Understanding how to implement sparse matrix operations efficiently

Simon Dobson

26 February 2025

Lots of modern software is built around linear algebra, for scientific computing and increasingly for systems that use machine learning, AI, or realistic immersive graphics. The matrices and tensors involved are typically *dense*, with the majority of elements being non-zero. This density makes the data structures easy to store in memory (as blocks of numbers) and to compute with using regular sequences of operations applied in bulk to the data in patterns that can usually be determined (and therefore scheduled) at compile-time. The basic operations of element-wise multiplication and row- or column-wise summation are now typically performed on GPUs and their variants (NPUs and TPUs) that exploit this regularity to obtain through parallelism a much higher performance than is possible on a traditional CPU..

There is however another family of matrix/tensor calculations that do not share these attractive features. *Sparse* matrices have the majority of elements being zero. This changes everything, since many (or even most) addition and multiplication operations will involve a zero operand, so performing the computations naïvely will waste time, while storing all the elements will waste space.

A typical example of sparse matrices comes from network science, where the connections between nodes can be represented by an adjacency matrix. Since most nodes are *not* connected to each other by edges, adjacency matrices are sparse. If one had a million-node network where each node has ten neighbours, then each row of the matrix would have only ten non-zero values.

Sparse matrices arise in lots of applications – including, increasingly, in AI, where new models such as DeepSeek generate sparse networks as part of their training. There seems therefore to be an increasing need to understand how sparse matrices can be efficiently represented and processed.

Efficiently programming with sparse matrices involves addressing several

problems simultaneously. There are several different storage formats, each better suited to some operations than to others. The basic arithmetic operations need to be expressed against these storage options. The calculations are irregular and are strongly dependent on the exact data values and patterns of sparsity, which works against using compile-time scheduling. Some useful programming techniques such as re-indexing to avoid data movement, affect locality of reference. This leads to more complicated interactions between core, cache, and memory, as well as offering fewer obvious opportunities for parallelism.

In this work we are studying these issues and their trade-offs, with a view to understanding whether and how we can accelerate sparse matrix calculations. This talk will focus on the systems problems involved, the approach we are using to explore them, and give examples of some of the irregularities of storage and calculation and their implications for efficient processing.