Sparse arrays for 3D sim

Brett Viren

November 14, 2024

Problem reminder

Calculate the *measure* of a *signal* observed through some *response* operation:

$$M = R \circ S$$

We model R is a set of field responses giving the induced current at equal time steps along a family of paths that begin on a sub-pixel 2D grid of *impact positions* that span each of a 2D grid of pixels. The measurement is a sum over the impact positions and the operation becomes a *partitioned convolution*:

$$M = \sum_p R_p * S_p$$

Where:

- All arrays are 3D: 1D ticks × 2D pixels.
- $p \in 10 \times 10 = 100$ selects the rows in R or S that correspond to a common (relative) *impact position* across the pixels.
- S_p is shaped (ticks × pixels) of about $10,000 \times 100 \times 100 = 10^8$ elements
- R_p is shaped $1000 \times 5 \times 5 = 2.5 \times 10^4$ elements

Strategy - exploit sparseness of signal

S_p is sparse, and we should (must) exploit that. Two basic strategies:

1. Apply $R_p * S_p$ convolution to **sparse array** representations.
2. Chunk S_p, ignore empty chunks, solve the rest densely, stitch the per-chunk solutions.

Here, we consider the implications of strategy 1.

Sparse arrays

Several ways to represent **2D arrays** sparsely

COO	coordinate, list of `(row,col,val)`
CSR	compressed sparse row, lists: `row_extens, col_indices, values`
CSC	transpose of CSR
BSR	block-sparse row, common-sized blocks and their locations
DIA	diagonal storage, list of block-diagonal dense matrices
DOK	dictionary of keys, `(row,col)->val`
LIL	list-of-lists, list of rows, row is list of `(col,val)`

Different formats have different speed/memory depending on nature of sparseness.

Different array dimensionality

The formats defined in terms of 2D, extension possible, but not common in practice.

- Numpy/SciPY, CuPY and Torch support most of these formats for 2D.
  - CuPY sparse arrays are built from initial SciPY arrays
- Only Torch supports 3D sparse arrays.

Library support for sparse convolution

To best I can find, there is only support for 2D.

- That is either in “experimental” modules or NN-related (or both).

Direct, interval-space sparse convolution

We may implement convolution on sparse N-D array “by hand”

- Exhaustive looping is far too slow.
- Parallelism subject to the many-to-many accumulation problem.

Convolution matrix representation

2D convolution can be converted to matrix-vector multiplication

$$B = K * A \rightarrow \vec{b} = M_k \cdot \vec{a}$$

With:

- *Unrolled* 2D arrays $\rightarrow$ 1D vectors $A \rightarrow \vec{a},\ B \rightarrow \vec{b}$ of sizes $n_a,\ n_b$, respectively.
- Transform the kernel array to a matrix $K \rightarrow M_k$ of shape (n_b, n_a)

The M_k is sparse matrix, each row has all the rows of K placed in according to how that row transforms $\vec{a}$ into one element of $\vec{b}$ through the elements of K.
See section 4.1 of https://arxiv.org/abs/1603.07285 for nice example.
We can now contemplate performing $M_k \cdot \vec{a}$ via sparse matrix-vector multiply.

Sparse matrix-vector multiply

$$\vec{b} = M \cdot \vec{a}$$

Mathematically, either or both of M and $\vec{a}$ may be sparse.

- No GPU library supports having both sparse.
- Oddly, SciPy supports only (csc)·(csr) and (dia)·(dok)

So, in practice we may only pick one as sparse. For our problem:

- $\vec{a}$ is size 10^8 with small but hard to estimate density.
- M_k is shape $(10^8, 10^8)$ with density $2.5 \times 10^4 / 10^8 = 0.025\%$

M_k is impossible to have dense in any case so we must require $\vec{a}$ to be dense.

Scaling of timing

Method

Dense matrix-vector multiply of dimension size N scales as $\mathcal{O}(N^2)$

- Perform sparse and dense matrix multiply timings at $N = 10^5$.
  - Take best time over supported sparse formats
- Scale by N^2 from $10^5 \rightarrow 10^8$.
  - Caveat: the actual sparse scaling is unknown (by me).
- Check density 2.5×10^{-4} which and more dense 2.5×10^{-3}

Results at density 2.5×10^{-4}

library	device	dense (s)	sparse (s)	dense 10^8	sparse 10^8
scipy	CPU	0.5	0.2	5.8 days	2.3 days
torch	CPU	0.6	0.03	6.9 days	8.3 hours
torch	GPU	0.003	0.001	50 minutes	17 minutes
cupy	GPU	0.003	0.004	50 minutes	67 minutes

The artificially denser 2.5×10^{-3} is a few to $10\times$ slower.

Conclusions

- Novel development needed to support 3D sparse array data type.
  - Unless PyTorch used.
- Novel development needed to support 3D sparse convolution.
- Sparse convolution-matrix method is untenable even with GPU.
- Computation time increases with density (unsurprising).

Sparse array approach is not worth pursuing.