Generalised vectorisation for sparse matrix–vector multiplication

Albert-Jan Yzelman

22nd of July, 2014

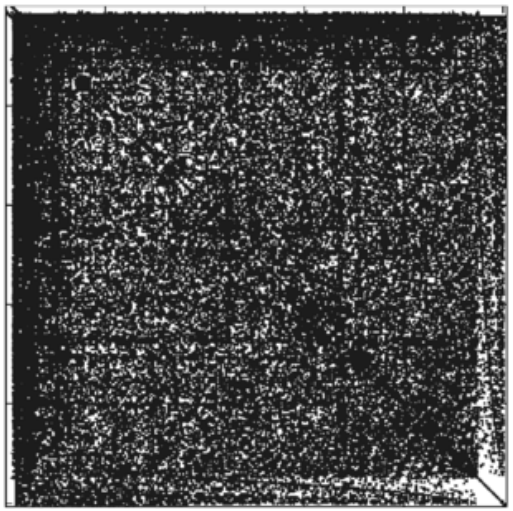
Context

Solve $y = Ax$, with

- A an $m \times n$ input matrix,
- x an input vector of length n, and
- y an output vector of length m.

Structured and unstructured **sparse** matrices:

Emilia_923
(unstructured mesh computations)



RH_Pentium
(circuit simulation)

Context

Past work on high-level approaches to SpMV multiplication:

	4 x 10	8 x 8
OpenMP CRS	8.8	7.2
PThread 1D	13.6	20.0
Cilk CSB	22.9	26.9
BSP 2D	21.3	30.8

4 x 10: HP DL-580, 4 sockets, 10 core Intel Xeon E7-4870
8 x 8 : HP DL-980, 8 sockets, 8 core Intel Xeon E7-2830

Yzelman and Roose, *High-Level Strategies for Parallel Shared-Memory Sparse Matrix–Vector Multiplication*, IEEE Trans. Parallel and Distributed Systems, doi:10.1109/TPDS.2013.31 (2014).

Yzelman, Bisseling, Roose, and Meerbergen, *MulticoreBSP for C: a high-performance library for shared-memory parallel programming*, Intl. J. Parallel Programming, doi:10.1007/s10766-013-0262-9 (2014).

This talk instead focuses on sequential, low-level optimisations.

Context

One operation, one flop:

scalar addition	$a := b + c$
scalar multiplication	$a := b \cdot c$

Context

One operation, l flops:

vectorised addition $\begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{l-1} \end{pmatrix} := \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_{l-1} \end{pmatrix} + \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{l-1} \end{pmatrix}$

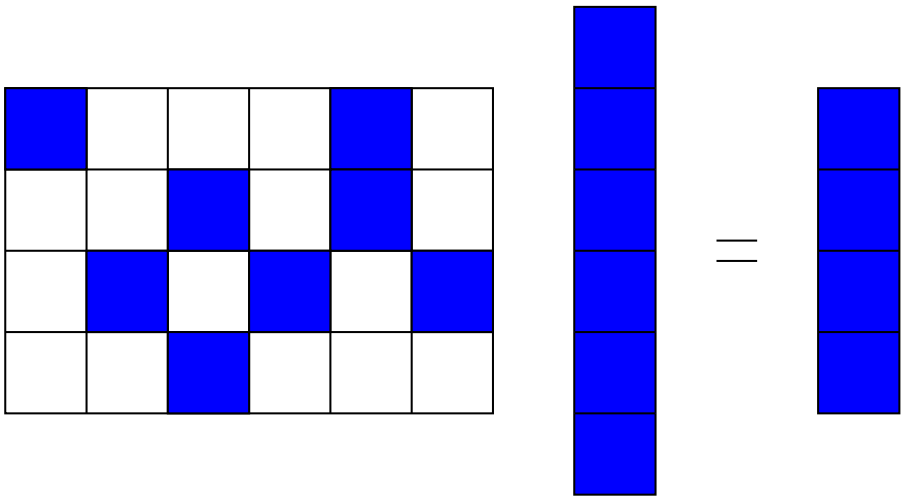
Context

One operation, l flops:

vectorised multiplication $\begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{l-1} \end{pmatrix} := \begin{pmatrix} b_0 \cdot c_0 \\ b_1 \cdot c_1 \\ \vdots \\ b_{l-1} \cdot c_{l-1} \end{pmatrix}$

Context

Exploiting sparsity through computation using only nonzeroes.
Leads to **sparse data structures**:



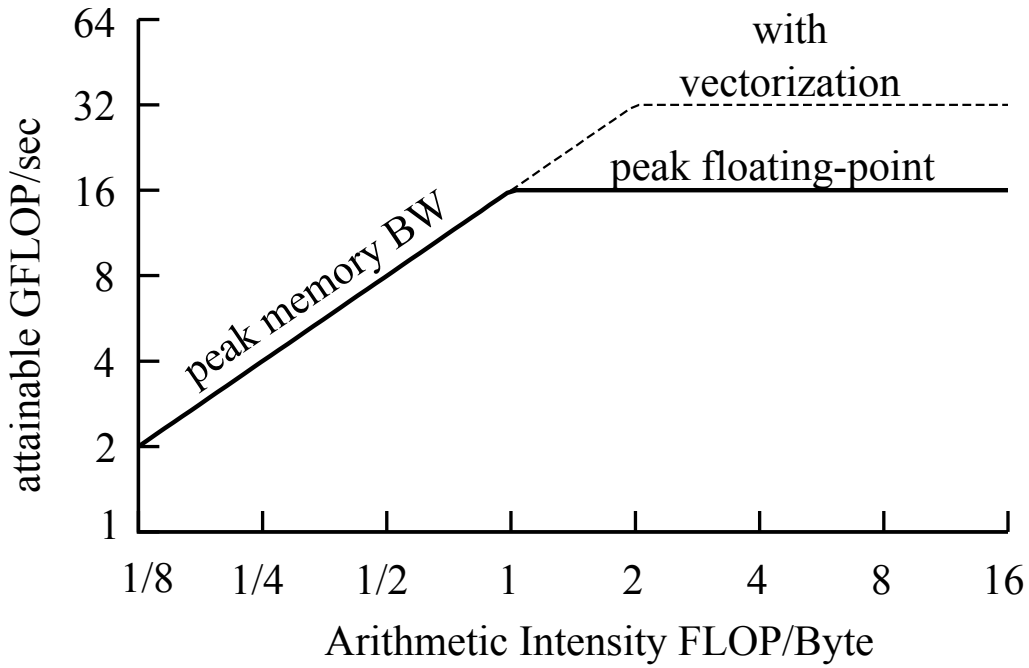
$$i = (0,\ 0,\ 1,\ 1,\ 2,\ 2,\ 2,\ 3)$$
$$j = (0,\ 4,\ 2,\ 4,\ 1,\ 3,\ 5,\ 2)$$
$$v = (a_{00},\ a_{04},\ \ldots,\ a_{32})$$

for $k = 0$ **to** $nz - 1$

$$y_{i_k} := y_{i_k} + v_k \cdot x_{j_k}$$

The coordinate (COO) format: two flops versus five data words.

Context

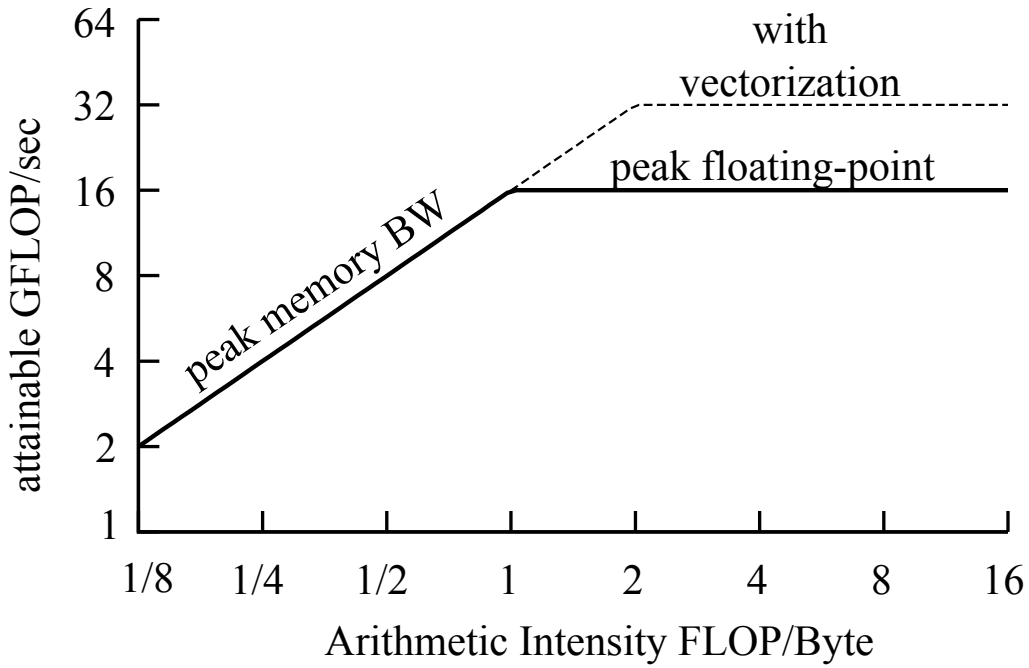


Theoretical turnover points: Intel Xeon E3-1225

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

(Image courtesy of Prof. Wim Vanroose, UA)

Context



Theoretical turnover points: Intel Xeon Phi

- 28 operations per word (with vectorisation)
- 4 operations per word (without vectorisation)

(Image courtesy of Prof. Wim Vanroose, UA)

Motivation

Why care about vectorisation:

(in sparse computations)

Computations can become **latency bound**.

- Good **caching** increases the effective bandwidth, reduces data access latencies.
- Vectorisation allows retrieving multiple data elements per CPU cycle; better **latency hiding**.

Motivation

Vectorisation also strongly relates to **blocking** and **tiling**.

Lots of earlier work:

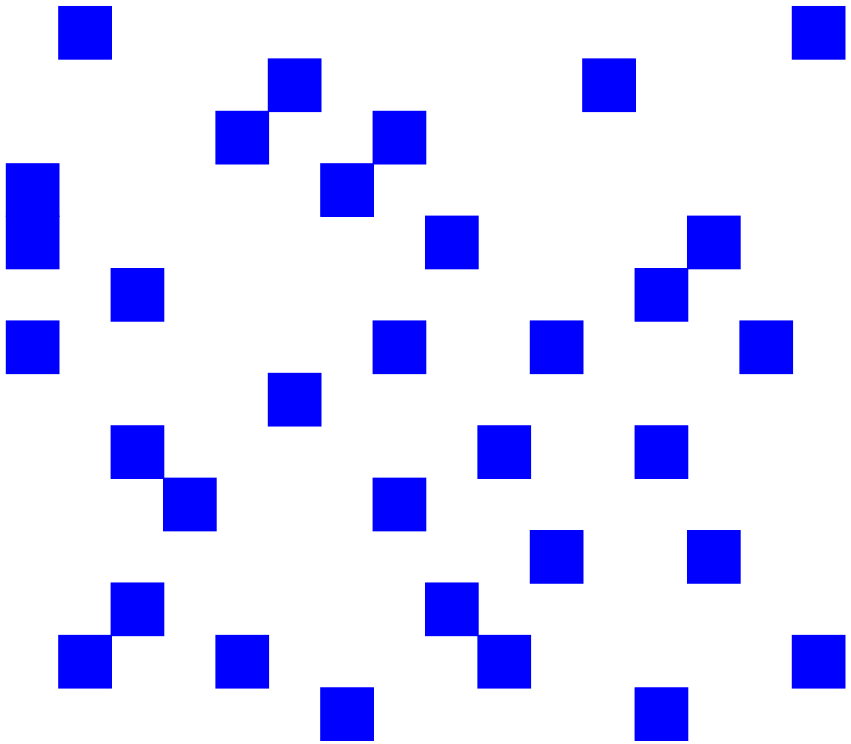
1. classical vector computing (ELLPACK/segmented scans),
2. SpMV register blocking (Blocked CRS, OSKI),
3. sparse blocking, and tiling.

This work

- generalises earlier approaches (1,2).
- illustrates sparse blocking and tiling (3) through the SpMV multiplication as well as the sparse matrix powers kernel.

Sequential SpMV

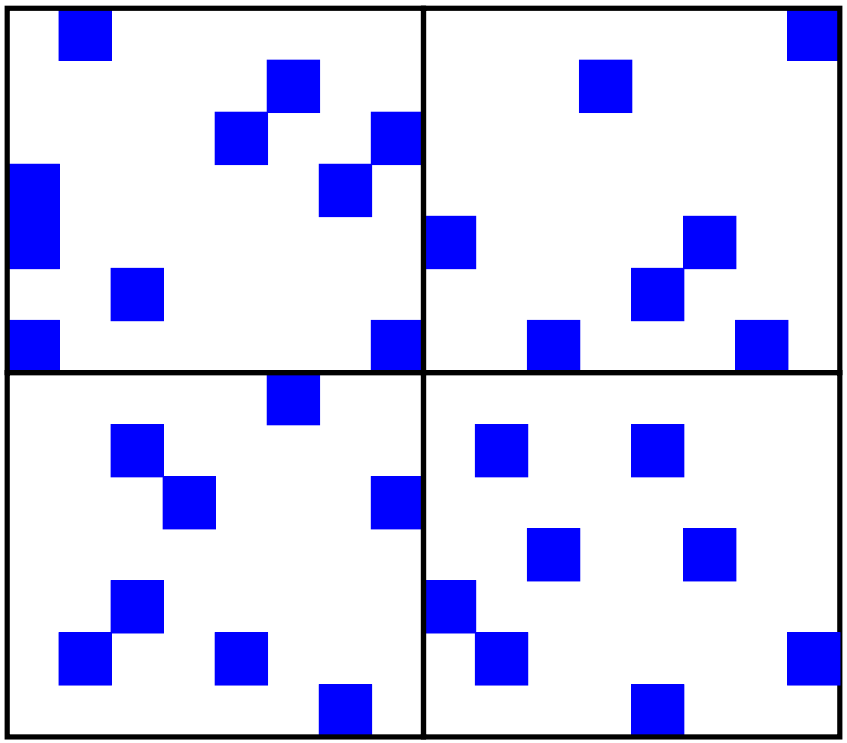
Blocking to fit subvectors into cache, **cache-obliviousness** to increase cache efficiency.



Ref.: Yzelman and Roose, “High-Level Strategies for Parallel Shared-Memory Sparse Matrix–Vector Multiplication”, IEEE Transactions on Parallel and Distributed Systems, doi: 10.1109/TPDS.2013.31 (2014).

Sequential SpMV

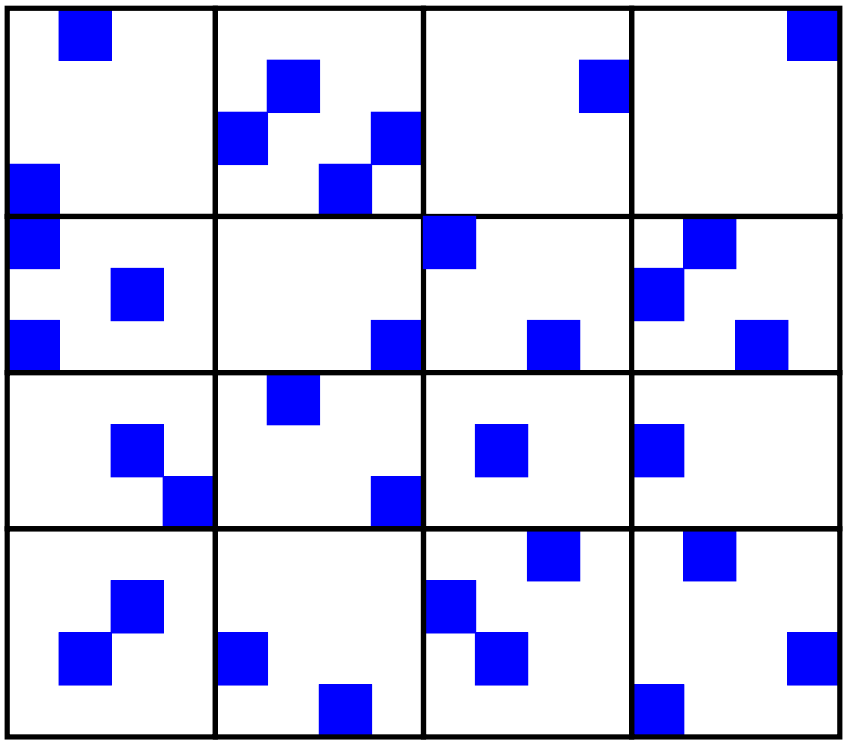
Blocking to fit subvectors into cache, **cache-obliviousness** to increase cache efficiency.



Ref.: Yzelman and Roose, “High-Level Strategies for Parallel Shared-Memory Sparse Matrix–Vector Multiplication”, IEEE Transactions on Parallel and Distributed Systems, doi: 10.1109/TPDS.2013.31 (2014).

Sequential SpMV

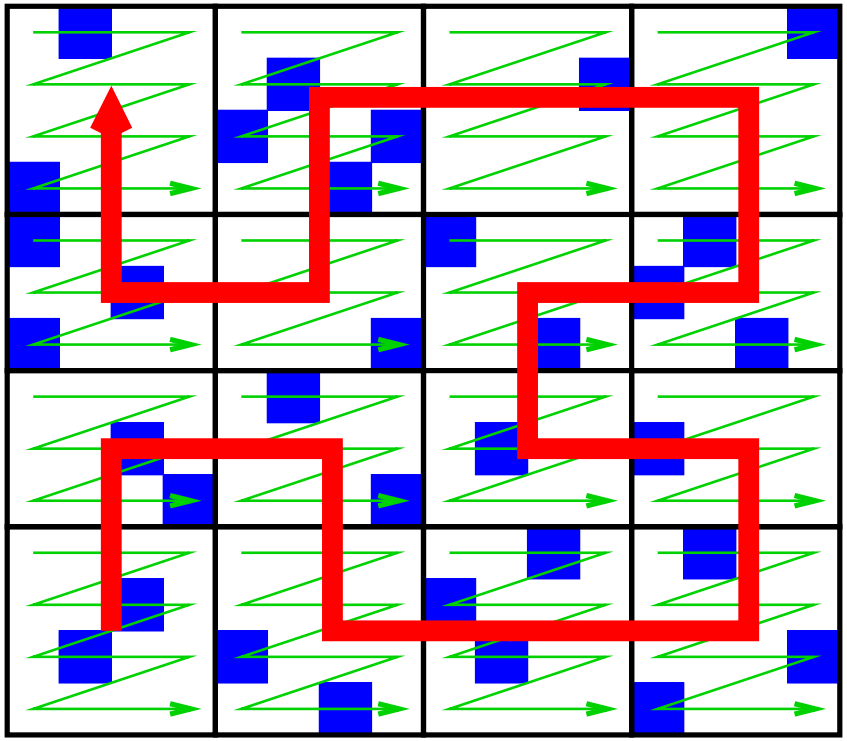
Blocking to fit subvectors into cache, **cache-obliviousness** to increase cache efficiency.



Ref.: Yzelman and Roose, “High-Level Strategies for Parallel Shared-Memory Sparse Matrix–Vector Multiplication”, IEEE Transactions on Parallel and Distributed Systems, doi: 10.1109/TPDS.2013.31 (2014).

Sequential SpMV

Blocking to fit subvectors into cache, **cache-obliviousness** to increase cache efficiency.



Ref.: Yzelman and Roose, “High-Level Strategies for Parallel Shared-Memory Sparse Matrix–Vector Multiplication”, IEEE Transactions on Parallel and Distributed Systems, doi: 10.1109/TPDS.2013.31 (2014).

SpMV vectorisation

What is needed for vectorisation:

- support for arbitrary nonzero traversals,
- handling of non-contiguous columns,
- handling of non-contiguous rows.

Basic operation using vector registers r_i:

$r_1 := r_1 + r_2 \cdot r_3$, the vectorised multiply-add.

How to get the right data in the vector registers?

- nonzeroes of A: steaming loads.
- elements from x, y: gather/scatter.

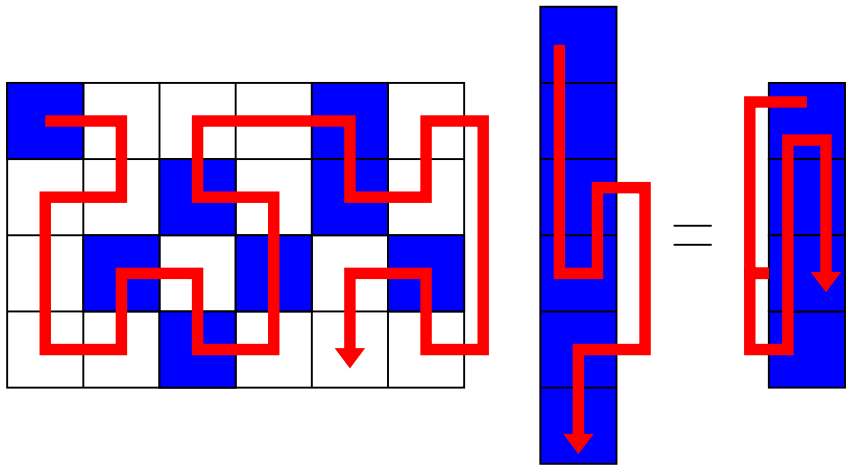
SpMV vectorisation

Streaming loads only apply to the sparse matrix data structure:

for $k = 0$ **to** $nz - 1$

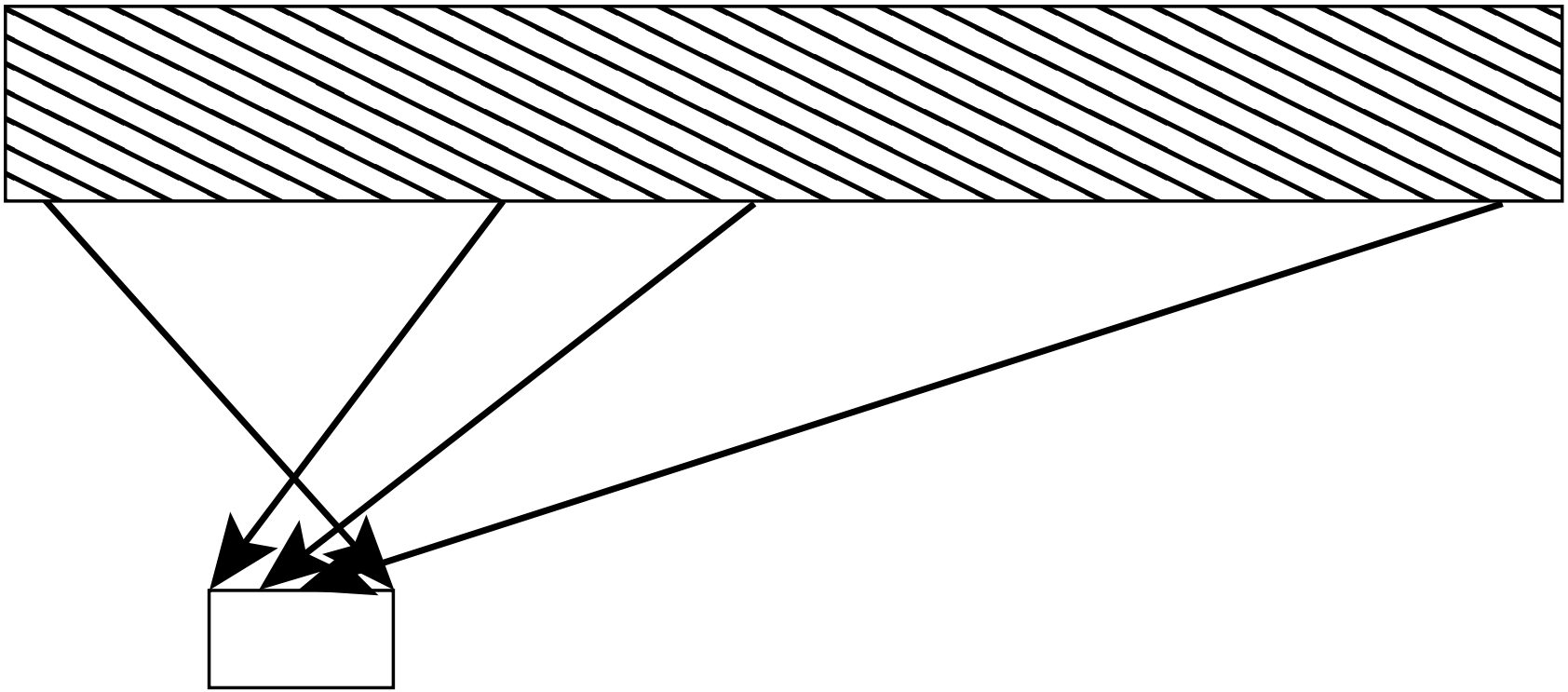
$$y_{i_k} := y_{i_k} + v_k \cdot x_{j_k}$$

Streaming loads in blue.



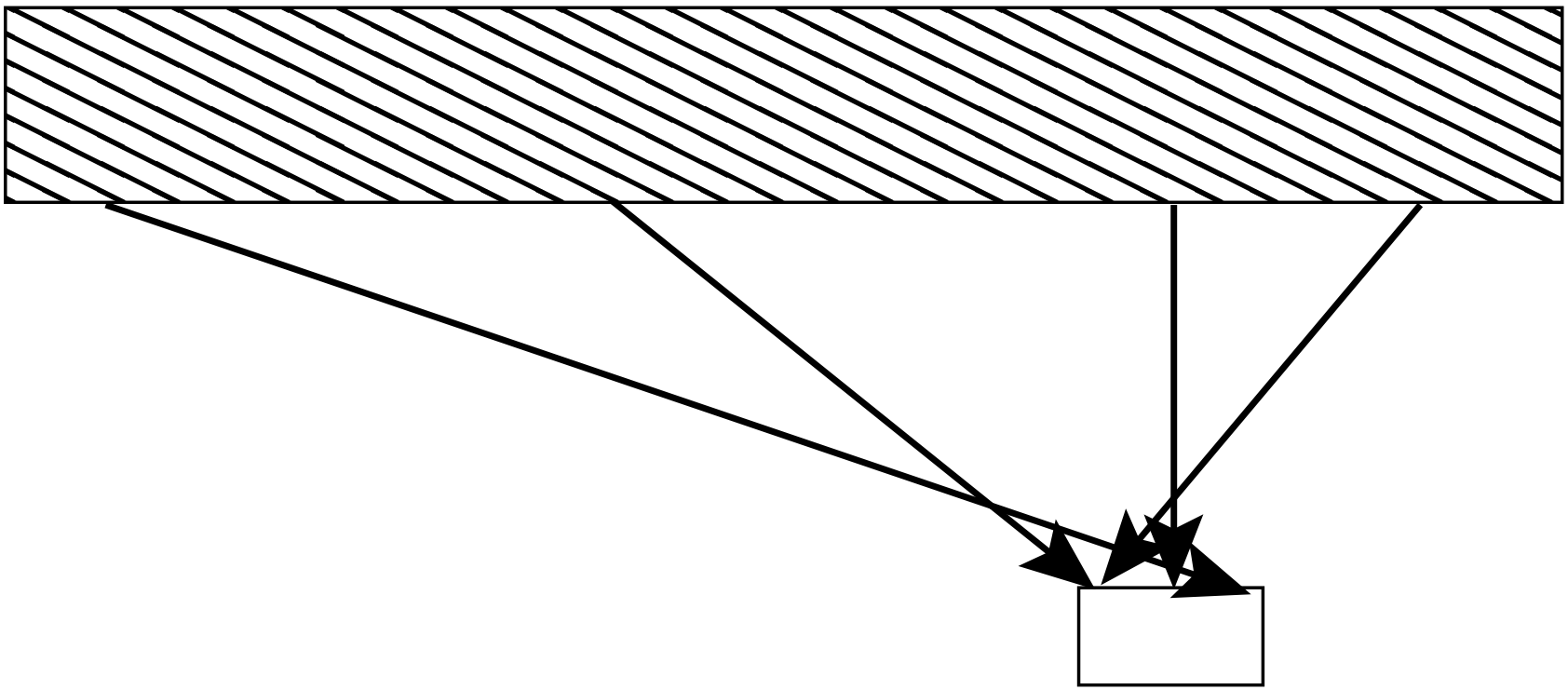
For accesses to x and y, alternatives are necessary.

SpMV vectorisation



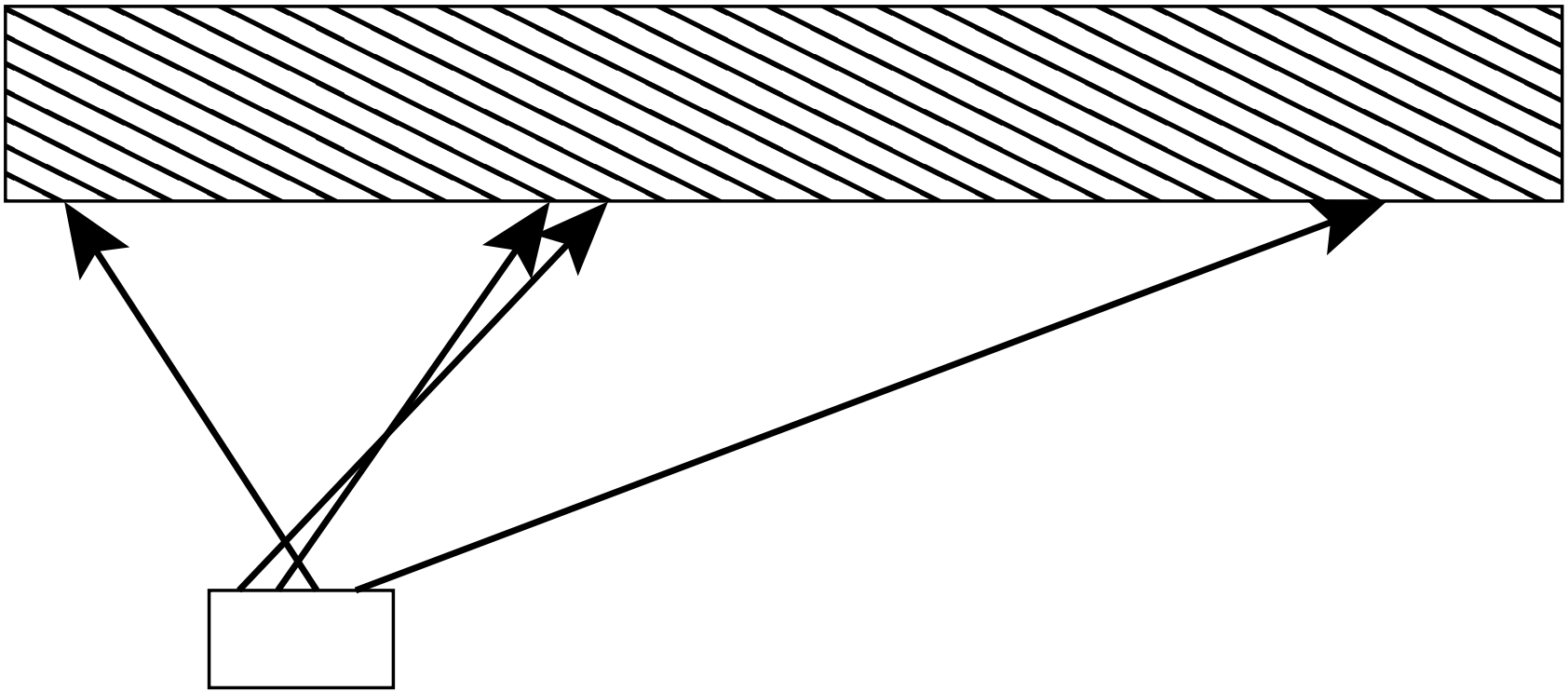
‘Gather’
read random memory areas into a single vector register

SpMV vectorisation



‘Gather’
read random memory areas into a single vector register

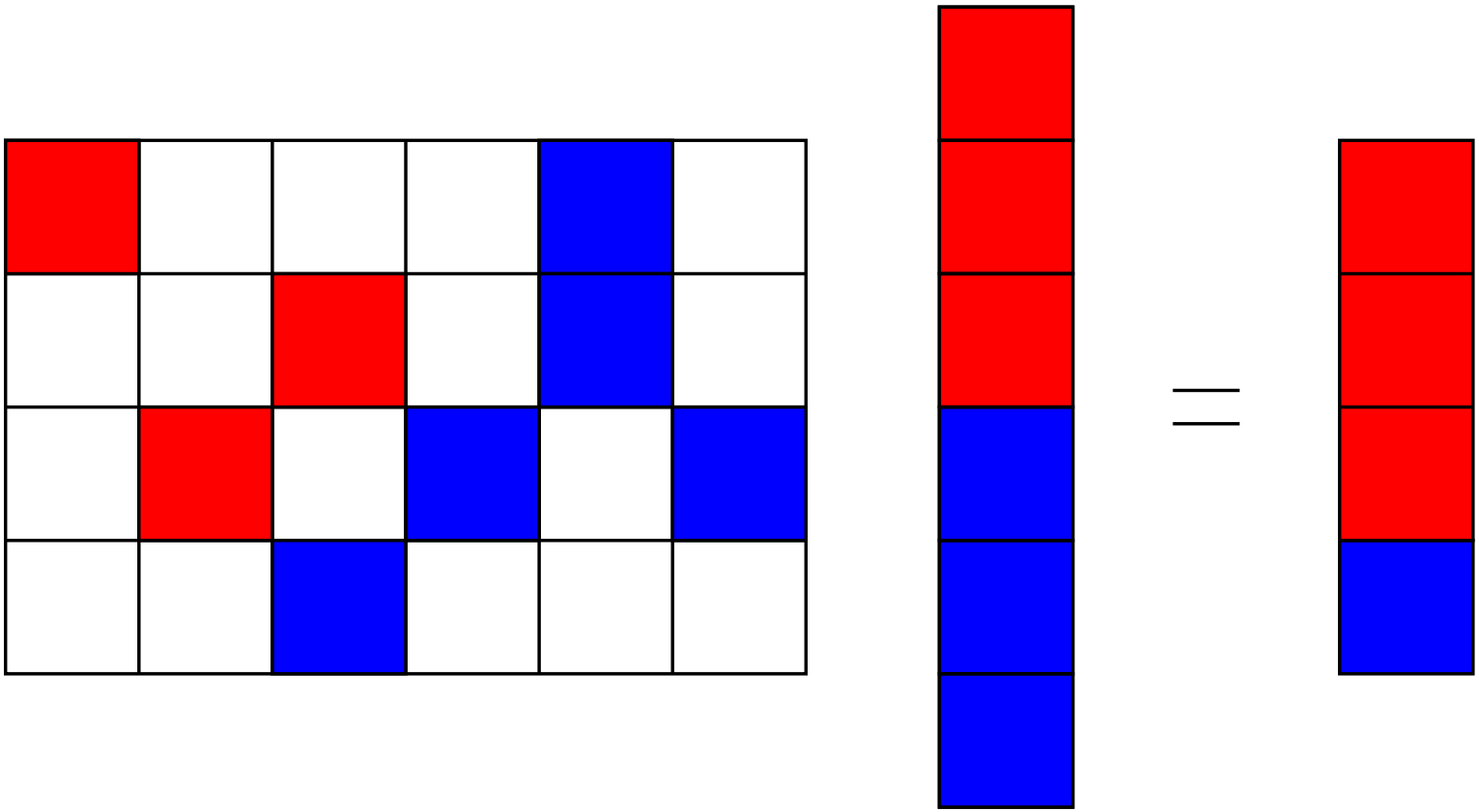
SpMV vectorisation



‘Scatter’
write back to random memory areas (inverse gather)

ELLPACK

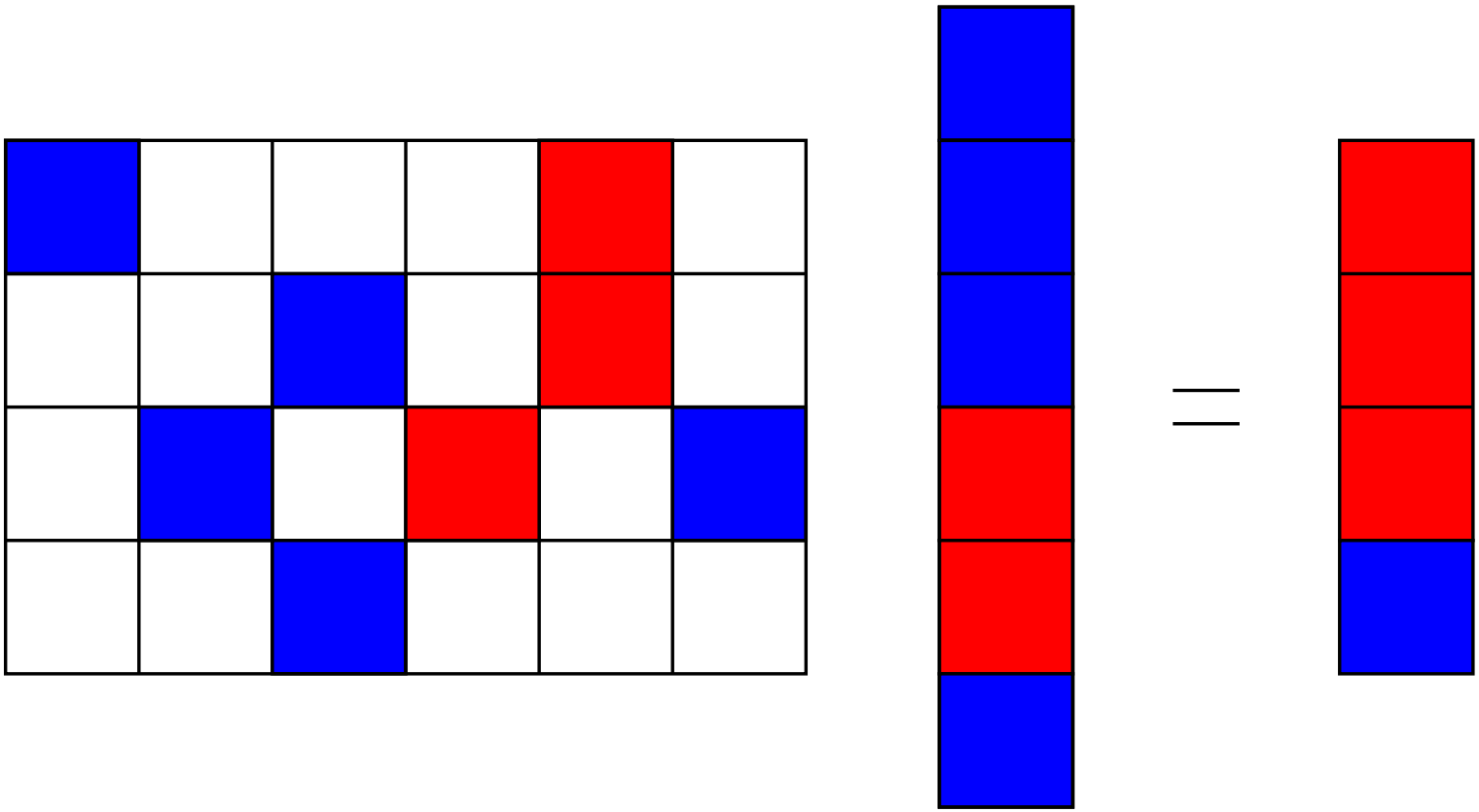
Sketch of the (sliced) ELLPACK SpMV multiply ($l = 3$):



Kincaid and Fassiotto, *ITPACK software and parallelization* in Advances in computer methods for partial differential equations, Proc. 7th Intl. Conf. on Computer Methods for Partial Differential Equations (1992).

ELLPACK

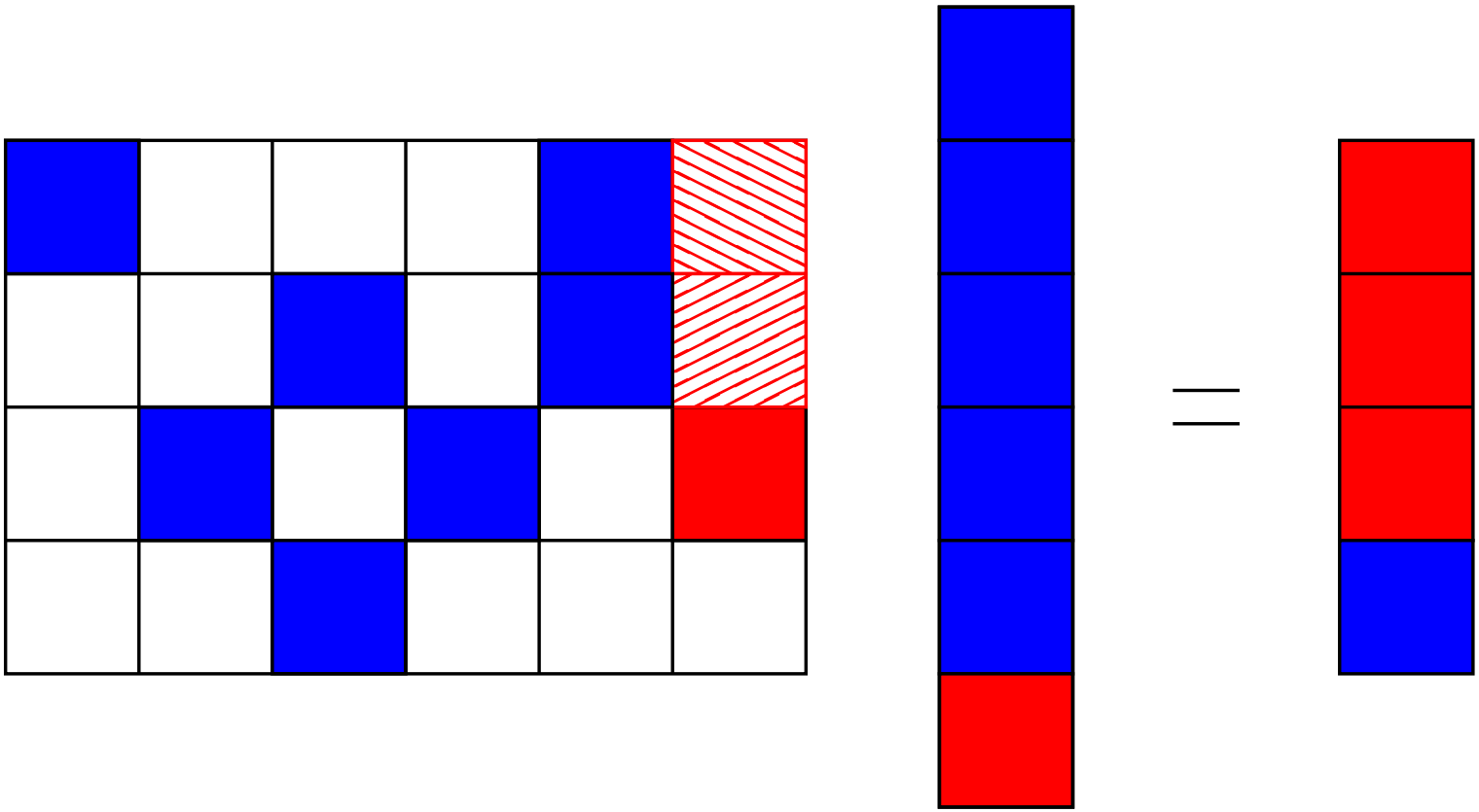
Sketch of the (sliced) ELLPACK SpMV multiply ($l = 3$):



Kincaid and Fassiotto, *ITPACK software and parallelization* in Advances in computer methods for partial differential equations, Proc. 7th Intl. Conf. on Computer Methods for Partial Differential Equations (1992).

ELLPACK

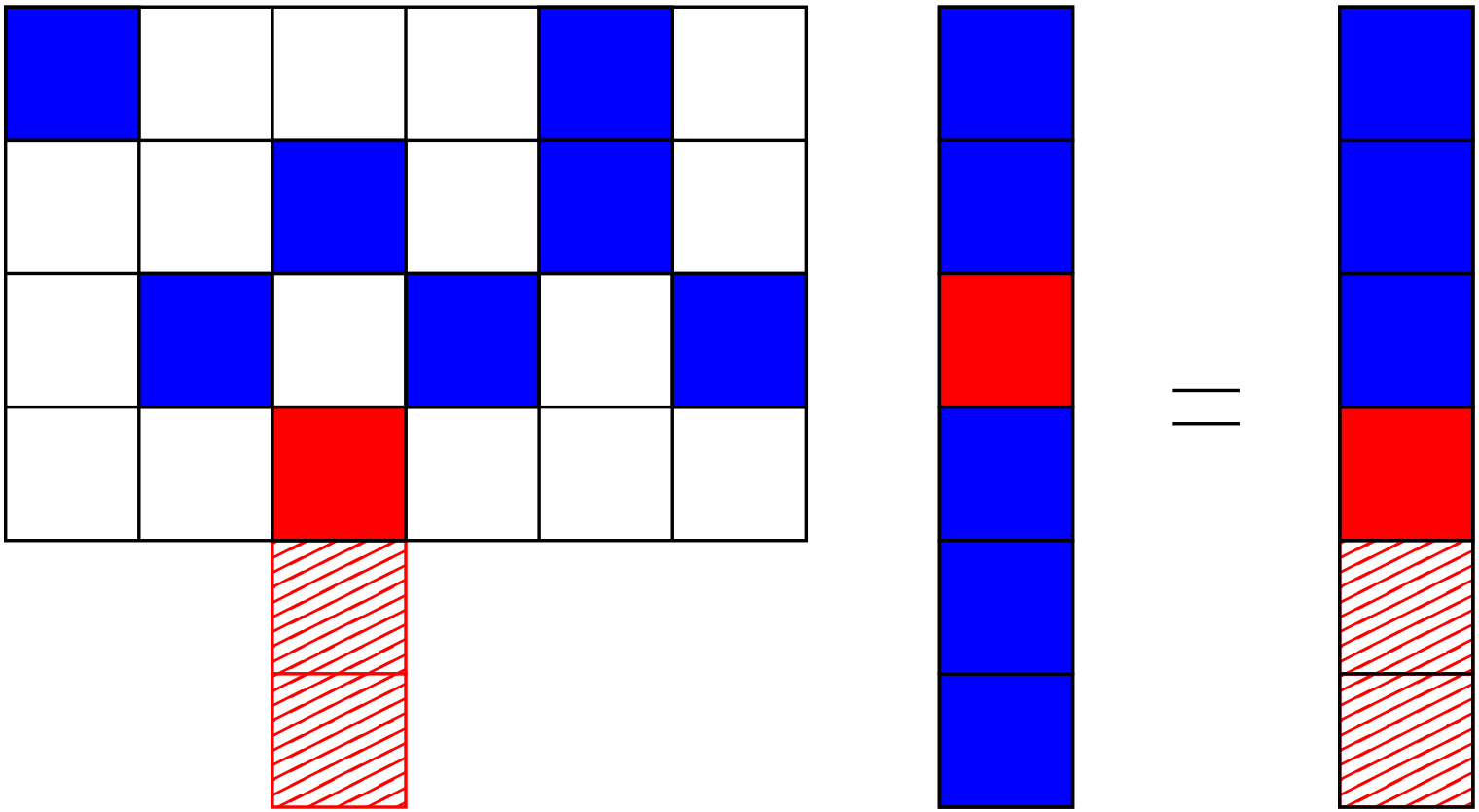
Sketch of the (sliced) ELLPACK SpMV multiply ($l = 3$):



Kincaid and Fassiotto, *ITPACK software and parallelization* in Advances in computer methods for partial differential equations, Proc. 7th Intl. Conf. on Computer Methods for Partial Differential Equations (1992).

ELLPACK

Sketch of the (sliced) ELLPACK SpMV multiply ($l = 3$):



Kincaid and Fassiotto, *ITPACK software and parallelization* in Advances in computer methods for partial differential equations, Proc. 7th Intl. Conf. on Computer Methods for Partial Differential Equations (1992).

Segmented scan/reduce

Sketch of a segmented-reduce enabled SpMV multiply:

$$j = (0,\ 4,\ 2,\ 4,\ 1,\ 3,\ 5,\ 2) \quad \rightarrow \quad (x_0,\ x_4,\ x_2\ x_4,\ x_1,\ x_3,\ x_5,\ x_2)$$

gather

$$\downarrow$$

$$v = (a_{00},\ a_{04},\ a_{13},\ \ldots,\ a_{32}) \quad \rightarrow \quad (x_0 a_{00},\ x_4 a_{04},\ x_2 a_{12},\ \ldots,\ x_2 a_{32})$$

vectorised multiply

$$\downarrow$$

summation control array $\rightarrow$

segmented reduce

$$\begin{pmatrix} x_0 a_{00} + x_4 a_{04} \\ x_2 a_{12} + x_4 a_{14} \\ x_1 a_{21} + x_3 a_{23} + x_5 a_{25} \\ x_5 a_{35} + x_3 a_{32} \end{pmatrix}$$

$$\downarrow$$

$$i = (0,\ 0,\ 1,\ 1,\ 2,\ 2,\ 2,\ 3) \quad \rightarrow \quad (y_0, y_1, y_2, y_3)$$

gather/load + vectorised summation

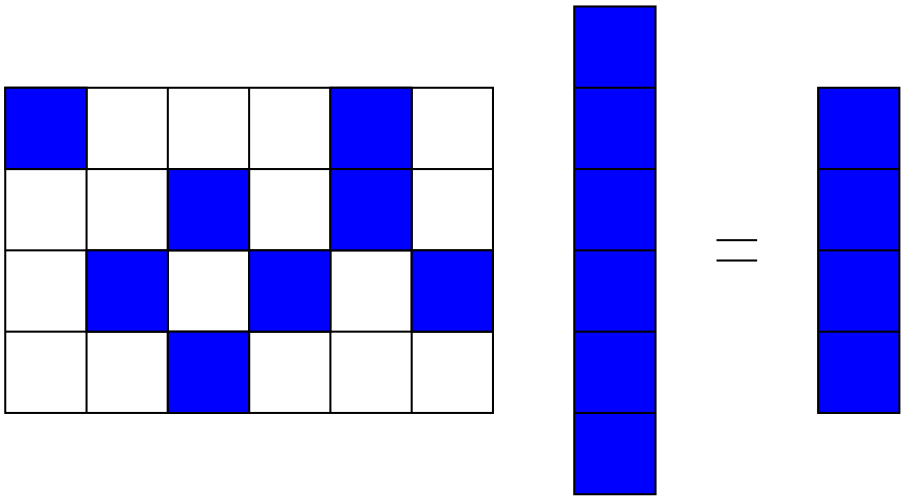
$$\downarrow \text{ scatter}$$

$$y$$

Blelloch, Heroux, Zagha, *Segmented operations for sparse matrix computation on vector multiprocessors* (1993).

BLOCKED CRS

The compressed row storage (CRS) format:



$$s = (0,\ 2,\ 4,\ 7,\ 8,\ 9)$$
$$j = (0,\ 4,\ 2,\ 4,\ 1,\ 3,\ 5,\ 2)$$
$$v = (a_{00},\ a_{04},\ \ldots,\ a_{32})$$

for $i = 0$ **to** $m - 1$

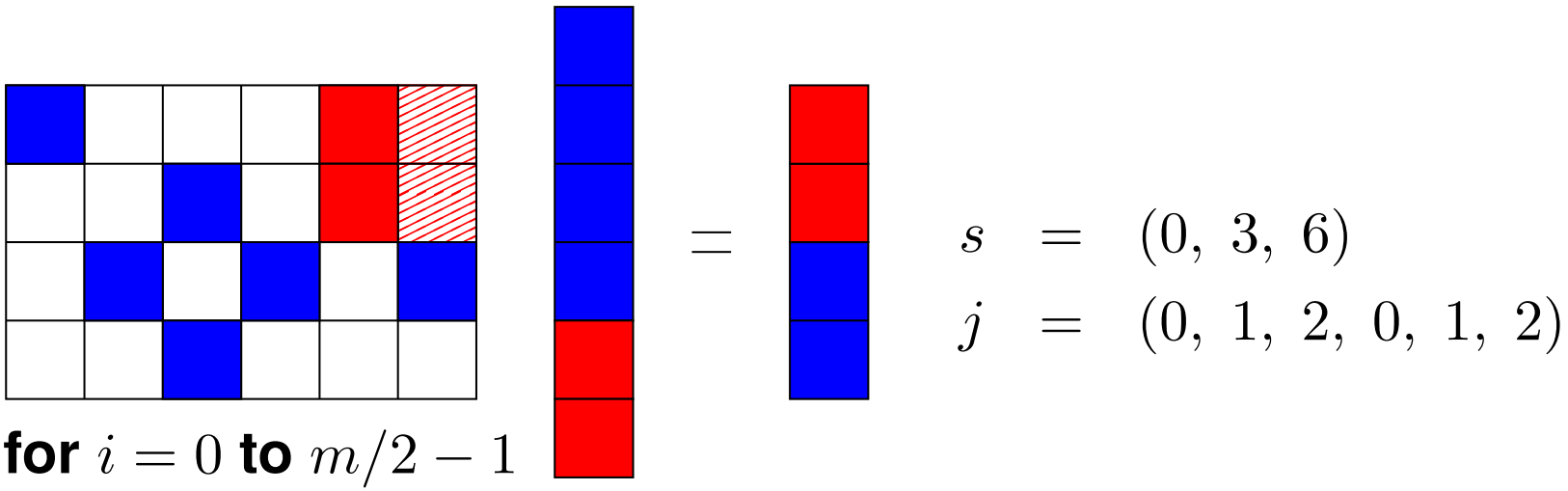
  for $k = s_i$ **to** $s_{i+1} - 1$

    $y_i := y_i + v_k \cdot x_{j_k}$

Eisenstat, Gursky, Schultz, Sherman, *Yale sparse matrix package II: the nonsymmetric codes*, Defense Technical Information Center, technical report (1977).

Blocked CRS

$$v = \left(\begin{pmatrix} a_{00} & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ a_{12} & 0 \end{pmatrix}, \begin{pmatrix} a_{04} & 0 \\ a_{14} & 0 \end{pmatrix}, V_3, V_4, V_5 \right)$$



$$s = (0,\ 3,\ 6)$$
$$j = (0,\ 1,\ 2,\ 0,\ 1,\ 2)$$

for $i = 0$ **to** $m/2 - 1$

for $k = s_i$ **to** $s_{i+1} - 1$

$$\begin{pmatrix} y_{2i} \\ y_{2i+1} \end{pmatrix} := \begin{pmatrix} y_{2i} \\ y_{2i+1} \end{pmatrix} + V_k \cdot \begin{pmatrix} x_{2j} \\ x_{2j+1} \end{pmatrix}$$

Pinar & Heath, *Improving performance of sparse matrix–vector multiplication* (1999);
Vuduc & Moon, *Fast sparse matrix–vector multiplication by exploiting variable block structure* (2005).

VECTORISED BICRS

while there are nonzeroes **do**

load nonzeroes into r_3 (load nonzero indices in r_4, r_5)

gather corresponding elements from x into r_2 (using r_4)

gather corresponding elements from y into r_1 (using r_5)

do vectorised multiply-add $r_1 := r_1 + r_2 \cdot r_3$

sum-reduce r_1 into unique elements from y

scatter unique elements from r_1 back to y

end while

Freedom to choose the number p of unique elements from y:

- choosing $p = 1$ leads to a **segmented-reduce** type SpMV,
- choosing $p = l$ leads to **sliced ELLPACK**.

Other choices lead to generic $p \times q$ blocks, reminiscent of Blocked CRS, with $pq = l$ the vector register width.

VECTORISED BICRS

while there are nonzeroes **do**

load nonzeroes into r_3 (load nonzero indices in r_4, r_5)

gather corresponding elements from x into r_2 (using r_4)

gather corresponding elements from y into r_1 (using r_5)

do vectorised multiply-add $r_1 := r_1 + r_2 \cdot r_3$

sum-reduce r_1 into unique elements from y

scatter unique elements from r_1 back to y

end while

Freedom to choose the number p of unique elements from y:

- choosing $p = 1$ leads to a **segmented-reduce** type SpMV,
- choosing $p = l$ leads to **sliced ELLPACK**.

Other choices lead to generic $p \times q$ blocks, reminiscent of Blocked CRS, with $pq = l$ the vector register width.

Vectorised BICRS

while there are nonzeroes **do**

load nonzeroes into r_3 (load nonzero indices in r_4, r_5)

gather corresponding elements from x into r_2 (using r_4)

gather corresponding elements from y into r_1 (using r_5)

do vectorised multiply-add $r_1 := r_1 + r_2 \cdot r_3$

sum-reduce r_1 into unique elements from y

scatter unique elements from r_1 back to y

end while

Freedom to choose the number p of unique elements from y:

- choosing $p = 1$ leads to a **segmented-reduce** type SpMV,
- choosing $p = l$ leads to **sliced ELLPACK**.

Other choices lead to generic $p \times q$ blocks, reminiscent of Blocked CRS, with $pq = l$ the vector register width.

Vectorised BICRS

while there are nonzeroes **do**

load nonzeroes into r_3 (load nonzero indices in r_4, r_5)
gather corresponding elements from x into r_2 (using r_4)
gather corresponding elements from y into r_1 (using r_5)
do vectorised multiply-add $r_1 := r_1 + r_2 \cdot r_3$
sum-reduce r_1 into unique elements from y
scatter unique elements from r_1 back to y

end while

Freedom to choose the number p of unique elements from y:

- choosing $p = 1$ leads to a **segmented-reduce** type SpMV,
- choosing $p = l$ leads to **sliced ELLPACK**.

Other choices lead to generic $p \times q$ blocks, reminiscent of Blocked CRS, with $pq = l$ the vector register width.

Vectorised BICRS

while there are nonzeroes **do**

load nonzeroes into r_3 (load nonzero indices in r_4, r_5)
gather corresponding elements from x into r_2 (using r_4)
gather corresponding elements from y into r_1 (using r_5)
do vectorised multiply-add $r_1 := r_1 + r_2 \cdot r_3$
sum-reduce r_1 into unique elements from y
scatter unique elements from r_1 back to y

end while

Freedom to choose the number p of unique elements from y:

- choosing $p = 1$ leads to a **segmented-reduce** type SpMV,
- choosing $p = l$ leads to **sliced ELLPACK**.

Other choices lead to generic $p \times q$ blocks, reminiscent of Blocked CRS, with $pq = l$ the vector register width.

VECTORISED BICRS

while there are nonzeroes **do**
load nonzeroes into r_3 (load nonzero indices in r_4, r_5)
gather corresponding elements from x into r_2 (using r_4)
gather corresponding elements from y into r_1 (using r_5)
do vectorised multiply-add $r_1 := r_1 + r_2 \cdot r_3$
sum-reduce r_1 into unique elements from y
scatter unique elements from r_1 back to y
end while

Freedom to choose the number p of unique elements from y:

- choosing $p = 1$ leads to a **segmented-reduce** type SpMV,
- choosing $p = l$ leads to **sliced ELLPACK**.

Other choices lead to generic $p \times q$ blocks, reminiscent of Blocked CRS, with $pq = l$ the vector register width.

Vectorised BICRS

while there are nonzeroes **do**
load nonzeroes into r_3 (load nonzero indices in r_4, r_5)
gather corresponding elements from x into r_2 (using r_4)
gather corresponding elements from y into r_1 (using r_5)
do vectorised multiply-add $r_1 := r_1 + r_2 \cdot r_3$
sum-reduce r_1 into unique elements from y
scatter unique elements from r_1 back to y
end while

Freedom to choose the number p of unique elements from y:

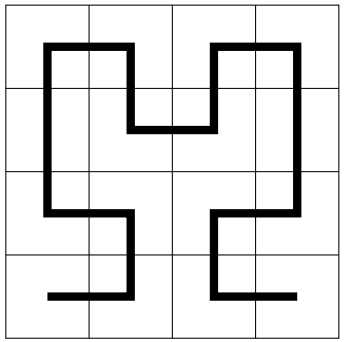
- choosing $p = 1$ leads to a **segmented-reduce** type SpMV,
- choosing $p = l$ leads to **sliced ELLPACK**.

Other choices lead to generic $p \times q$ blocks, reminiscent of Blocked CRS, with $pq = l$ the vector register width.

VECTORISED BICRS

Example multiplication using 2×2 blocking, #1

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



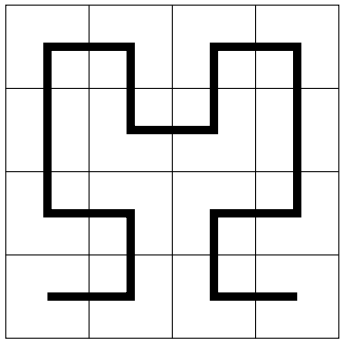
$$A = \begin{cases} V & [7\ 1\ 0\ 0\ /\ 4\ 1\ 2\ 0\ /\ 3\ 3\ 0\ 0\ /\ 2\ 0\ 1\ 1\ ] \\ \tilde{J} & [0\ 0\ 0\ 0\ /\ 4\ 1\ 2\ 0\ /\ 2\ 1\ 0\ 0\ /\ 5\ 0\ -1\ 0\ ] \\ \tilde{I} & [3\ 2\ /\ -3\ -1\ /\ 2\ 1\ ] \end{cases}$$

- Multiply $(7, 1, 0, 0)$ with $(x_0, x_0, 0, 0)$ and add to (y_3, y_2).

VECTORISED BICRS

Example multiplication using 2×2 blocking, #2

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



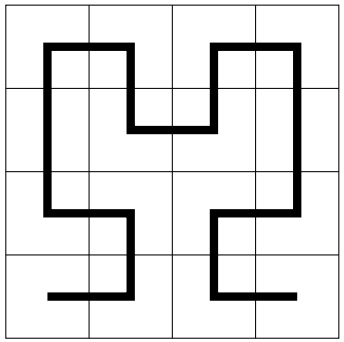
$$A = \begin{cases} V & [7\ 1\ 0\ 0\ /\ 4\ 1\ 2\ 0\ /\ 3\ 3\ 0\ 0\ /\ 2\ 0\ 1\ 1\ ] \\ \tilde{J} & [0\ 0\ 0\ 0\ /\ 4\ 1\ 2\ 0\ /\ 2\ 1\ 0\ 0\ /\ 5\ 0\ -1\ 0\ ] \\ \tilde{I} & [3\ 2\ /\ -3\ -1\ /\ 2\ 1\ ] \end{cases}$$

- Multiply $(4, 1, 2, 0)$ with $(x_0, x_1, x_2, 0)$ and add to (y_0, y_1).

VECTORISED BICRS

Example multiplication using 2×2 blocking, #3

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



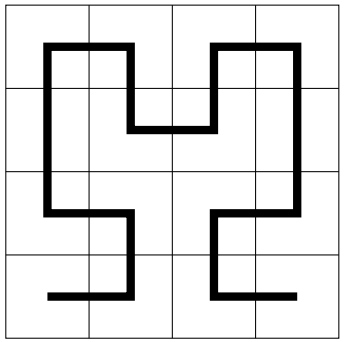
$$A = \begin{cases} V & [7\ 1\ 0\ 0\ /\ 4\ 1\ 2\ 0\ /\ 3\ 3\ 0\ 0\ /\ 2\ 0\ 1\ 1\ ] \\ \tilde{J} & [0\ 0\ 0\ 0\ /\ 4\ 1\ 2\ 0\ /\ 2\ 1\ 0\ 0\ /\ 5\ 0\ -1\ 0\ ] \\ \tilde{I} & [3\ 2\ /\ -3\ -1\ /\ 2\ 1\ ] \end{cases}$$

- Multiply $(3, 3, 0, 0)$ with $(x_2, x_3, 0, 0)$ and add to (y_0, y_1).

VECTORISED BICRS

Example multiplication using 2×2 blocking, #4

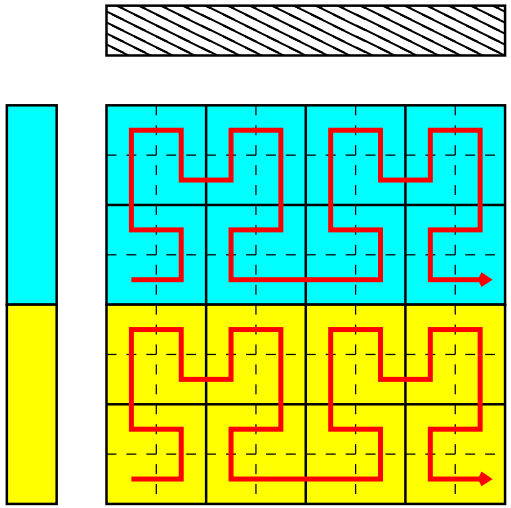
$$A = \begin{pmatrix} 4 & 1 & 3 & 0 \\ 0 & 0 & 2 & 3 \\ 1 & 0 & 0 & 2 \\ 7 & 0 & 1 & 1 \end{pmatrix}$$



$$A = \begin{cases} V & [7\ 1\ 0\ 0\ /\ 4\ 1\ 2\ 0\ /\ 3\ 3\ 0\ 0\ /\ 2\ 0\ 1\ 1\ ] \\ \tilde{J} & [0\ 0\ 0\ 0\ /\ 4\ 1\ 2\ 0\ /\ 2\ 1\ 0\ 0\ /\ 5\ 0\ -1\ 0\ ] \\ \tilde{I} & [3\ 2\ /\ -3\ -1\ /\ 2\ 1\ ] \end{cases}$$

- Multiply $(2, 0, 1, 1)$ with $(x_3, 0, x_2, x_3)$ and add to (y_2, y_3).

SpMV parallelisation

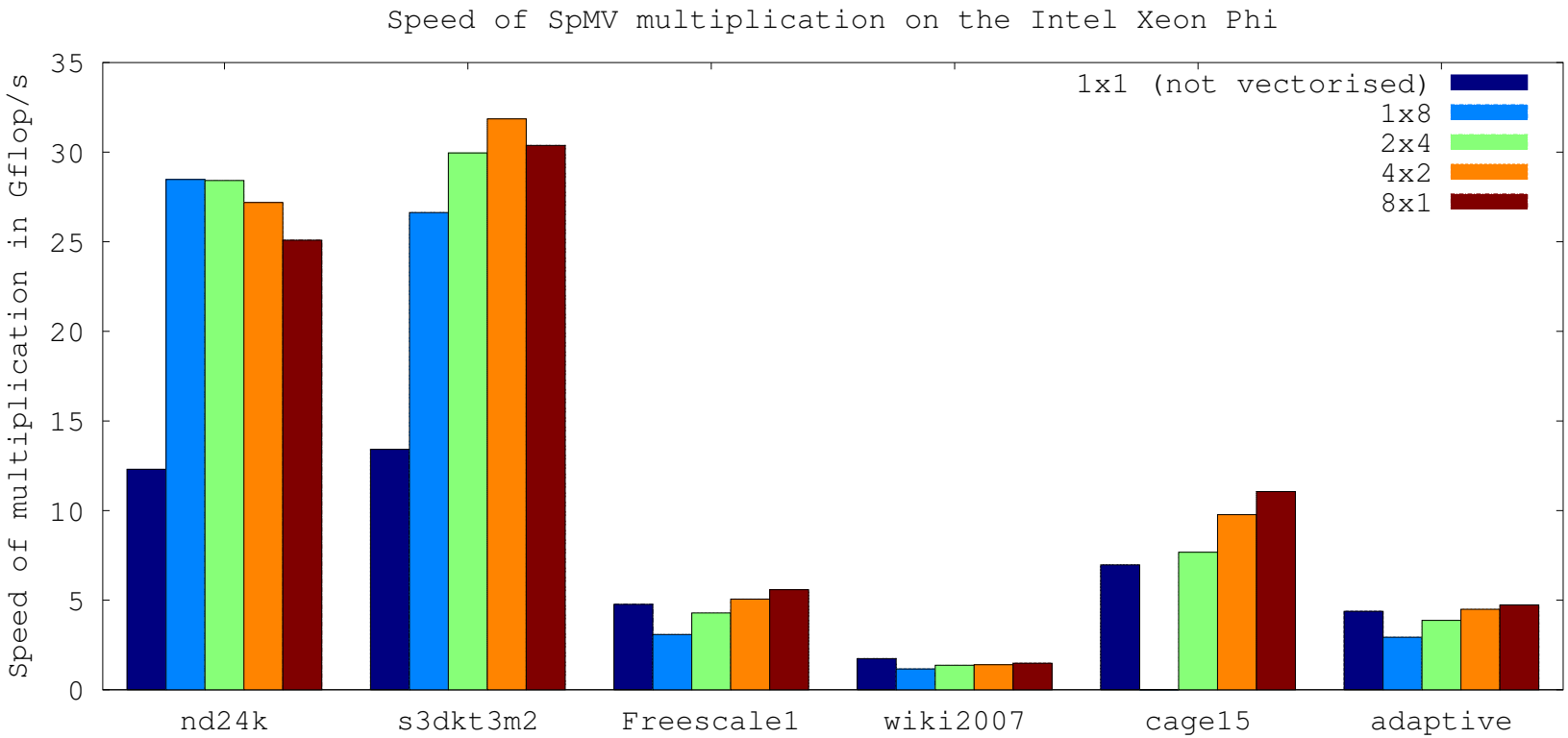


Implementation on the Intel Xeon Phi:

- load-balanced row distribution over 240 threads;
- each thread locally uses Hilbert-ordered sparse blocks;
- nonzeroes within blocks stored using vectorised BICRS;
- 1×8, 2×4, 4×2, and 8×1 block sizes.

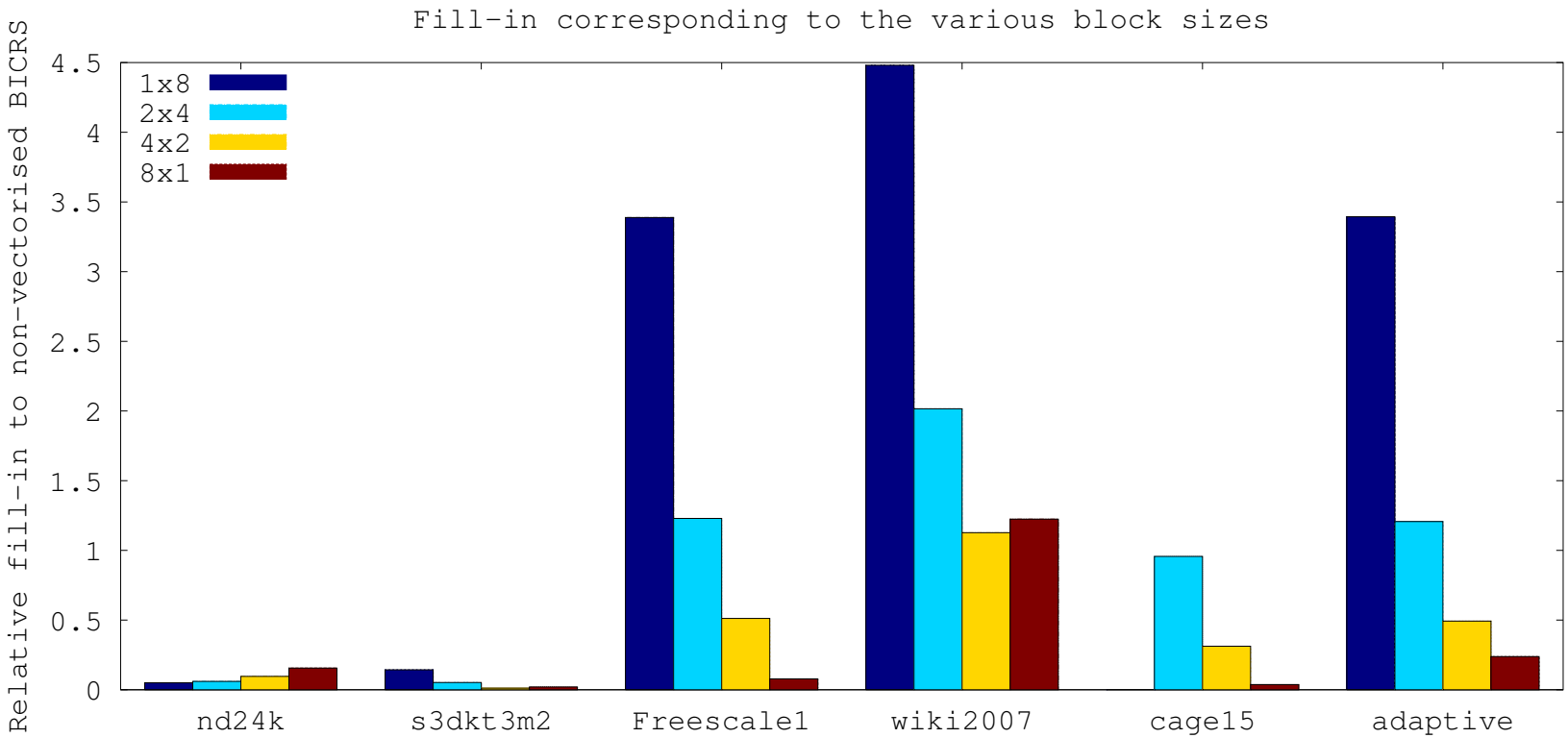
Yzelman and Roose, “High-Level Strategies for Parallel Shared-Memory Sparse Matrix–Vector Multiplication”, IEEE Trans. Parallel and Distributed Systems, doi: 10.1109/TPDS.2013.31 (2014).

SpMV results



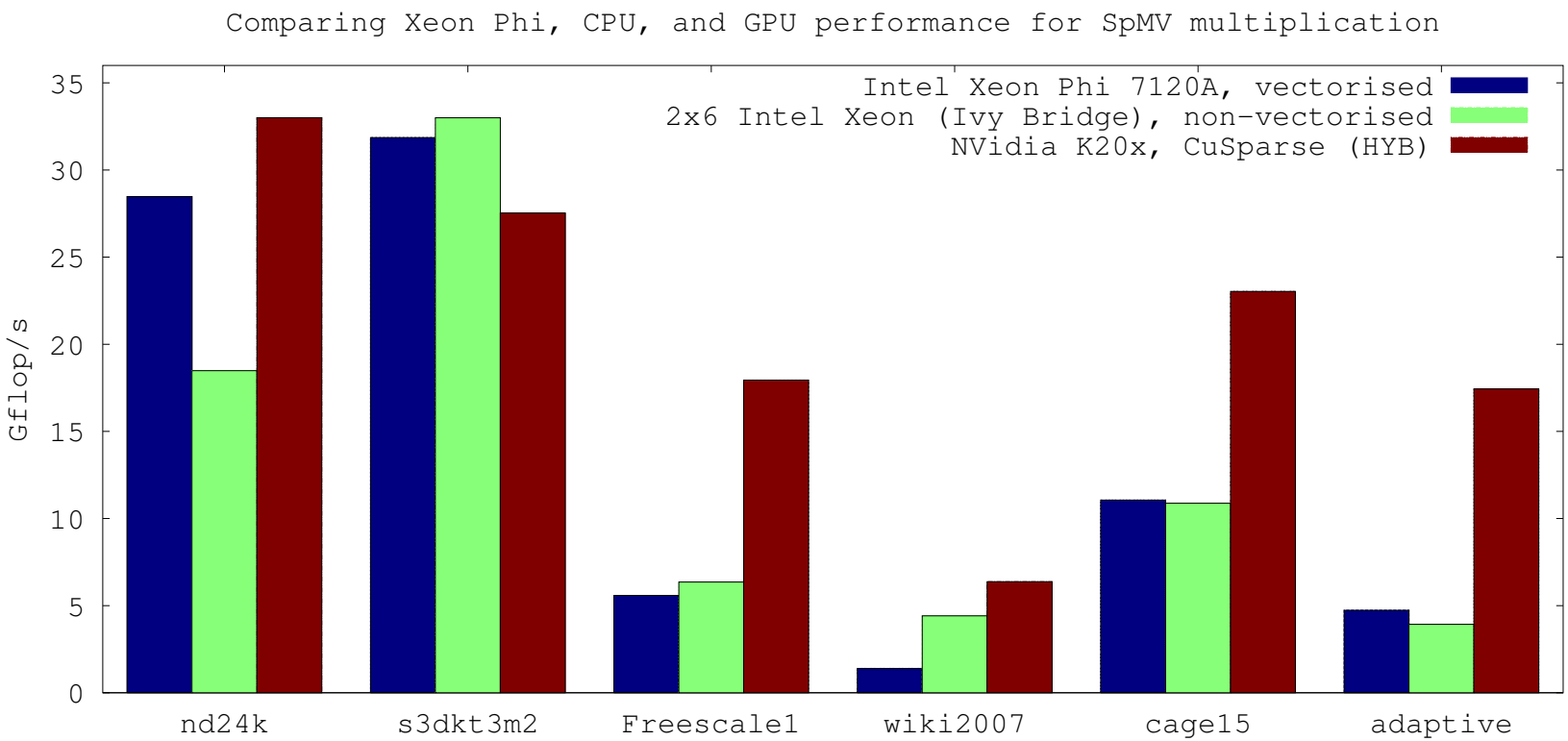
Results of the partially distributed parallel SpMV multiplication using vectorised BICRS on an Intel Xeon Phi 7120A.

SpMV results



In general, blocking sizes that result in the least fill-in achieve the highest performance.

SpMV results



Comparing vectorised, non-vectorised, and a competing approach on the Xeon Phi, a dual-socket machine, and a GPU.

SpMV results

	Structured	Unstructured	Average
Intel Xeon Phi	21.6	8.7	15.2
2x Ivy Bridge CPU	23.5	14.6	19.0
NVIDIA K20X GPU	16.7	13.3	15.0

Aggregate results over 24 matrices. Generalised statements:

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

But mostly:

no one solution fits all.

Conclusions

Optimised the BICRS data structure that

- fully exploits vector operations through gather/scatter,
- controls fill-in through block size selection,
- supports arbitrary nonzero traversals, while
- retaining all opportunities for compression.

The new approach generalises several earlier approaches.

`http://albert-jan.yzelman.net/software#SL`