int size)
void coord(IndexVar p, IndexVar i)
void parallelize(IndexVar i, ParallelUnit pu,
    ReductionStrategy rs)
void unroll(IndexVar i, int unrollFactor)
void bound(IndexVar i, int min, int max)
void precompute(IndexExpr e, vector<IndexVar> vars,
    Tensor w);
int compute(taco_tensor_t *y, taco_tensor_t *A, taco_tensor_t *x) {
    ...
    for (int32_t i = 0; i < A1_dimension; i++) {  
        for (int32_t jA = A2_pos[i]; jA < A2_pos[i + 1]; jA++) {
            int32_t j = A2_crd[jA];
            y_vals[i] = y_vals[i] + A_vals[jA] * x_vals[j];
        }
    }
    return 0;
}
Sparse Iteration Space Transformations

```c
int compute(taco_tensor_t *y, taco_tensor_t *A, taco_tensor_t *x) {
    ...  
    int32_t i_pos = 0;
    int32_t i = 0;
    for (int32_t fp = 0; fp < A->dim[1]; fp++) {
        if (fp >= A->dim[1])
            break;

        int32_t f = A->data[fp];
        while (fp >= A->dim[1]) {
            i_pos = fp_val;
            i = i_pos;
            }  
            y_val[i] += A_val[fp] * x_val[f];
        return 0;
    }
```
Sparse Iteration Space Transformations

```c
int compute(taco_tensor_t *y, taco_tensor_t *A, taco_tensor_t *x) {

    int32_t i_pos = 0;
    int32_t i = 0;
    for (int32_t block = 0; block < ((A2_pos[A1_dimension] + 2047) / 2048); block++) {
        for (int32_t warp = 0; warp < 8; warp++) {
            for (int32_t thread = 0; thread < 32; thread++) {
                int32_t fpos2 = thread + 8 + thread_nz;
                int32_t fpos1 = warp + 32 + fpos2;
                int32_t fposA = block + 2048 + fpos1;
                if (fposA >= A2_pos[A1_dimension]) break;

                int32_t f = A2_ord[fposA];
                while (fposA == A2_pos[i_pos + 1]) {
                    i_pos = i_pos + 1;
                    i = i_pos;
                }
                y_vals[i] = y_vals[i] + A_vals[fposA] * x_vals[f];
            }
        }
    }
    return 0;
}
```

Scheduling Commands

```c
.fuse(i, j, f)
.pos(i, fpos, A(i, j))
.split(fpos, block, fpos1, NNZ_PER_TB)
.split(fpos1, warp, fpos2, NNZ_PER_WARP)
.split(fpos2, thread, thread_nz, NNZ_PER_THREAD)
.reorder({block, warp, thread, thread_nz})
```
Sparse Iteration Space Transformations

Scheduling Commands

.fuse((i, j, f))

.split(fpos, block, fpos1, NNZ_PER_T3)

.split(fpos1, warp, fpos2, NNZ_PER_WARP)

.split(fpos2, thread, thread_nz, NNZ_PER_THREAD)

.reorder((block, warp, thread, thread_nz))

.parallelize(block, ParallelUnit::GPUBlock, OutputRaceStrategy::IgnoreRaces)

.parallelize(warp, ParallelUnit::GPUWarp, OutputRaceStrategy::IgnoreRaces)

.parallelize(thread, ParallelUnit::GPUThread, OutputRaceStrategy::Atoms)

int compute(taco_tensor_t *y, taco_tensor_t *A, taco_tensor_t *x)
{
    gpuErrchk(cudaMallocManaged(&xvals, sizeof(int32_t) * ((A2_pos[A1_dimension] + 2947) / 2048));
    i blockStarts = taco_binarySearchBeforeBlockLaunch(A2_pos, i blockStarts, (int32_t) 0, A1_dimension, (int32_t) 2048, (int32_t) 256, ((A2_pos[A1_dimension] + 2947) / 2048));

    computeDeviceKernel<<<(A2_pos[A1_dimension] + 2947) / 2048, 32 * B>>>(A, 1 blockStarts, x, y);

    cudaMemcpy(y + threadIdx.y, xvals[i], sizeof(int32_t) * 32);
Sparse Iteration Space Transformations

Scheduling Commands

```
.int compute(taco_tensor_t *y, taco_tensor_t *A, taco_tensor_t *x) {
  ....
  .gpuErrchk(cudaMallocManaged((void**)&i_blockStarts, sizeof(int32_t) *
  ((A2_pos[A1_dimension] + 2047) / 2048 + 1)));
  i_blockStarts = taco_binarySearchBeforeBlockLaunch(A2_pos, i_blockStarts,
  int32_t &s, A1_dimension, (int32_t) 2048, (int32_t) 256, ((A2_pos[A1_dimension] +
  2047) / 2048));

  .computeDeviceKernel0<<<(A2_pos[A1_dimension] + 2047) / 2048, 32 * 8>>>(A,
  i_blockStarts, x, y);
  .cudaDeviceSynchronize();
....
```
Sparse Iteration Space Transformations

Scheduling Commands

```c
.fuse(i, j, f)
.pos(f, fpos, A(i, j))
.split(fpos, block, fpos1, NNZ_PER_TB)
.split(fpos1, warp, fpos2, NNZ_PER_WARP)
.split(fpos2, thread, thread_nz, NNZ_PER_THREAD)
.reorder({block, warp, thread, thread_nz})
.precompute(precomputedExpr, thread_nz, thread_nz_pre, precomputed)
.unroll(thread_nz_pre, NNZ_PER_THREAD)
.parallelize(block, ParallelUnit::GPUBlock, OutputRaceStrategy::IgnoreRaces)
.parallelize(warp, ParallelUnit::GPUPWarp, OutputRaceStrategy::IgnoreRaces)
.parallelize(thread, ParallelUnit::GPUPThread, OutputRaceStrategy::Atomic)
```

```c
int compute(taco_tensor_t *y, taco_tensor_t *A, taco_tensor_t *x) {
    ... 
    gpuErrchk(cudaMallocManaged((void **)i_blockStarts, sizeof(int32_t) * 
    i_blockStarts = taco_binarySearchBeforeBlockLaunch(A2_pos, i_blockStarts, 
    ..
    ..
    ..
    ..
```
SpMV on GPU (NVIDIA V100)

MTTKRP on GPU (NVIDIA V100)
Format CSR({Dense, Sparse});
Format CSF({Sparse, Sparse, Sparse});
Format SVEC({Sparse});

Tensor<double> A({1024, 1024}, CSR);
Tensor<double> B = read("B.tns", CSF);
Tensor<double> c = read("c.tns", SVEC);

A(i, j) = B(i, j, k) * c(k);

A.compile();
A.assemble();
A.compute();

PyTACO

```python
import taco
from taco import sparse, dense
csr = taco.format([dense, sparse])
csf = taco.format([sparse, sparse, sparse])
sv = taco.format([sparse])

A = taco.Tensor([2, 3], csr, dtype=taco.float32)
B = taco.Tensor([2, 3, 4], csf, dtype=taco.float32)
C = taco.Tensor([14], sv, dtype=taco.float32)

B[0, 0, 0] = 1.0
B[1, 2, 0] = 2.0
B[1, 2, 1] = 3.0
C[0] = 4.0
C[1] = 5.0

# Using C++ style API
i, j, k = taco.get_index_vars(3)
A[i, j, k].assign(B[i, j, k] * C[k])
print(A)

# Using Python Style API
print(taco.tensor_dot(B, C))

# Using taco parser
print(taco.parse("A[i, j] = B[i, j, k] * C[k], B, C"))

# Using numpy style einsum
print(taco.einsum("ijk, k->ij", B, C))
```

Online Code Generator

tensor-compiler.org/codegen/

tensor-compiler.org

github.com/tensor-compiler/taco

Availability of TACO

Some Rights Reserved
(Vanilla) MIT License

tensor-compiler.org
Speeding Up Sparse Array Programming In The Python Ecosystem

SciPy
PyData/Sparse
NumPy
Python

xor(A, B)
right_shift(A, B)

ldexp(A, B)

power(A, B)
output fill=1
Publications On TACO

1. WACO: Learning workload-aware co-optimization of the format and schedule of a sparse tensor program.
   Jaeyeon Won, Chaiti Mendis, Joel Emer, Saman Amarasinghe.
   International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).
   Apr, 2023. BibTeX.

   Stephen Chou, Saman Amarasinghe.
   Proceedings of the ACM on Programming Languages.
   New York, NY, USA, Oct, 2022. BibTeX.

   Stephen Chou.
   Cambridge, MA, Aug, 2022. BibTeX.

   Rawn Henry, Olivia Hsu, Rohan Yadav, Stephen Chou, Kunle Okukotun, Saman Amarasinghe, Fredrik Kjolstad.
   Proceedings of the ACM on Programming Languages.
   Chicago, IL, USA, Oct, 2021. BibTeX.

   Stephen Chou, Fredrik Kjolstad, Saman Amarasinghe.

   Rawn Henry.
   MEng Thesis, Massachusetts Institute of Technology.

   MEng Thesis, Massachusetts Institute of Technology.

   Ryan Seranayake.
   MEng Thesis, Massachusetts Institute of Technology.

   Suzanne Mueller.
   MEng Thesis, Massachusetts Institute of Technology.

10. Sparse Tensor Algebra Compilation.
    Fredrik Kjolstad.

    Patricio Noyola.
    MEng Thesis, Massachusetts Institute of Technology.
    Cambridge, MA, May, 2019. BibTeX.

12. Tensor Algebra Compilation with Workspaces.
    Fredrik Kjolstad, Peter Ahrens, Shoaib Kamal, Saman Amarasinghe.
    International Symposium on Code Generation and Optimization.
    Feb, 2019. BibTeX.

    Sachin Dilip Shinde.
    MEng Thesis, Massachusetts Institute of Technology.
    Cambridge, MA, Feb, 2019. BibTeX.

    Stephen Chou, Fredrik Kjolstad, Saman Amarasinghe.
    Proceedings of the ACM on Programming Languages.
    New York, NY, USA, Oct, 2018. BibTeX.

    Stephen Chou.
    SM Thesis, Massachusetts Institute of Technology.
    Cambridge, MA, Feb, 2018. BibTeX.

16. The Tensor Algebra Compiler.
    Fredrik Kjolstad, Shoaib Kamal, Stephen Chou, David Lugato, Saman Amarasinghe.
    Proceedings of the ACM on Programming Languages.
    New York, NY, USA, Oct, 2017. BibTeX.

17. Distinguished Distinguished Paper Award.
    Fredrik Kjolstad, Stephen Chou, David Lugato, Saman Amarasinghe.
    Piscataway, NJ, USA, 2017. BibTeX.

    Parker Allen Tew.
    MEng Thesis, Massachusetts Institute of Technology.
    Cambridge, MA, Jun, 2018. BibTeX.

http://groups.csail.mit.edu/commit/?page=publications
What Is Next...

Commit Group

- Novel formats for
  - Faster computation
  - More compactness
  - More types of structured data
- Scheduling
  - Algorithmic Auto-scheduling
  - Learned Auto-scheduling
- Unifying Sparsity
  - Tensors: TACO
  - Graphs: GraphIt
- Hardware Support for Sparsity

Stanford Compiler and Language Group

- Scheduling
- Algorithmic Auto-scheduling
- Learned Auto-scheduling

Unified Sparse

- Sparse Array
- Relational

Hardware Support for Sparsity

- Reimplementation of TACO sparse tensor theory in MLDIR

Integrating TACO into Pydata Sparse
Commit Group

• Current & recent projects
  • SEQ: A DSL for bio informatics
  • TACO: A DSL for sparse tensor algebra
  • GraphIt: A DSL for graph analysts
  • BuildIt: A Multistage programming framework in C++
  • CoLa: A DSL for data compression
  • SimIt: A DSL for sparse systems
  • MILK: A DSL for Optimizing indirect memory references
  • Cimple: A DSL for Instruction and Memory Level Parallelism
  • Codon: A Pythonic DSL framework
  • Tiramisu: A polyhedral compiler for data parallel algorithms
  • Ithemal: Performance prediction using machine learning
  • VeGen: Generating Vectorizers for vector instructions beyond SIMD
  • Vemal: Vectorization using machine learning
  • goSLP & Revec: Modernizing vectorization technology
  • OpenTuner: An extensible framework for program autotuning
Thank You

http://tensor-compiler.org/

This Work Supported By: