Towards Stationary Iterative Solvers with Adaptive Precision on FPGAs

Germán León, Rafael Mayo, Enrique S. Quintana-Ortí

UNIVERSITAT JAUME•I

Motivation

Jacobi method for the solution of a sparse linear system

Given $Ax = b$, where $A \in \mathbb{R}^{n\times n}$ is a sparse matrix

$$
\begin{aligned}
x^{\{k\}} &= D^{-1}(b-(A-D)x^{\{k-1\}} = \\
&= D^{-1}b + Mx^{\{k-1\}}, \quad k = 1,2,\cdots
\end{aligned}
$$

Motivation

Jacobi method for the solution of a sparse linear system

Given $Ax = b$, where $A \in \mathbb{R}^{n \times n}$ is a sparse matrix

$$x^{\{k\}} = D^{-1}(b - (A - D)x^{\{k-1\}} =$$
$$= D^{-1}b + Mx^{\{k-1\}}, \quad k = 1,2,\cdots$$

$$D = diag(A)$$

Motivation

Jacobi method for the solution of a sparse linear system

Given $Ax = b$, where $A \in \mathbb{R}^{n \times n}$ is a sparse matrix

$$x^{\{k\}} = D^{-1}(b - (A - D)x^{\{k-1\}} =$$
$$= D^{-1}b + Mx^{\{k-1\}}, \quad k = 1,2, \cdots$$

This iteration basically requires a SpMV

$Mx^{\{k-1\}}$

Motivation

Jacobi method for the solution of a sparse linear system

Given $Ax = b$, where $A \in \mathbb{R}^{n\times n}$ is a sparse matrix

$$x^{\{k\}} = D^{-1}(b - (A - D)x^{\{k-1\}} =$$
$$= D^{-1}b + Mx^{\{k-1\}}, \quad k = 1,2,\cdots$$

It's possible to use an adaptive precision solver with different mantissa width

"Adaptive Precision Solvers for Sparse Linear Systems" H. Anzt, J. Dongarra and E.S. Quintana-Ortí. 3rd International Workshop on Energy Efficient Supercomputing. 2015

Motivation

Jacobi method for the solution of a linear system

Given $Ax = b$, where $A \in \mathbb{R}^{n \times n}$ is a sparse matrix

$$
\begin{aligned}
x^{\{k\}} &= D^{-1}(b - (A - D)x^{\{k-1\}} = \\
&= D^{-1}b + Mx^{\{k-1\}}, \quad k = 1,2,\cdots
\end{aligned}
$$

We test the development of a SpMV operator on a FPGA with different mantissa width

SpMV. Data structure

$$\begin{bmatrix} m_{0,0} & m_{0,1} & & m_{0,3} & m_{0,4} \\ & & m_{1,2} & & \\ & & & m_{0,0} & \\ m_{3,0} & & m_{3,2} & & \\ & m_{4,1} & m_{4,2} & & m_{4,4} \end{bmatrix}$$

Elements	$m_{0,0}$	$m_{0,1}$	$m_{0,3}$	$m_{0,4}$	0	0	$m_{1,2}$	0	0	$m_{2,3}$	0	0	$m_{3,0}$	$m_{3,2}$	0	$m_{4,1}$	$m_{4,2}$	$m_{4,4}$
Indices	0	1	3	4	4	-4	2	2	-2	3	3	-3	0	2	-2	1	2	-4

SpMV. Data structure

$$\begin{bmatrix} m_{0,0} & m_{0,1} & & m_{0,3} & m_{0,4} \\ & & m_{1,2} & & \\ & & & m_{0,0} & \\ m_{3,0} & & m_{3,2} & & \\ & m_{4,1} & m_{4,2} & & m_{4,4} \end{bmatrix}$$

Elements	$m_{0,0}$	$m_{0,1}$	$m_{0,3}$	$m_{0,4}$	0	0	$m_{1,2}$	0	0	$m_{2,3}$	0	0	$m_{3,0}$	$m_{3,2}$	0	$m_{4,1}$	$m_{4,2}$	$m_{4,4}$
Indices	0	1	3	4	4	-4	2	2	-2	3	3	-3	0	2	-2	1	2	-4

Blocks with constant number of elements (in our implementation 8 elments per block)

SpMV. Data structure

$$\begin{bmatrix} m_{0,0} & m_{0,1} & & m_{0,3} & m_{0,4} \\ & & m_{1,2} & & \\ & & & m_{0,0} & \\ m_{3,0} & & m_{3,2} & & \\ & m_{4,1} & m_{4,2} & & m_{4,4} \end{bmatrix}$$

Elements	$m_{0,0}$	$m_{0,1}$	$m_{0,3}$	$m_{0,4}$	0	0	$m_{1,2}$	0	0	$m_{2,3}$	0	0	$m_{3,0}$	$m_{3,2}$	0	$m_{4,1}$	$m_{4,2}$	$m_{4,4}$
Indices	0	1	3	4	4	-4	2	2	-2	3	3	-3	0	2	-2	1	2	-4

Padding elements, each row occupies completely one or more blocks

SpMV. Data structure

$$\begin{bmatrix} m_{0,0} & m_{0,1} & & m_{0,3} & m_{0,4} \\ & & m_{1,2} & & \\ & & & m_{0,0} & \\ m_{3,0} & & m_{3,2} & & \\ & m_{4,1} & m_{4,2} & & m_{4,4} \end{bmatrix}$$

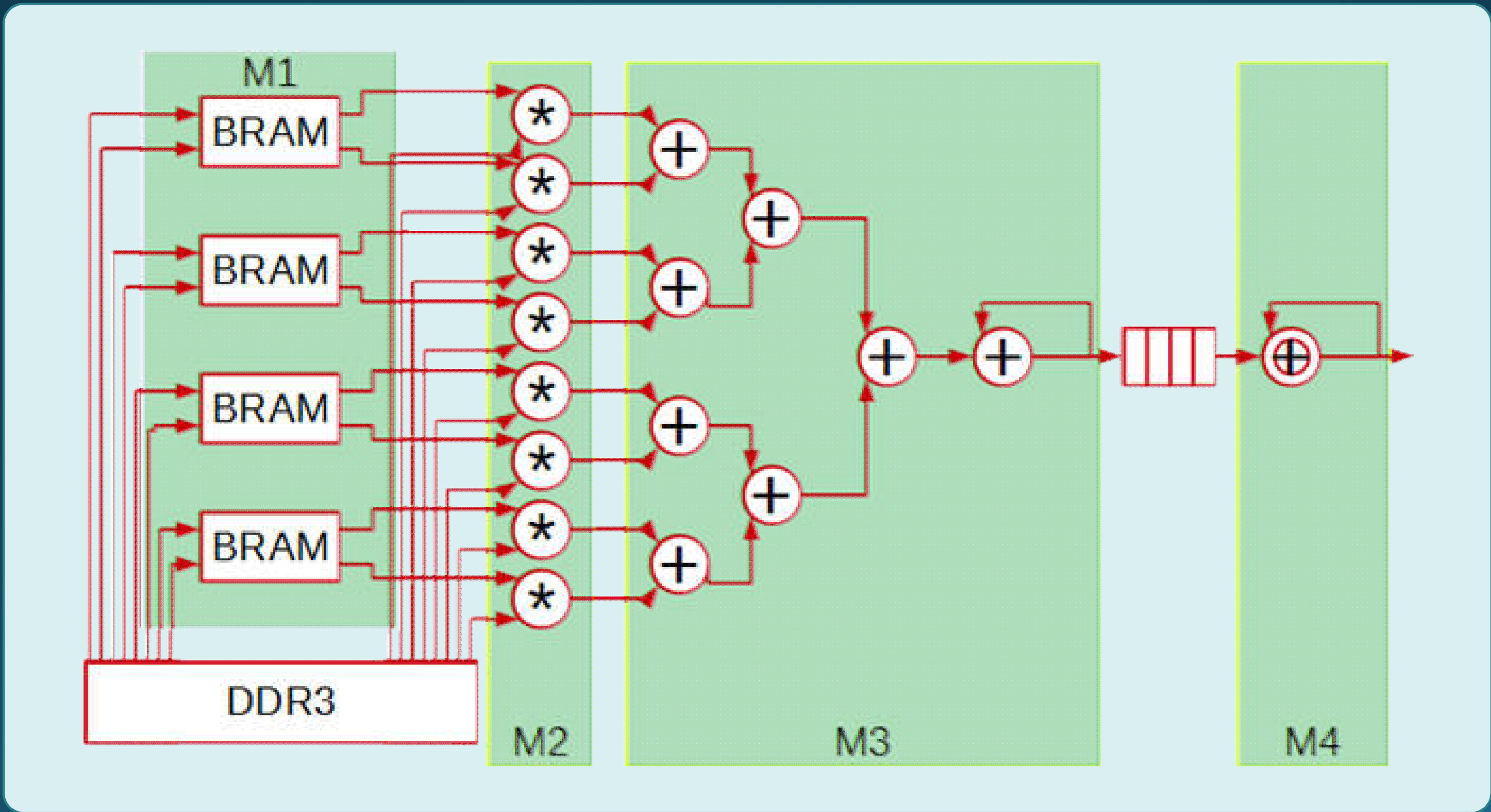
Elements	$m_{0,0}$	$m_{0,1}$	$m_{0,3}$	$m_{0,4}$	0	0	$m_{1,2}$	0	0	$m_{2,3}$	0	0	$m_{3,0}$	$m_{3,2}$	0	$m_{4,1}$	$m_{4,2}$	$m_{4,4}$
Indices	0	1	3	4	4	-4	2	2	-2	3	3	-3	0	2	-2	1	2	-4

Negative indeces mark the end of row

SpMV. Algorithm

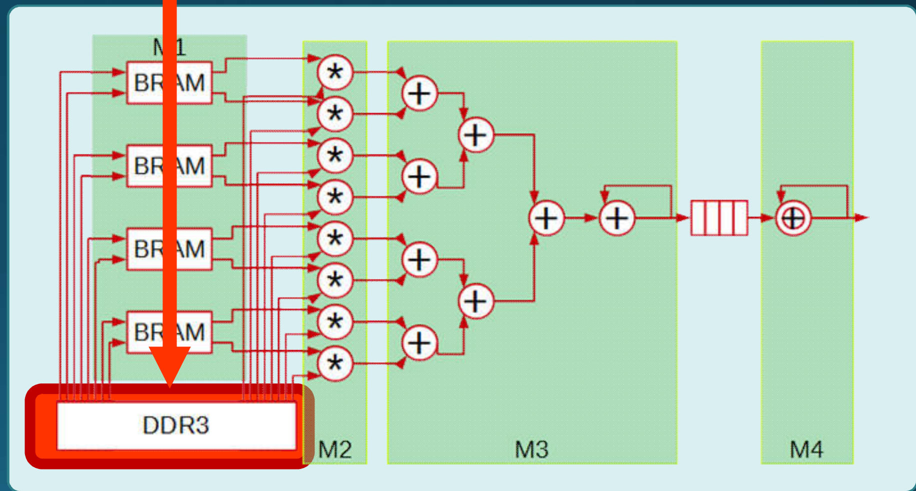
```
j =0;
for ( i = 0; i < m; i++ ) {
   end_row = 0;
   while ( ! end_row ) {
      e0 = M[j]   * x[col[j]];
      e1 = M[j+1] * x[col[j+1]];
      ...
      e6 = M[j+6] * x[col[j+6]];
      if ( end_row = (col[j+7]<0) )
         e7 = M[j+7] * x[-col[j+7]];
      else
         e7 = M[j+7] * x[ col[j+7]];
      x[i] += e0 + e1 + ... + e7;
      j += 8;
} }
```

SpMV. Schema of the operator



SpMV. Schema of the operator

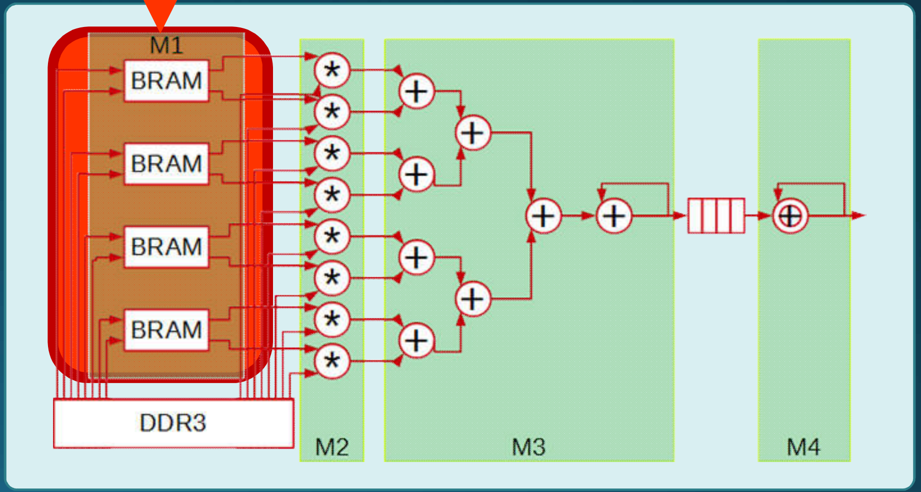
```
j =0;
for ( i = 0; i < m; i++ ) {
   end_row = 0;
   while ( ! end_row ) {
      e0 = M[j]   * x[col[j]];
      e1 = M[j+1] * x[col[j+1]];
      ...
      e6 = M[j+6] * x[col[j+6]];
      if ( end_row = (col[j+7]<0) )
      e7 = M[j+7] * x[-col[j+7]];
      else
      e7 = M[j+7] * x[ col[j+7]];
      x[i] += e0 + e1 + ... + e7;
      j += 8;
} }
```



Elements	$m_{0,0}$	$m_{0,1}$	$m_{0,3}$	$m_{0,4}$	0	0	$m_{1,2}$	0	0	$m_{2,3}$	0	0	$m_{3,0}$	$m_{3,2}$	0	$m_{4,1}$	$m_{4,2}$	$m_{4,4}$
Indices	0	1	3	4	4	-4	2	2	-2	3	3	-3	0	2	-2	1	2	-4

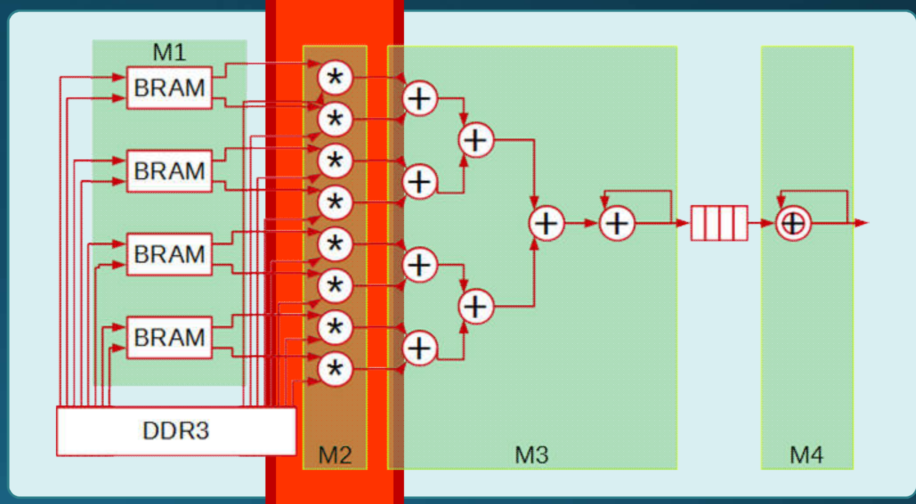
SpMV. Schema of the operator

```
j =0;
for ( i = 0; i < m; i++ ) {
   end_row = 0;
   while ( ! end_row ) {
      e0 = M[j]   * x[col[j]];
      e1 = M[j+1] * x[col[j+1]];
      ...
      e6 = M[j+6] * x[col[j+6]];
      if ( end_row = (col[j+7]<0) )
      e7 = M[j+7] * x[-col[j+7]];
      else
      e7 = M[j+7] * x[ col[j+7]];
      x[i] += e0 + e1 + ... + e7;
      j += 8;
} }
```



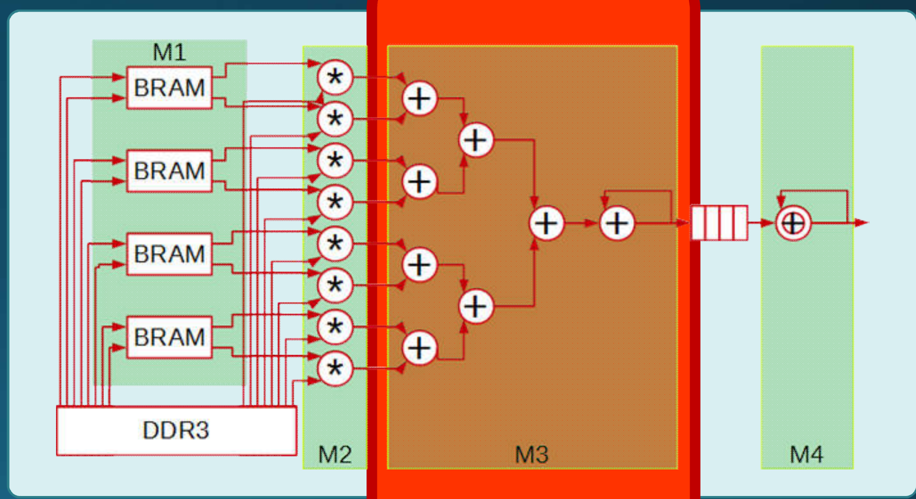
SpMV. Schema of the operator

```
j =0;
for ( i = 0; i < m; i++ ) {
   end_row = 0;
   while ( ! end_row ) {
      e0 = M[j]    * x[col[j]];
      e1 = M[j+1] * x[col[j+1]];
      ...
      e6 = M[j+6] * x[col[j+6]];
      if ( end_row = (col[j+7]<0) )
         e7 = M[j+7] * x[-col[j+7]];
      else
         e7 = M[j+7] * x[ col[j+7]];
      x[i] += e0 + e1 + ... + e7;
      j += 8;
} }
```



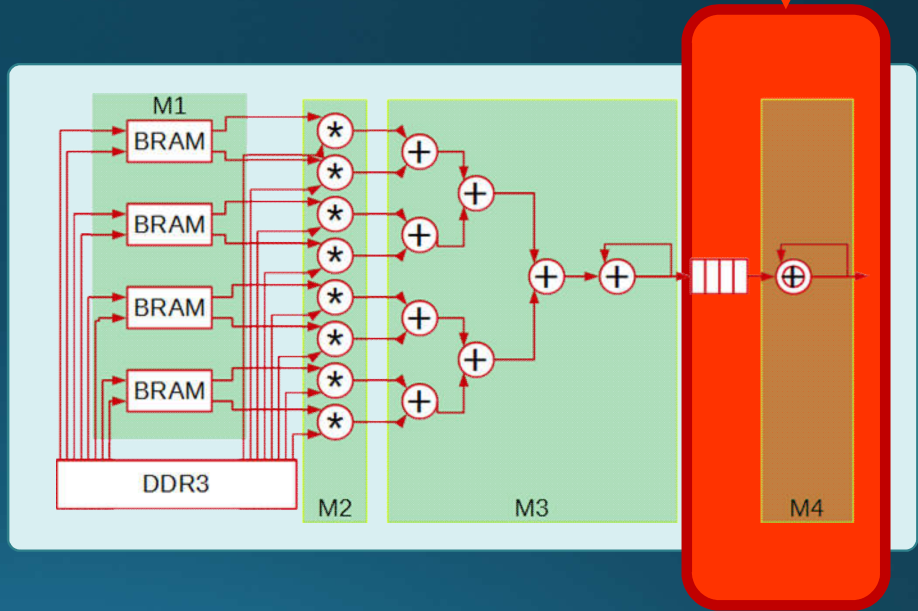
SpMV. Schema of the operator

```
j =0;
for ( i = 0; i < m; i++ ) {
   end_row = 0;
   while ( ! end_row ) {
      e0 = M[j]   * x[col[j]];
      e1 = M[j+1] * x[col[j+1]];
      ...
      e6 = M[j+6] * x[col[j+6]];
      if ( end_row = (col[j+7]<0) )
         e7 = M[j+7] * x[-col[j+7]];
      else
         e7 = M[j+7] * x[ col[j+7]];
      x[i] += e0 + e1 + ... + e7;
      j += 8;
} }
```



SpMV. Schema of the operator

```
j =0;
for ( i = 0; i < m; i++ ) {
   end_row = 0;
   while (    end_row ) {
      e0 = M[j]    * x[col[j]];
      e1 = M[j+1] * x[col[j+1]];
      ...
      e6 = M[j+6] * x[col[j+6]];
      if ( end_row = (col[j+7]<0) )
         e7 = M[j+7] * x[-col[j+7]];
      else
         e7 = M[j+7] * x[ col[j+7]];
      x[i] += e0 + e1 + ... + e7;
      j += 8;
} }
```



SpMV. First results. Latency

Mantissa width (bits)	Multiplier M2	Adder M3	Accumulator M4	Total
13	7	8 (x4 stages)	34	74
21	7	12 (x4 stages)	58	114
53	9	12 (x 4 stages)	58	116

Vivando Design Suite from Xilinx

SpMV. First results. Area and Power

Mantissa width (bits)	Area		Power (W)		
	Slice LUTs	%FPGA	Static	Dynamic	Total
13	13,286	3.07%	0.375	0.300	0.675
21	16,279	3.76%	0.376	0.342	0.718
53	33,519	7.74%	0.379	0.646	1.025

Vivando Design Suite from Xilinx

Work in progress

- Add the selection mechanism for the mantissa width
- Obtain precise real power consumption
- Integrate the connection with the host