Maximizing the SIMD Behavior in SPMD Engines

Călin Bîra, Liviu Gugu, Mihaela Maliţa, Gheorghe M. Ştefan

***Abstract*—Almost all one-chip parallel architectures are unable to reach a maximum degree of parallelism in performing real applications, mainly due to their weak or too costly support for interconnections between the computational cells. Minimizing or "hiding" the inter-cell communication, totally or partially, is the way we use to improve the degree of parallelism. Few typical algorithms – AES encryption, FFT, Batcher's mergesort – are adapted for this purpose. Our approach is supported by a fine grain cellular structure – implemented as the *Connex* system – featured with enough big local data memory. A high level architectural description for the SPMD system Connex, and a simulator in SCHEME are used to write and evaluate our approach.**

***Index Terms*—parallel computing, parallel algorithms, AES, FFT, Batcher's sort.**

I. Introduction

FLYNN'S taxonomy makes the life difficult for now a days parallel machine designers. A really efficient architecture always falls in the interval between categories. It is the case for one of the most used parallel style of computation: SPMD (Single-Program-Multiple-Data). Some people consider it as a generalized SIMD, other people claim it is a subcategory of MIMD. In this paper, we consider the cellular system *Connex*, designed as a generalized SIMD able to perform predicated SPMD executions. It is implemented as a linear array, of small and simple cells[1].

For the purpose of our approach we defined the functional high level architectural description *Connex High Level Architecture* (CHLA), described in SCHEME. CHLA allows different low level organizations, each having its own ISA. CHLA is also justified by a specific feature of the *Connex* approach: between the register level and the main memory level a large vector buffer is considered. Thus, the *Connex* architecture is an actual update of SWAR (SIMD Within A Register) architecture [4]. Let us call it ***SWAM*** – *SIMD Within A Memory*.

A many-cell organization is the premise for accelerating data intense computations, but such an organization does not guarantee the performance without a special care for how data is exchanged between cells in the array or between the array and the external memory. For example, a many-cell engine is able to solve very efficiently the problem of performing a big number of arithmetic operations requested by the FFT algorithm, but how the intermediary results are exchanged between cells, according to the "butterfly connections", is a non-trivial problem.

More generally, when SPMD is supported by an improved SIMD organization, efficiency is obtained only maximizing the pure SIMD execution. The goal of this paper is to provide simple tricks for hiding the costly inter-cell data exchange operations or inefficient predicated executions.

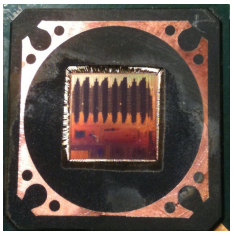

Fig. 1. BA1024 chip

The CHLA is developed and evaluated based on a previous actual implementations of the one chip parallel engine – the 1024-cell *Connex* system BA1024 (see Figure 1) – designed for HDTV applications [10] [8] [9] [6].

II. *Connex* High Level Architecture

The high level architectural view of the *Connex* p-cell system is expressed and simulated using a functional language (we selected a restricted version of SCHEME). The user's image of the system consists of two arrays defining two distinct memory domains:

- S domain: an array of scalars $S =< s_0, s_1, \ldots, s_{n-1} >$, the **external memory**
- V domain: an array of vectors as the **internal vector buffer** $V =< v_0, v_1, \ldots, v_{m-1}, ix >$, a two-dimension array containing m p-scalar ***horizontal vectors*** distributed along the linear array of cells:

$$v_0 =< x_{00}, \ldots, x_{0\,p-1} >$$
$$v_1 =< x_{10}, \ldots, x_{1\,p-1} >$$
$$\ldots$$
$$v_{m-1} =< x_{m-1\,0}, \ldots, x_{m-1\,p-1} >$$

  and the *index* vector $ix =< 0, 1, ..., p-1 >$.

Each "column" in the array V, $w_i =< x_{0i}, \ldots, x_{m-1\,i} >$, is a ***vertical vector*** of scalars associated to cell i.

CHLA hides to the user the vector register level and the scalar register level operations. Three independent sub-architectures are defined in CHLA:

- *data processing* sub-architecture: functions defined on V and S returning values in V or S
- *data transfer* sub-architecture: functions used to transfer data between V and S
- *inter-cell communication* sub-architecture: functions used to perform specific data transfers in V.

Manuscript received July 23, 2013.

Călin Bîra is with the *Politehnica University of Bucharest*, Bucharest, Romania, e-amil: bcalin1984@yahoo.com.

Liviu Gugu is with *On Semiconductor*, Bucharest, Romania, e-amil: liviu_02051987@yahoo.com.

Mihaela Maliţa is with the *Saint Anselm College*, Manchester, NH, USA, e-mail: mmalita@anselm.edu.

Gheorghe M. Ştefan is with the *Politehnica University of Bucharest*, Bucharest, Romania, e-amil: gstefan@arh.pub.ro.

[1]A version, called **Connex*Array*^{TM}**, designed and implemented in silicon for HDTV applications, is described in [8], [9].

A. Data processing sub-architecture

Data processing sub-architecture defines functions performed on vectors, scalars or specific scalar-vector combinations. The main functions are:

- `(ResetActive)`: activates all the components of the vectors involved in the operation that will be executed
- `(Where x)`: keep active, from the active vector components, only the vector components where the Boolean vector `x` returns 1
- `(ElseWhere)`: keep active only the vector components where the Boolean vector `x` returned 0 in the previously executed `Where` function
- `(EndWhere)`: restore the configuration of active components before the execution of the previous `Where`
- `(Test x y)`: where `Test` $\in$ {`Lt, Gt, ...`}, and `x` and `y` are combinations of scalar and/or vector operands; returns a Boolean vector
- `(SetVector x y)`: where `x` is the vector's address in V, and `y` is the vector's content
- `(SetStream x y)`: where `x` is the start address in S and `y` is the stream's content
- `(UnaryOp x)`: where `UnaryOp` $\in$ {`Inc, Dec, Abs, ...`}, and `x` is the scalar or vector operand
- `(BinaryOp x y)`: where `BinaryOp` $\in$ {`Add, Sub, Mult, ...`}, and `x, y` are any combination of scalar and/or vector operands
- `(RedOp x)`: where `RedOp` $\in$ {`RedAdd, RedMax, RedOr, ...`} are the reduction operations, `x` is the vector's content, while `RedOp` returns a scalar
- `(Search x y)`: search the scalar `x` in the active cells of the vector `y`; only the cells where the vector `y` has the value `x` remain active
- `(SearchCond x y)`: condition search the scalar `x` in the vector `y`; the search operation is performed only in cells preceded by an active cell; only the cells where the match is accomplished remain activated
- `(Insert x y)`: `x` is inserted in the active part of vector `y` in the position of the first active position
- `(Delete x)`: the first scalar from the active part of the vector `x` is deleted.
- ...

B. Data transfer sub-architecture

The data transfer sub-architecture defines the transfer functions between the two memory domains, S and V. The main functions are:

- `(StoreVector many vAddr sAddr)`: stores `many` vectors from V, starting at the address `vAddr`, in S starting with the address `sAddr`
- `(LoadVector many vAddr sAddr)`: loads `many` vectors in V, starting at the address `vAddr`, from S starting with the address `sAddr`
- `(StoreVectorStrided many vAddr sAddr burst stride)`: performs `StoreVector` in bursts of size `burst` strided in S with stride `stride`
- `(LoadVectorStrided many vAddr sAddr burst stride)`: performs `LoadVector` in bursts of size `burst` strided in S with stride `stride`
- ...

The previous functions are performed sharing with the other two sub-architectures the internal vector memory V. These functions *compete very little* with the main user of the shared resource – the processing sub-architecture. Indeed, the function `(LoadVector vAddr sAddr)`, for example, reads in one clock cycle an entire vector from V and requires $t_{trans} \in O(p)$ cycles for the whole transfer.

C. Inter-cell communication sub-architecture

The inter-cell communication sub-architecture supports data exchanges between *vertical vectors* in V. The main functions are:

- `(ShiftOp x y)`: with `ShiftOp` $\in$ {`ShiftLeft, ShiftRight, ...`}, `x` is the vector address in V, and `y` is the size of the shift operation; the operation is executed in time belonging to $O(y)$
- `(Transpose x y)`: transpose the square matrices of $y \times y$ elements from V, stored in V starting at the address `x`, in `y` vectors
- `(Permute x y)`: permutes the vector `x` according to the vector of indexes `y`
- ...

These functions use more intense the shared resource V. A shift operation, for example, accesses two times the vector memory V. For small shifts the share of the vector memory use could be important. In order to be able to optimize the execution, the access to the vector memory V could be dynamically prioritized.

D. Parallel execution in the three sub-architectures

The previously defined sub-architectures request a special function which allow the parallel execution of up to three threads of program: *data processing*, *data transfer*, and *inter-cell communication* thread. The function `ParallelEval` has two or three arguments, as follows:

```
(ParallelEval Prog1 Prog2)
(ParallelEval Prog1 Prog2 Prog3)
```

where, for example, `Prog1` is a data processing program and `Prog2` is a data transfer program, or an inter-cell communication program.

The two or the three program threads interact by flags. The following functions are defined to use the flags:

```
(define (Set f)(define f true))
(define (Reset f)(define f false))
(define (Wait cond)(do()(cond)))
```

III. Algorithms with Maximum Degree of Parallelism

The working hypothesis is that an efficiently implemented and transparently executed matrix transpose operation provides the main tool to maximize the degree of parallel execution in our architecture. The appropriate use of the two dimensions of the vector memory V allows us to avoid the main limitation of our architecture: the simplicity of the interconnections between cells. In-vector operations are inefficient when applied on *horizontal vectors*, while the same operations applied on *vertical vectors* are executed very efficient. Indeed, adding x_{ij} with x_{ik} is a very costly operation if $|j - k|$ is big, because the communication between cell j and cell k is performed by shift operations in

ISBN: 978-988-19252-3-7
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

time $O(|j-k|)$, while adding x_{ji} with x_{ki} is performed in constant time because both operands are stored in the local memory of the same cell. The solution: a horizontal vector or sub-vector is transformed in a vertical vector or sub-vector by the transpose operation.

In this section three typical algorithms are investigated. The first one, AES encryption, uses the translation from horizontal vectors to vertical ones, performed through a transpose operation, only at the beginning and at the end of the process. The second function, FFT, calls the transpose in the middle of the process and at the end, while in the last example, Batcher's sort, the transpose operation is performed tightly interleaved with the sorting operations.

The algorithms designed for the previous three functions are evaluated on programs tested on *Connex Simulator* [7] written in Dr.Racket, a version of the SCHEME language.

A. AES

The AES algorithm is performed on fix sized 128-bit blocks organized in 16 8-bit words. Each block is stored as a stream of 4 32-bit scalars in the external memory S. The algorithm on a p-cell engine is executed in three steps:

- **load & transpose blocks** which means:
  - p blocks of data to be encrypted are loaded from S memory domain into V memory domain as four successive vectors, each vector containing $p/4$ successive blocks
  - on the loaded data perform `(Transpose vAddr 4)`, so as the p horizontal sub-vectors, $p/4$ in each vector, become p vertical sub-vectors, one per cell

  and is performed by the function:

```
(define (LoadTrans vAddr sAddr)
    (LoadVector 4 vAddr sAddr)
    (Transpose vAddr 4)
)
```

- **encrypt blocks**, working in a pure SIMD mode, performs the AES encryption of p blocks with a maximum degree of parallelism, because no inter-cell communication is required; the function defined as `(AES sAddr)`, which computes sequentially AES on one block from S starting at `sAddr`, will be performed in parallel as

  `(AES vAddr)`

  on p blocks in V starting from `vAddr`
- **transpose & store blocks** which means:
  - re-organize the result as p horizontal sub-vectors
  - store back in the external memory the result as a stream of 4-element sub-vectors

  performed by the function:

```
(define (TransStore vAddr sAddr)
   (Transpose vAddr 4)
   (StoreVector 4 vAddr sAddr)
)
```

The computation performed in the second step – **encrypt blocks** – is accelerated p times.

But, the first step – **load blocks** – and the last step – **store blocks** – add, because of the transpose operations, a certain overload. The overload is avoided making it transparent because *load*, *store* and *transpose* operations do not belong to the processing sub-architecture, thus they can be performed in a sort of multi-threaded execution associated to two distinct buffers organized in V. Let us define the vector buffer `vb1` starting at the address `vAddr1` in V, and the vector buffer `vb2` starting at the address `vAddr2` in V. We define also the addresses `sAddr1, ... sAddr4`, in S, where super-blocks of p successive blocks start. Two parallel executed routines – `TRANS_AES` and `ENC_AES` – can be now defined using two *flags* for synchronizing them. The two-thread program performing the AES encryption for $4 \times p$ 128-bit blocks is the following:

```
(define (AES)
  (Reset f1)(Reset f2)
  (ParallelEval(TRAN_AES)(ENC_AES))
)
```

The two threads executed in parallel are defined as follows:

```
(define (TRAN_AES)              |(define (ENC_AES)
 (LoadTrans vAddr1 sAddr1)      | (Wait f1)
 (Set f1)                       | (AES vAddr1)
 (LoadTrans vAddr2 sAddr2)      | (Reset f1)
 (Set f2)(Wait(not f1))         | (Wait f2)
 (TransStore vAddr1 sAddr1)     | (AES vAddr2)
 (LoadTrans vAddr1 sAddr3)      | (Reset f2)
 (Set f1)(Wait(not f2))         | (Wait f1)
 (TransStore vAddr2 sAddr2)     | (AES vAddr1)
 (LoadTrans vAddr2 sAddr4)      | (Reset f1)
 (Set f2)(Wait(not f1))         | (Wait f2)
 (TransStore vAddr1 sAddr3)     | (AES vAddr2)
 (Wait(not f2))                 | (Reset f2)
 (TransStore vAddr2 sAddr4)     |)
)
```

Because the sequence of operations

```
(LoadTrans vAddr sAddr)(Set f)
(Wait(not f))(TransStore vAddr sAddr)
```

is executed faster than the sequence of operations

```
(Wait f)(AES vAddr)(Reset f)
```

the `transfer & transpose` operations are transparent, and the execution time is provided only by the execution of the `(AES vAddr)` operations.

The case of the AES algorithm is a simple one, because the process of rearranging data in the vector memory V is done in constant and short time. (In the next two examples more complex situations must be solved.) We conclude that, for this simple case, the parallel program works on vectors like a sequential program on scalars accessed from a local buffer.

Time, area and power performances for the AES encryption: In order to compare CHLA architecture with a standard sequential architecture we will consider a *Connex* system implementation, with 64 16-bit cells each having 0.5 KB local static memory – *Connex64* – which has the area and the power consumption of a Cortex A9 ARM core when both are synthesized to run in the same technology and at the same frequency.

The direct implementation of the AES algorithm on *Cortex A9* provided 173 cycle/byte performance. Because, TRAN_AES function works in less than $1/4$ cycles than the function ENC_AES, the overall execution time is the time for ENC_AES function. It is measured to 2.1 cycle/byte on *Connex64*. Results the use of area & power is $173/2.1 =$ **82 times higher** on *Connex64* than on *Cortex A9*. The improvement is more than 64x on a 64-cell engine, because on the Connex Architecture the transfer and transpose operations *can* be performed transparently, and, fortunately, for the AES algorithm these operations *are* performed transparently, because the size of each block is small and constant.

ISBN: 978-988-19252-3-7
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

B. FFT

Any parallel implementation for the FFT algorithm distributes very efficiently the arithmetic operations over the computational cells, but is confronted with the "butterfly interconnections" between the cells. The difficulties are maximal because of the inter-cell communication time, which is determined by the way the shift operations are executed.

Let us take, for example, a 16-cell engine and a 16-sample FFT. The 16 samples are distributed, one in each cell as the horizontal vector $V_1 =< x_0, x_1, \ldots x_{15} >$. In the first stage, each cell performs an addition/subtract between the local component of the vector, x_i and the component received from the cell $i+8$ or $i-8$. Then, in the last 8 cells the result is multiplied with the twiddle number $w_i = real(w_i) + j \times im(w_i)$. The second stage is similar, only the spatial distance in the linear array of cells is smaller. In the third stage the data must move only 2 cells, while in the last stage, the numbers to be added/subtracted are located in adjacent cells. It is obvious that, for a big number of samples, n, the time is dominated by the data move in the array. Indeed, the time involved in performing the arithmetic operations is in $O(log\ n)$ (with 75% degree of parallelism), while the time for inter-cell data movement is in $O(n)$. From $O(n \log n)$, for the sequential execution, we are now at $O(n)$, obtaining an $O(\log n)$ acceleration with p cells. Maybe we can do more.

We will continue to minimize the execution time and the degree of parallelism in two steps. First, the arithmetic computation is segregated from the data move, building a "compact" algorithmic section for data move (see also [5]), in order to obtain 100% degree of parallelism, then computation and data move will be performed transparent to each other.

1) Segregating computation by data move: The limitation to be removed is due to the fact that we process horizontal vectors or sub-vectors. If the horizontal vectors or sub-vectors can be translated into vertical sub-vectors, then the problem should have a solution similar to the AES encryption. Unlike AES encryption, where the blocks are fix sized, the number of samples in FFT computation can have any size. Thus, for a small n an AES-like solution works, providing a p times acceleration. But, unfortunately, a solution for big n must be provided also.

If one vertical sub-vector is not big enough to include all the n samples, more than one sub-vectors are used, organizing the samples in a two-dimension array of r rows and c columns. In the vector memory V in r vectors are loaded the samples for p/c FFTs, having for each FFT c vertical sub-vectors stored in c successive cells. The resulting algorithm has three stages:

- compute in parallel p r-sample FFTs, in the time for one r-sample FFT, with 100% degree of parallelism
- make $2 \times (r/c) \times p/c$ `(Transpose vAddr c)` operations, in the time for $2 \times (r/c)$ `(Transpose vAddr c)` operations
- compute $r/c \times p$ c-sample FFTs, in the time for r/c c-sample FFTs, with 100% degree of parallelism.

Going back to our example of 16-sample FFT, instead of performing 1 FFT by distributing one vector of samples along the 16 cells of the parallel engine, we will compute, for example, 4 16-sample FFTs organizing the input data in $r \times c = 4 \times 4$ arrays. The 4 sequences of samples are:

$$< a_0, a_1, \ldots a_{15} >$$
$$< b_0, b_1, \ldots b_{15} >$$
$$< c_0, c_1, \ldots c_{15} >$$
$$< d_0, d_1, \ldots d_{15} >$$

each loaded in the 16-cell engine, *Connex16*, as two-dimension 4×4 arrays in the following 4 vectors:
$V_0 =< a_0,\ a_1,\ a_2,\ a_3,\ b_0,\ b_1,\ \ldots d_0,\ d_1\ d_2,\ d_3 >$
$V_1 =< a_4,\ a_5,\ a_6,\ a_7,\ b_4,\ b_5,\ \ldots d_4,\ d_5\ d_6,\ d_7 >$
$V_2 =< a_8,\ a_9,\ a_{10}, a_{11}, b_8,\ b_9,\ \ldots d_8,\ d_9\ d_{10}, d_{11} >$
$V_3 =< a_{12}, a_{13}, a_{14}, a_{15}, b_{12}, b_{13}, \ldots d_{12}, d_{13} d_{14}, d_{15} >$
Let be:

```
x0  x1  x2  x3
x4  x5  x6  x7
x8  x9  x10 x11
x12 x13 x14 x15
```

the generic array used to describe the algorithm, where, the symbol `x` stands for a, b, c or d. Any operation applied to `x5`, for example, applies simultaneously to a_5, b_5, c_5 and d_5.

The previous 3-stage algorithm for p/c $(r \times c)$-sample FFTs, translates in three functions, as follows:

- `(FFT1)`: compute 16 4-sample FFTs on the vertical real vectors $< a_0, a_4, a_8, a_{12} >$, ..., $< d_3, d_7, d_{11}, d_{15} >$ stored in `v1, v2, v3, v4`; this stage provides the result in `v0, v1, v2, v3`, for the real part, and `v4, v5, v6, v7`, for the imaginary part
- `(TRANSPFFT)`: the 4 4×4 arrays stored in `v0, v1, v2, v3` (containing the real components) and the 4 4×4 arrays stored in `v4, v5, v6, v7` (containing the imaginary components) are transposed
- `(FFT2)`: continue by the 16 4-sample FFTs on the vertical complex vectors stored in `v0, ... v7`.

Thus, the program for computing 16-sample FFT is (see for details [7]):

```
(define(FFT16)
    (FFT1)
    (TRANSPFFT)
    (FFT2)
)
```

The functions `FFT1` and `FFT2` are executed with maximum degree of parallelism using the engine with maximum efficiency. On the other hand the function `TRANSPFFT` is defined using transpose operations `(Transpose x y)`, which are performed, unfortunately, in $O(y^2)$ time, while the functions `FFT1` and `FFT2` are performed in $O(y \log y)$. Therefore, there is a y_0 so as for $y > y_0$ the computation time of `FFTn`, where $n = y^2$, is dominated by the execution time of the function `TRANSPFFT`.

2) Performing transparently the computation and the data move: Because the *Connex* system is featured with a relatively large memory at the level of each cell, a double buffer approach will help to hide the transpose operations for a reasonable big n, while for bigger $n > n_0$ the computation time will remain to be determined by the transpose time.

The computation is performed on two sets of vectors. Revisiting the previous example, besides the initial set of 8 vectors `VA = {v0, ... v7}`, another set of 8 vectors, `VB = {v16, ... v23}` is considered in order to interleave two sets of FFTs. The algorithm will run as two threads synchronized with two flags: `f1, f2`. The functions working on `VA` are suffixed with `A`, while the functions working on `VB` are suffixed with `B`. The two-thread program is:

```
(define (FFT2T) ; FFT on 2 threads
```

ISBN: 978-988-19252-3-7
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

```
   (Reset f1)(Reset f2)
   (ParallelEval(COMPUTE_FFT)(TRANPOSE_FFT))
)
```

where the two threads are defined as follows:

```
(define(COMPUTE_FFT)     | (define(TRANPOSE_FFT)
  (FFT1A)(Set f1)        |   (Wait f1)(TRANSPFFTA)
  (FFT1B)(Set f2)        |   (Reset f1)
  (Wait (not f1))(FFT2A)|   (Wait f2)(TRANSPFFTB)
  (Wait (not f2))(FFT2B)|   (Reset f2)
)                        | )
```

3) Time, area and power performances for FFT: Let be t_{TRANSP} the execution time for (TRANSPFFTA) or (TRANSPFFTB), and t_{COMP} the execution time for (FFT1A) or (FFT2A) or (FFT1B) or (FFT2B). As long as

$$t_{COMP} \geq t_{TRANSP}$$

the running time for computing p/c $(r \times c)$-sample FFTs is:

$$t_{FFT} = 2 \times t_{COMP}$$

and the **parallel acceleration** is in $O(p)$.

But, there is a $n = n_0$ starting from which $t_{TRANSP} > t_{COMP}$, because $t_{TRANSP} \in O(n)$, while $t_{COMP} \in O(r \times log\, r)$, and the worst case, when n_0 is minimal, is for $r = c = \sqrt{n}$, because $t_{COMP} \in O(\sqrt{n} \times log\, n)$. Then, the running time for computing $p/\sqrt{n}$ n-sample FFTs is:

$$t_{FFT} = t_{COMP} + t_{TRANSP}$$

and the **parallel acceleration** is in $O(p \times \frac{log\, n}{\sqrt{n}})$, because n-sample FFTs are computed in parallel $p/\sqrt{n}$ in time belonging to $O(n)$ instead of $O(n\, log\, n)$. We can keep us away from the worst case by increasing the value of m (the size of the vector space), thus allowing big values for the ratio r/c.

The threshold value for n, n_0, depends also on the actual execution time for the arithmetic functions performed in the cells. For small and simple cells the execution time for integer multiplication is around 10 cycles, for example. But, in this case the value for n_0 is higher and on the same silicon area more cells can be accommodated. The optimal value for an actual design will be established taking into account the parameters imposed by the application domain.

Because for computing FFT 32-bit computation is requested, a 32 32-bit cell engine, called *Connex32*, is used to evaluate the performance. It has the size and the power consumption of a *Cortex A9* core. If $n < n_0$, then a **18.8x *improvement in area and power use is obtained for FFT execution***. The *Connex32* engine does not have hardware multipliers in its cell. Therefore, the multiplication is performed in 10 cycles, resulting less than 32x improvement.

C. The Batcher sort

The Batcher sorter [1] has a twice recursive definition. Loading the sequence to be sorted, $< x_0, x_1, \ldots x_{15} >$, as a horizontal vector, the execution time will be dominated by the shift operations requested to gather, in the appropriate cells, the pairs to be sorted. As in the previous two examples, AES and FFT, the vertical dimension of the CHLA must be smartly involved.

1) Using transpose to rearrange data for pure SIMD computation: If the sequences of numbers to be sorted are short enough, then each cell will be loaded with one sequence and the sequential algorithm is applied in parallel on p sequences. But in real applications we must consider also the less friendly case when the length of the sequence is too big to be loaded in one cell. Therefore, the case of a two-dimension array loaded in more than one cell must be considered. Let us consider that each n length sequence is loaded as a $r \times c$ array, as r-length vertical vectors in c successive cells. Then, the first $(log_2 r)(1 + log_2 r)/2$ stages of the sorting algorithm are performed in a pure SIMD mode on p r-length sequences of numbers. For the next stages appropriate transpose operations must be performed, in order to rearrange each sequence of numbers in vertical sub-vectors allowing pure SIMD operations.

Let us consider again an example with a 16-cell engine processing the following 8 16-scalar sequences:

$$\begin{array}{c} < a_0, a_1, \ldots a_{15} > \\ < b_0, b_1, \ldots b_{15} > \\ \ldots \\ < h_0, h_1, \ldots h_{15} > \end{array}$$

each loaded in the 16-cell engine as a two-dimension 8×2 arrays in the following 8 vectors:

$V_0 = < a_0, a_8,\ b_0,\ b_8,\ \ldots h_0, h_8\ >$
$V_1 = < a_1, a_9,\ b_1,\ b_9,\ \ldots h_1, h_9\ >$
$\ldots$
$V_7 = < a_7, a_{15}, b_7, b_{15}, \ldots h_7, h_{15} >$

Thus, eight sequences are sorted in parallel. Let be

```
x0  x8
x1  x9
x2  x10
x3  x11
x4  x12
x5  x13
x6  x14
x7  x15
```

the generic sequence, organized as a 8×2 array, considered for explaining the algorithm. Such a data structure is loaded in each group of 2 cells starting with $cell_0, cell_1$ and ending with $cell_{14}, cell_{15}$. The sub-sequence $x_0 \ldots x_7$ belongs to the vertical vector w_0, the sub-sequence $x_8 \ldots x_{15}$ belongs to the vertical vector w_1, and so on.

For $n = 16$, the algorithm has 10 stages. The first 6 stages involve variables contained in the same vertical vectors. For the next stages, appropriate transpose operations will be performed. Thus, in each of the following stages the 8×2 arrays are reconfigured by transpose operations, if needed, and the elementary sorting operations, S, are performed in each activated cells. The program is (see for details [7]):

```
(define(SORT)
    (Sort1to6)     ; stages 1 to 6 of sort
    (Trans0to6)    ; 4 2x2 transpose
    (Sort7to9)     ; stages 7 to 9 of sort
    (Trans0to6)    ; back to the initial form
    (Sort10first)  ; first sorts of stage 10
    (Trans0)       ; one 2x2 transpose
    (Sort10last)   ; last sort of stage 10
    (Trans0)       ; back to the initial form
)
```

For this algorithm also, as for the FFT algorithm, a big r is requested for minimizing the execution time. For big c the

ISBN: 978-988-19252-3-7
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

weight of the transpose operation increases. Fortunately, for the same size, m, of the vector memory V, the value for r is bigger than the value used in the FFT algorithm, because for performing the sorting operation no extra space is needed than the space used to load the sorted sequence.

Because for big c the weight of the transpose operation increases, we must provide a solution for hiding as much as possible the effect of the transpose operation.

2) Interleaving `Sort` *with* `Trans`*:* For interleaving `Sort` with `Trans`, the working space in V is divided in two sub-spaces, `VA` and `VB`, in order to accept two sets of streams to be sorted. The functions working on `VA` are suffixed with `A`, while the functions working on `VB` are suffixed with `B`. One thread performs the function `COMPUTE_SORT`, while the other performs the function `TRANSPOSE_SORT`. The two threads are synchronized using the flags `f1` and `f2`. The two functions are defined as follows:

```
(define(COMPUTE_SORT)   |(define(TRANSPOSE_SORT)
 (Wait(not f1))         | (Wait f1)(Trans0to6A)
 (Sort1to6A)(Set f1)    | (Reset f1)
 (Wait(not f2))         | (Wait f2)(Trans0to6B)
 (Sort1to6B)(Set f2)    | (Reset f2)
 (Wait(not f1))         | (Wait f1)(Trans0to6A)
 (Sort7to9A)(Set f1)    | (Reset f1)
 (Wait(not f2))         | (Wait f2)(Trans0to6B)
 (Sort7to9B)(Set f2)    | (Reset f2)
 (Wait(not f1))         | (Wait f1)(Trans0A)
 (Sort10firstA)(Set f1)| (Reset f1)
 (Wait(not f2))         | (Wait f2)(Trans0B)
 (Sort10firstB)(Set f2)| (Reset f2)
 (Wait(not f1))         | (Wait f1)(Trans0A)
 (Sort10lastA)(Set f1)  | (Reset f1)
 (Wait(not f2))         | (Wait f2)(Trans0B)
 (Sort10lastB)(Set f2)  | (Reset f2)
)                       |)
```

and the resulting program is:

```
(define (SORT2T); sort on two threads
 (Reset f1)(Reset f2)
 (ParallelEval(COMPUTE_SORT)(TRANSPOSE_SORT))
```

3) Time, area and power performances for Batcher sort: The previously described algorithm for parallel sort of $64/2$ 16-number sequences provides an acceleration of 84x on our *Connex64* compared with ARM's *Cortex A9*. But, in this case the computing time is provided almost exclusively by the run of the COMPUTE_SORT thread. For bigger n and another ratio r/c, the weight of TRANSPOSE_SORT is expected to increase. *It is hard to predict* how will evolve the play between `Sort` and `Trans` for various $n = r \times c$. But, for the worst case, the `(Trans v i)` operations will dominate the `(Sort v w)` operations. Then, our estimation is that for an hypothetical limit case (when instead of `(Sort v w)` operations, `(Trans v i)` operations are counted) the acceleration is 70x. Therefore, a conservative estimate is a linear acceleration of sorting using the Connex technology.

IV. Concluding Remarks

The three sub-architectures of CHLA – *data processing* sub-architecture, *data transfer* sub-architecture and *inter-cell communication* sub-architecture – provide a programming and execution environment well fitted for hiding the limits introduced by the simplest interconnection network used to organize our cellular engine.

An efficiently implemented transpose operation, which is executed transparently to the operations of the data sub-architecture, allows the pure SIMD operations to dominate the execution time for many real applications.

The investigated algorithms – AES, FFT, Batcher's sort – proved to be accelerated at least linearly in our architecture as long as there is enough space in the V domain.

Big sized local memory in each cell is the first requirement for designing a multi-buffer computation which allows to hide both, the data transfer between V domain and S domain, and the inter-cell communication in V domain.

SPMD is general, but SIMD is efficient. A SIMD organization able to perform predicated execution becomes a fairly efficient SPMD system, but, the degree of parallelism is damaged. In order to maintain the degree of parallelism as high as possible, the weight of pure SIMD operations are increased by the play, between horizontal and vertical vectors, based on a transparent transpose operation.

Playing with horizontal and vertical vectors by using transpose operation, could open a wide algorithmic research domain. Our SWAM architecture stimulates the transpose game in order to maximize the pure SIMD behavior.

Using a functional language for simulating and programming an engine working as accelerator is a good solution. Data intense computations have a pronounced functional aspect which is efficiently expressed by a functional language, in our case the SCHEME programming language. In [6] we already have emphasized the advantages of defining parallel architectures by using Backus's FP forms.

Acknowledgment

The authors got a lot of support from the main technical contributors to the development of the *ConnexArray*^{TM} technology, the *BA1024* chip, the associated language, and its first application: E. Altieri, F. Ho, B. Mîţu, M. Stoian, D. Thiebaut, T. Thomson, D. Tomescu.

References

[1] Kenneth E. Batcher: "Sorting networks and their applications", in *Proc. AFIPS Spring Joint Computer Conference*, vol. 32, 1968.

[2] Eleanor Chu, Alan George: *Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms*. CRC Press, 2000.

[3] James W. Cooley and John W. Tukey. "An algorithm for the machine calculation of complex fourier series". *Mathematics of Computation*, 19(90):297301, 1965.

[4] Randall J. Fisher, All J. Fisher, Henry G. Dietz: "Compiling For SIMD Within A Register" in *11th Annual Workshop on Languages and Compilers for Parallel Computing*, 1998.

[5] Istvan Lorentz, Mihaela Maliţa, Răzvan Andonie: "Fitting FFT onto an Energy Efficient Massively Parallel Architecture", in *The Second International Forum on Next Generation Multicore/ Manycore Technologies*, June, 2010.

[6] Mihaela Maliţa, Gheorghe M. Ştefan: "Parallel RISC Architecture. A Functional Approach Based on Backus's FP language", in *Proceedings of the 2011 International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, 2011, pp 492-498.

[7] Mihaela Maliţa, Liviu Gugu, Călin Bîră: *Connex Simulator*, posted at: `http://www.anselm.edu/internet/compsci/Faculty_Staff/mmalita/HOMEPAGE/research.html`

[8] Gheorghe Ştefan, Anand Sheel, Bogdan Mîţu, Tom Thomson, Dan Tomescu: "The CA1024: A Fully Programmable System-On-Chip for Cost-Effective HDTV Media Processing", in *Hot Chips: A Symposium on High Performance Chips*, Memorial Auditorium, Stanford University, August 20 to 22, 2006.

[9] Gheorghe Ştefan: "One-Chip TeraArchitecture", in *Proceedings of the 8th Applications and Principles of Information Science Conference*, Okinawa, Japan on 11-12 January 2009. Posted at: `http://arh.pub.ro/gstefan/teraArchitecture.pdf`

[10] Gheorghe Ştefan: The *Connex* Project, posted at: `http://arh.pub.ro/gstefan/Connex.html`.