Efficient sparse matrix computations and their generalization to graph computing applications

Albert-Jan Yzelman
Parallel Computing and Big Data
Paris Research Centre
10th of February, 2017
Usable and efficient sparse matrix–vector multiplication

Introduction

Given a **sparse** $m \times n$ matrix $A$, and corresponding vectors $x, y$.

- How to calculate $y = Ax$ as fast as possible?
- How to make the code **usable**?

![Wikipedia link matrix ('07) with on average \approx 12.6 nonzeros per row.](image)

Figure: Wikipedia link matrix ('07) with on average $\approx 12.6$ nonzeros per row.
Usable and efficient sparse matrix–vector multiplication

Central obstacles for SpMV multiplication

**Shared-memory:**
- inefficient cache use,
- limited memory bandwidth, and
- non-uniform memory access (NUMA).

**Distributed-memory:**
- inefficient network use.
Central obstacles for SpMV multiplication

Shared-memory:

- inefficient cache use,
- limited memory bandwidth, and
- non-uniform memory access (NUMA).

Distributed-memory:

- inefficient network use.

Shared-memory and distributed-memory share their objectives:

minimisation of data movement.

Usable and efficient sparse matrix–vector multiplication

Inefficient cache use

Visualisation of the SpMV multiplication $Ax = y$ with nonzeroes processed in row-major order:

Accesses on the input vector are completely unpredictable.
Usable and efficient sparse matrix–vector multiplication

Enhanced cache use: nonzero reorderings

**Blocking** to cache subvectors, and **cache-oblivious traversals**.

Other approaches: no blocking (Haase et al.), Morton Z-curves and bisection (Martone et al.), Z-curve within blocks (Buluç et al.), composition of low-level blocking (Vuduc et al.), ...

Usable and efficient sparse matrix-vector multiplication

Enhanced cache use: nonzero reorderings

**Blocking** to cache subvectors, and **cache-oblivious traversals**.

Sequential SpMV multiplication on the Wikipedia ’07 link matrix:
345 (CRS), 203 (Hilbert), 245 (blocked Hilbert) ms/mul.

Usable and efficient sparse matrix–vector multiplication

Enhanced cache use: matrix permutations

Usable and efficient sparse matrix–vector multiplication

Enhanced cache use: matrix permutations

Practical gains:

Figure: the Stanford link matrix (left) and its 20-part reordering (right).

Sequential execution using CRS on Stanford:

18.99 (original), 9.92 (1D), 9.35 (2D) ms/mul.

Usable and efficient sparse matrix–vector multiplication

Bandwidth

Theoretical turnover points: Intel Xeon E3-1225

- 64 operations per word (with vectorisation)
- 16 operations per word (without vectorisation)

(Image taken from da Silva et al., DOI 10.1155/2013/428078, Creative Commons Attribution License)
Usable and efficient sparse matrix–vector multiplication

Bandwidth

Consequence: compression leads to better performance.

- Coordinate format storage: $\Theta(3nz)$
- Compressed Row Storage (CRS): $\Theta(2nz + m + 1)$
- Bi-directional Incremental CRS: $\Theta(2nz + row\_jumps + 1)$

$$A = \begin{pmatrix} 4 & 1 & 3 & 0 \\ 0 & 0 & 2 & 3 \\ 1 & 0 & 0 & 2 \\ 7 & 0 & 1 & 1 \end{pmatrix}$$

Need to consider the whole picture; good cache efficiency but no compression or compression but no cache optimisation? No gain!

Usable and efficient sparse matrix–vector multiplication

Efficient bandwidth use

With BICRS you can

- vectorise,
- compress,
- do blocking,
- have arbitrary nonzero or block orders.

Optimised BICRS takes less than or equal to $2nz + m$ of memory.


Each socket has **local** main memory where access is **fast**.

Memory access between sockets is slower, leading to *non-uniform memory access* (NUMA).
One-dimensional data placement

Coarse-grain row-wise distribution, compressed, cache-optimised:

- explicit allocation of separate matrix parts per core,
- explicit allocation of the output vector on the various sockets,
- interleaved allocation of the input vector,

Two-dimensional data placement

Distribute row- and column-wise (individual non-zeroes):

- most work touches only local data,
- inter-process communication minimised by partitioning;
- incurs cost of partitioning.


Results

Sequential CRS on Wikipedia '07: 472 ms/mul. 40 threads BICRS:

21.3 (1D), 20.7 (2D) ms/mul. Speedup: \(\approx 22x\).
# Results

Sequential CRS on Wikipedia ’07: **472 ms/mul.** 40 threads BICRS:

\[
21.3 \text{ (1D), } 20.7 \text{ (2D) ms/mul. Speedup: } \approx 22x.
\]

Average speedup on six large matrices:

<table>
<thead>
<tr>
<th></th>
<th>2 x 6</th>
<th>4 x 10</th>
<th>8 x 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>–, 1D fine-grained, CRS*</td>
<td>4.6</td>
<td>6.8</td>
<td>6.2</td>
</tr>
<tr>
<td>Hilbert, Blocking, 1D, BICRS*</td>
<td>5.4</td>
<td>19.2</td>
<td>24.6</td>
</tr>
<tr>
<td>Hilbert, Blocking, 2D, BICRS†</td>
<td>–</td>
<td>21.3</td>
<td><strong>30.8</strong></td>
</tr>
</tbody>
</table>

†: uses an updated test set. (Added for reference versus a good 2D algorithm.)

As **NUMA scales up, 1D algorithms lose efficiency.**


Usable and efficient sparse matrix–vector multiplication

Usability

Wish list:

- Performance and scalability.
- Better usability. Standardised API? Generalised Sparse BLAS:
  GraphBLAS.org
- Interoperability with Big Data:
  EYWA, Spark, Hadoop, DSLs, ...
- Interoperability with classic HPC:
  MPI + { PThreads, Cilk, OpenMP, ...}

Ref.: Buluč & Gilbert, The Combinatorial BLAS, ICHPCA, 2011
Ref.: Zhang, Zalewski, Lumsdaine, Misurda, & McMillan, GBTL-CUDA, IPDPS, 2016
Ref.: Ekanadham, Horn, Kumar, Jann, Moreira, Pattnaik, Serrano, Tanase, & Yu, Graph programming interface (GPI), ACM ICCF, 2016
Usable and efficient sparse matrix–vector multiplication

GraphBLAS

A ‘generalised’ semiring is given by

\[ <D_1, D_2, D_3, D_4, \oplus, \otimes, 0, 1> \]

with

\[ \oplus: D_3 \times D_4 \rightarrow D_4 \]
\[ \otimes: D_1 \times D_2 \rightarrow D_3 \]

These operators have to follow some basic rules, such as:

\[ \oplus(a, b) = \oplus(b, a), \quad \oplus(\oplus(a, b), c) = \oplus(a, \oplus(b, c)), \quad \otimes(\otimes(a, b), c) = \otimes(a, \otimes(b, c)), \quad \oplus(a, 0) = \oplus(0, a) = a, \quad \otimes(a, 1) = \otimes(1, a) = a, \quad \otimes(a, 0) = \otimes(0, a) = 0. \]

If these are true, (sparse) linear algebra ‘works’; we can apply all of our performance optimisations regardless of the operators selected!
Platforms like Spark allow programmers to ignore data placement issues, thus negatively impacting performance. It’s a classic tradeoff:

- **automatic mode** vs. **direct mode**
- **ease-of-use** vs. **performance**

Platforms like Spark allow programmers to **ignore data placement issues**, thus negatively impacting performance. It’s a classic tradeoff:

- **automatic mode** vs. **direct mode**
- **ease-of-use** vs. **performance**


A bridge between Big Data and HPC:

- **Spark I/O via native RDDs** and native Scala interfaces;
- Rely on serialisation and the JNI to **switch to C**;
- Intercept Spark’s execution model to **switch to direct mode**;
- Set up and enable **inter-process RDMA communications**.

Both usable *and* performant!
Usable and efficient sparse matrix–vector multiplication

Bridging Big Data and HPC

We have a shared-memory prototype. Preliminary results:

- SpMM multiply, SpMV multiply, and basic vector operations;
- one machine learning application, plus one on graph analysis.

Cage15, $n = 5\,154\,859$, $nz = 99\,199\,551$. Using the 1D method:

This is ongoing work. Performance will be improved, functionality extended.
Conclusions and future work

We know how to do fast sparse computations

- use same techniques for graph computing.

Future work:

- faster partitioning to enable scalable 2D sparse computations,
- sparse power kernels,
- symmetric matrix support,
- hypergraph and sparse tensor computations,
- support various hardware and execution platforms (Hadoop?).

The high performance (non-generalised) SpMV multiplication codes are free:
http://albert-jan.yzelman.net/software#SL

Thank you!
Backup slides
A working example:

```cpp
#include <graphblas.hpp>
int main() {
    const size_t num_cities = ... //some input matrix size
    grb::init();
    grb::Matrix< double > distances( num_cities, num_cities );
    grb::build( distances, ... ); //input data from file
    //or memory
    grb::Vector< double > x( num_cities ), y( num_cities );
    grb::set( x, 0.0, 4 );  //set city number 4 to
    //have distance 0.0
    ...
```
A working example (continued):

...  
//declare an alternative semiring on doubles:
grb::Semiring< double, double, double, double,
grb::operators::min,   //‘plus’
grb::operators::add,   //‘multiply’
grb::identities::infinity //‘0’
grb::identities::zero  //‘1’
> ring;

//calculate the shortest distances from all cities to
//city #4, allowing only a single path
grb::mxv( y, distances, x, ring);
...
A working example (continued):

```cpp
...
//calculate the shortest distances from all cities to
//city #4, allowing only a single path
grb::mxv( y, distances, x, ring );

//calculate the shortest distances from all cities to
//city #4, allowing two ‘hops’
grb::mxv( x, distances, y, ring );

//example output via iterators and exit:
writeResult( x.cbegin(), x.cend(), ... );
grb::finalize();
return 0;
```
Results: cross platform

Cross platform results over 24 matrices:

<table>
<thead>
<tr>
<th></th>
<th>Structured</th>
<th>Unstructured</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Xeon Phi</td>
<td>21.6</td>
<td>8.7</td>
<td>15.2</td>
</tr>
<tr>
<td>2x Ivy Bridge CPU</td>
<td>23.5</td>
<td>14.6</td>
<td>19.0</td>
</tr>
<tr>
<td>NVIDIA K20X GPU</td>
<td>16.7</td>
<td>13.3</td>
<td>15.0</td>
</tr>
</tbody>
</table>

no one solution fits all.

If we must, some generalising statements:

- Large structured matrices: GPUs.
- Large unstructured matrices: CPUs or GPUs.
- Smaller matrices: Xeon Phi or CPUs.

Usable and efficient sparse matrix–vector multiplication

Vectorised BICRS

A. N. Yzelman