SIMD - Programming our SIMD Computer

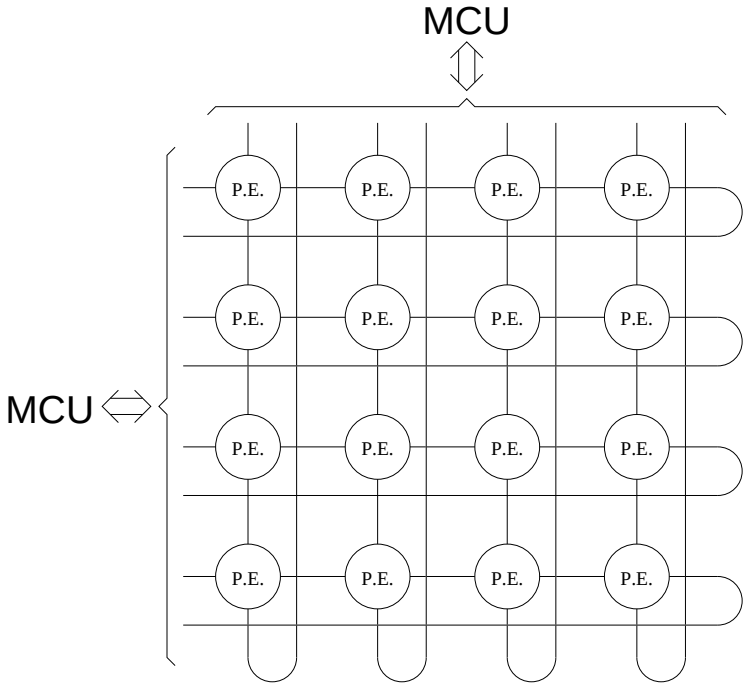
Any operation may be broken down into a *collection of operands*, a *calculation* and a *distribution of results*. Collection and distribution are data movement operations.

Data movement

- Simple memory access for a sequential machine.
- Local or non-local memory access for a distributed memory machine.

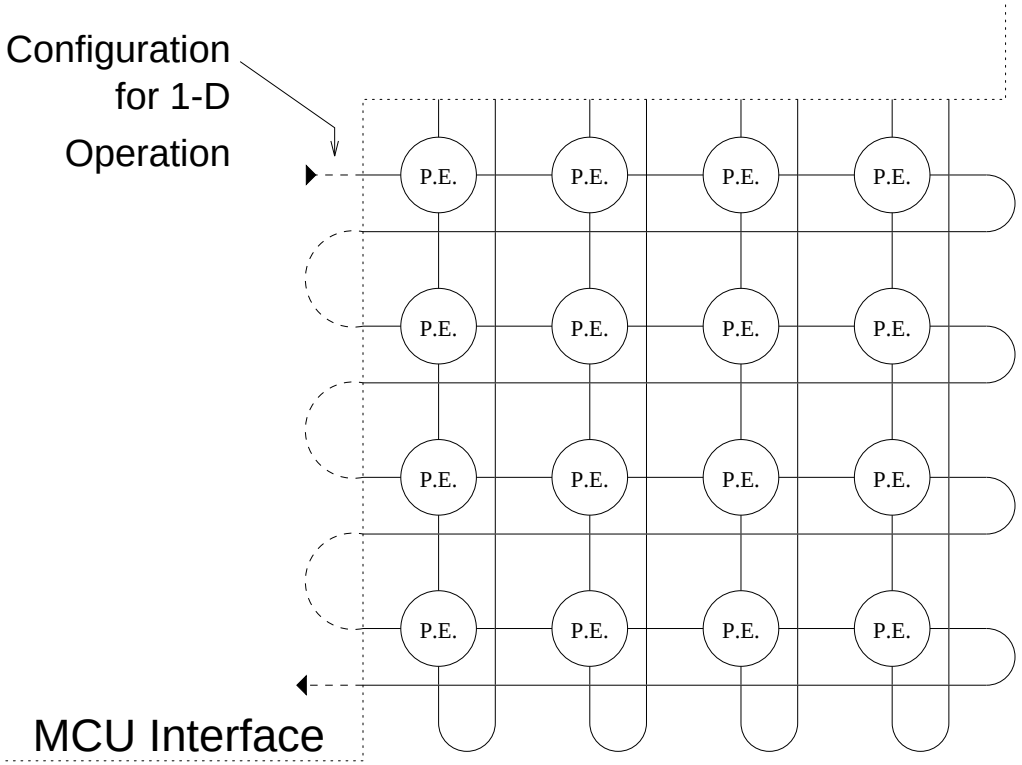
For our SIMD distributed memory machine, a non-local memory access requires data transfer via the inter-PE communication links.

Available Data Transfer Primitives



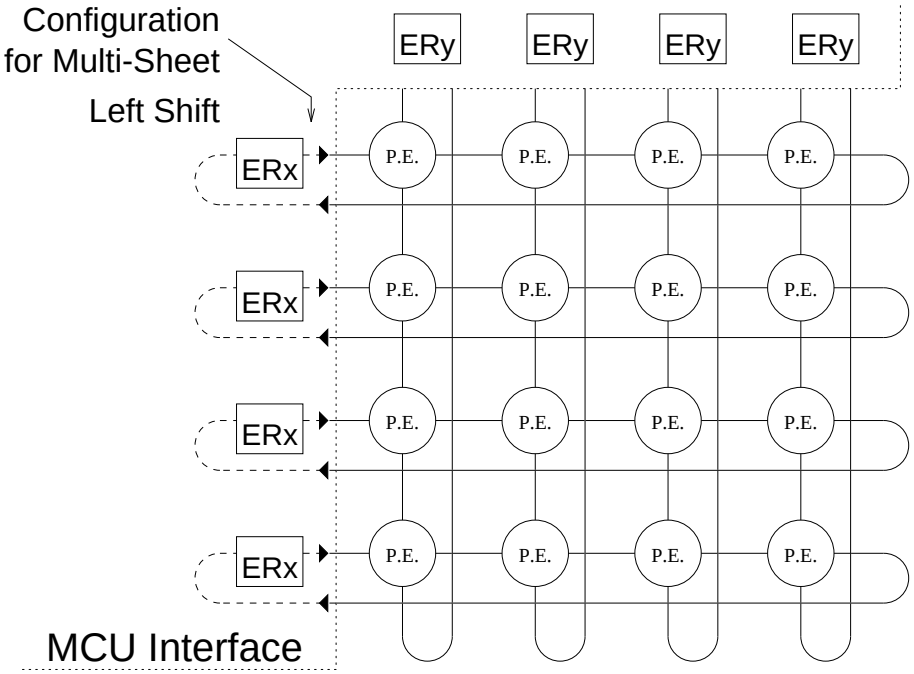
Given the network topology and the constraint that all PEs must perform identical operations, the only data transfer primitive possible is a shift.

1D Configuration

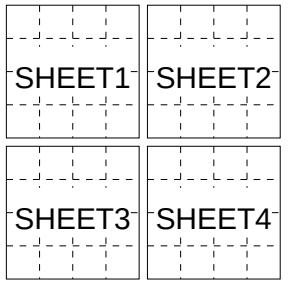


We have already mentioned that the MCU is responsible for re-configuring the network for 1D arrays.

Multi Sheet Arrays



To shift a 4 sheet array:




```
SHIFT_X (SHEET1, +1)
ROTATE_X(SHEET2, +1)
SHIFT_X (SHEET3, +1)
ROTATE_X(SHEET4, +1)
```

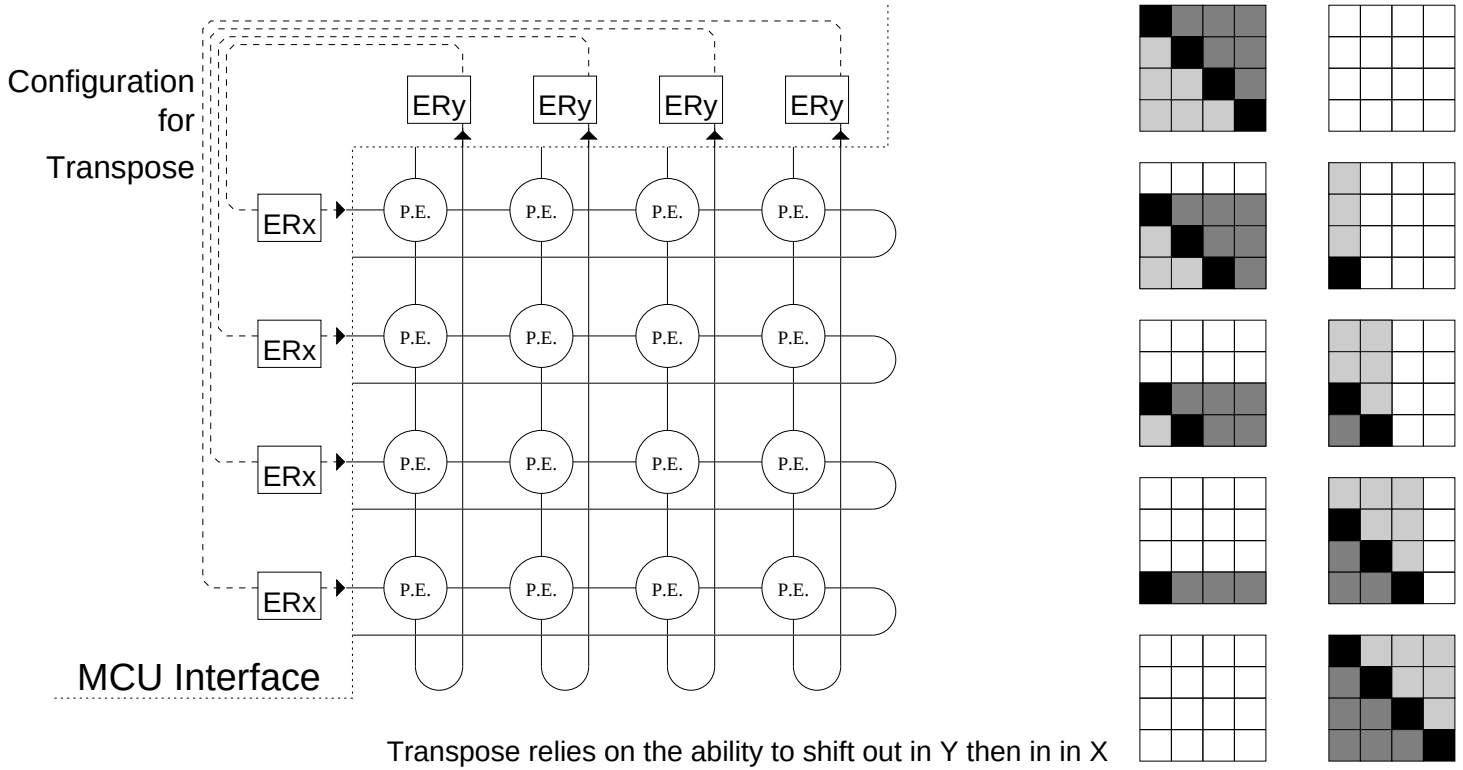
Where

`SHIFT_X`	shifts out to ERx
`SHIFT_X`	shifts in a zero
`ROTATE_X`	shifts out to ERx
`ROTATE_X`	shifts in from ERx

Edge registers within the interface can provide the temporary storage required for multi-sheet shift.

SIMD - Data communications

Transpose

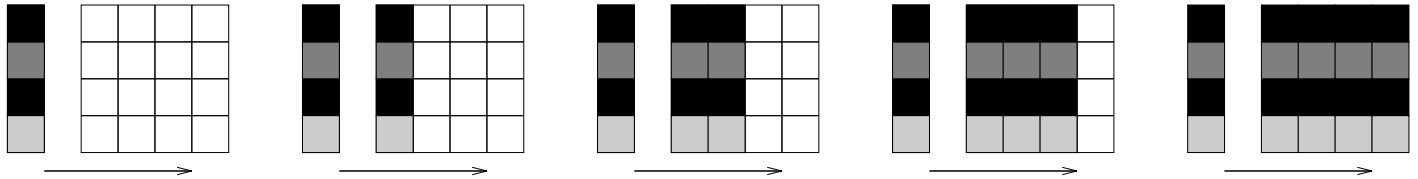


Transpose relies on the ability to shift out in Y then in in X

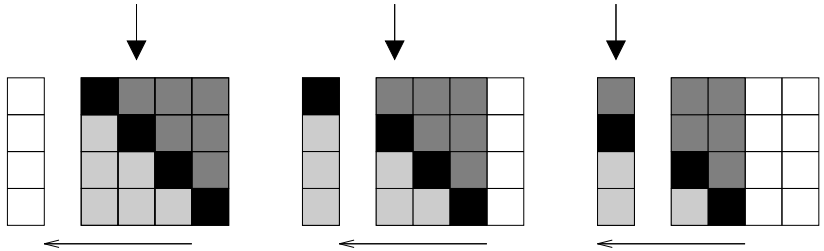
With a little extra connectivity we can perform transpose operations with our edge registers.

Broadcast & Selection

Column Broadcast

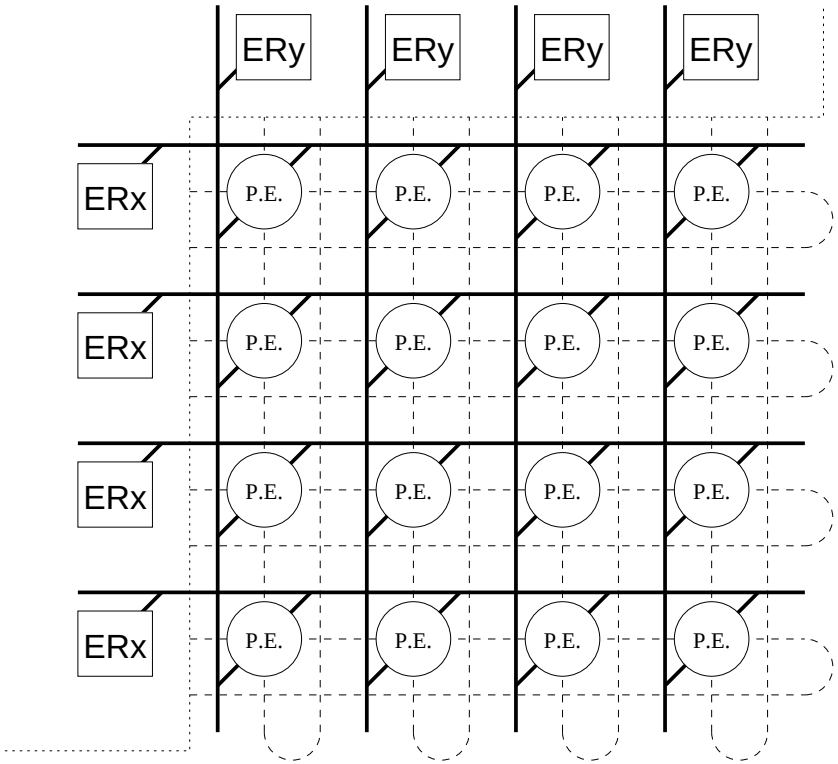


Column Selection



Edge registers can also be used for column/row broadcast, and column/row selection.

SIMD - Data communications



Alternatively we might add column/row *wire or* broadcast lines, allowing single cycle column/row extraction and column/row broadcast.[1]

[1]Combining these connections can give global status information

SIMD - Programming our SIMD Computer

Let us consider the coding of a problem which requires more complex data transfer.

Matrix multiplication

$$\begin{pmatrix} A(1,1) & A(2,1) & A(3,1) & A(4,1) \\ A(1.2) & A(2.2) & A(3.2) & A(4.2) \\ A(1,3) & A(2,3) & A(3,3) & A(4,3) \\ A(1,4) & A(2,4) & A(3,4) & A(4,4) \end{pmatrix} \bullet \begin{pmatrix} B(1,1) & B(2,1) & B(3,1) & B(4,1) \\ B(1.2) & B(2.2) & B(3.2) & B(4.2) \\ B(1,3) & B(2,3) & B(3,3) & B(4,3) \\ B(1,4) & B(2,4) & B(3,4) & B(4,4) \end{pmatrix} = \begin{pmatrix} R(1,1) & R(2,1) & R(3,1) & R(4,1) \\ R(1.2) & R(2.2) & R(3.2) & R(4.2) \\ R(1,3) & R(2,3) & R(3,3) & R(4,3) \\ R(1,4) & R(2,4) & R(3,4) & R(4,4) \end{pmatrix}$$

$$R(i,j) = \sum_{k=1}^{n} A(k.j) \times B(i,k)$$

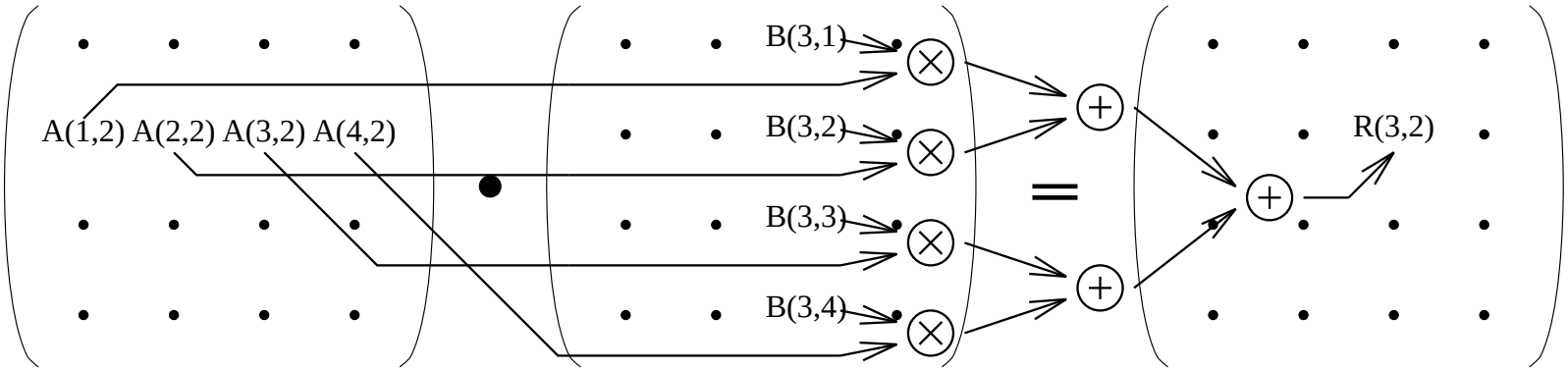
For simplicity we will assume that the two matrices to be multiplied are the same size as the processor array.

---

e.g. *all values zero?*

SIMD - Programming our SIMD Computer

Examine the data flow for a single result:



There is a similar data flow graph for each result.

We must allocate each node on the data flow graph to a single processor.

SIMD - Programming our SIMD Computer

We wish to:

- Minimize Communications
- Maximize Parallelism

Looking at the problem we appear to have quite complex communications. The allocation of nodes to processors is not obvious.

In our favour:

- The result matrix is this the same size as the processor matrix.
- We have an excess of parallelism. (n^3 parallelism, n^2 processors)

SIMD - Programming our SIMD Computer

We can partition the work such that each processor does all the calculations required to produce a single element of the result matrix.

Simply this involves providing a Processor `P(i,j)` with a copy of the two vectors `A( ,j)`[2] and `B(i, )`. Then the calculation of `R(i,j)` is a local calculation.

In order to accomplish the required data movement we must take element `A(i,j)` and smear it across all processors `P( ,j)` using `SMEAR_X`.[3]

Similarly we must take element `B(i,j)` and smear it across all processors `P(i, )` using `SMEAR_Y`.

[2] `A( ,j) = A(1:n,j)`

[3] `SMEAR_X` is accomplished in hardware via column selection and broadcast.

SIMD - Programming our SIMD Computer

We can perform the relevant data movements and calculations on the array in the following manner:

(Pseudo Fortran 90)

```
        R_tot = 0

        DO loop k = 1, n

          A_col_k = SMEAR_X( A, k )
          B_row_k = SMEAR_Y( B, k )
          R_tot   = R_tot + ( A_col_k * B_row_k )
loop
```

Note that all operations act on all array elements at once. The *Dot Product* of the two *n by n* matrices is the elemental sum of *n* elemental multiplications.

Data Transforms

The Matrix multiplication can be divided into two distinct tasks:

- Data Transform
- Calculation

We have shown how regular data transforms can be built from simple shift operations which make use of the edge registers.
We have also shown that certain operations may be accelerated using column and row broadcast lines.

Other Transforms require the use of *activity control* in order selectively move data items. With *activity control* it is possible to produce an arbitrary data transform. Although this may take a very long time.

Hardware Data Transforms

Data Transforms are needed in most real problems.

Depending on the problem we might spend more time re-arranging data than processing it.

- *Hardware Data Transforms.*

  We can implement specific transforms with extra hardware in order to reduce transform time.

 We must balance our communications capability and our processing capability.

How do we know if we have done it right?

It is often easier to say that it is wrong!

Conclusions

Process

- Decide Algorithm
- Map Problem onto array
  - Minimize Communications
  - Maximize Parallelism

We can't ignore

- Our machine is SIMD
- Processor array size
- Processor connectivity