Low-Power Stereo Vision Accelerator for Automotive

Mihaela Maliţa
St. Anselm College, NH, USA
mmalita@anselm.edu

Octavian Nedescu
Assystem Romania
octavian.nedescu@gmail.com

Alexandru Negoiţă
Assystem Romania
alexandru.negoita@yahoo.com

Gheorghe M. Ştefan
Politehnica Univ. of Bucharest
gheorghe.stefan@upb.ro

***Abstract*—Various forms of Convolutional Neural Network (CNN) architectures are used as Machine Learning (ML) tools for learning the similarity measure on video patches in order to run the *stereo matching algorithm* – the most computationally intensive stage of the pipeline for the stereo vision function used in designing an autonomous car. We propose a hybrid system for real-time, low-power and high-temperature implementations of the algorithm. The accelerator part of the system is a programmable many-core system with a Map-Reduce Architecture. Our paper describes and evaluates the proposed accelerator for running versions of the stereo matching algorithm.**

I. Introduction

The new form in which the Artificial Intelligence (AI) revives in the last few years is ML implemented with CNN technology which includes three kinds of neural networks (NN): fully connected layers of NN (FNN), convolution layers (CL), recurrent NN (RNN). The CNN domain stresses the computational resources of the current computers in two stages: the *training* stage (the network weight matrix is established) and the *inference* stage (the network works in a real application). In the first stage we are challenged by the training time, while in the second stage the main challenge is the power consumption in real time applications (for example automotive video processing or data center data mining, where both energy and time are critical).

The computational resources involved in training or inference are CPU (ex.: Intel's i7), MIC (ex.: Intel's Xeon Phi), GPU (ex: Nvidia), FPGA or ASIC with domain-specific architecture (ex.: Google's TPU). Each of these solutions have their good and bad aspects. Our proposal is a programmable accelerator, with a Map-Reduce architecture, for a hybrid system designed as an Accelerated Processing Unit (APU).

This paper describes an APU for the *stereo matching algorithm*, able to provide the computational power for real time automotive video processing, in the limits imposed by the actual application for energy consumption. We consider the algorithms for stereo matching developed by Žbontar and LeCun [13] [14].

The second section refers to the state of art in implementing real application with CNN. The third section emphasizes the set of the main functions requested for running efficiently a CNN. The organization and the architecture of our proposal is described in the fourth section. In the fifth section the implementation on our system of the CNN involved in stereo matching is shown. The evaluation of speed and energy at high temperature is done in the sixth section.

II. State of the Art

The easy solution for hybrid computation is to take from the shelf many-core accelerators. Unfortunately, the available many-core accelerators (ex.: Nvidia's GPU or Intel's Xeon Phi) do not fit for a specific application such as CNN. Their huge computational power is too much under used. While from Intel's i7 CPU, with 112 GFLOPs/sec, 32% is used for real time object detection, with Titan X GPU, for 40-90 fps, are used maximum 63 GFLOPs/sec from its peak performance of 6 TFLOPs/sec [10], or with Xeon Phi accelerator with 57 cores, having peak performance at 2 TFLOPs/sec, only 0.48 GFLOPs/sec is used from each core that is able to provide 35.2 GFLOPs/sec [9]. It is hard to explain why from 32% use of the peak performance for CPU we go to around to 1% use for the many-core accelerators? We suppose it concerns an architectural and organizational inadequacy.

The last, more efficient solution for a CNN implementation is the Google's Tensor Processing Unit (TPU). While GPU/CPU relative performance per Watt is only 2.9×, TPU/CPU is 83× and TPU/GPU is 83× [2]. The explanation for these very good improvements are:

- TPU is an ASIC with a domain-specific architecture; a more appropriate name should be Tensor Execution Unit, because it executes the stream of instructions provided by the host processor through a PCIe interface, with no loop-control back to the host
- the core of the chip is a systolic array [5] of 256×256 8-bit multipliers
- while for training the CNN floating-point arithmetic is used, for inference the floating-point weights are very frequently quantized to 8-bit signed integers

The peak performance for TPU is 92 TOPS/sec of 8-bit operations. The chip has $\geq 662 mm^2$ in $28 nm$ and works at $700 MHz$ with $TDP = 75 Watt$.

The good performance of TPU must be combined with the flexibility of GPU or MIC solutions. Our proposal, the Map-Reduce accelerator, provides performance in between the current many-cores and the TPU ASIC solution.

III. Functional Requirements

All types of NN involved in ML applications – FNN, CL and RNN – are based on the same computational pattern: matrix-vector multiplication. The only difference between them is the flow of data. For FNN the model of matrix-vector multiplication is direct: the weight matrix is multiplied with the input vector and provides a vector whose components are submitted to the activation function. For RNN, only the loop from output to the input of the FNN is added by concatenating the input vector with (a part of) the output vector. For CL computation, a little data moving work must be added. It depends on the number of cells of the accelerator, the size of the local memory in cells and the parameters of CL. Following [3], we provide some useful details to be used in understanding how our solution works.

In contrast to the standard neural layer, characterized by a two-dimension weight matrix, a convolutional layer has a more complex structure. The structure of a convolutional layer can be summarized by the following parameters:

- the input of a convolutional layer accepts a "volume" of size $V_i = W_i \times H_i \times D_i$ values
- the definition of the transformation produced by CL requires four parameters:
  - the number of filters K
  - the spatial extent of the receptive field of volume, $V_r = F \times F \times D_i$, with $F << W_i$ and $F << H_i$, used to explore the input volume
  - the stride, S, with $S \leq F$, used to explore the input plane $W_i \times H_i$
  - the amount of zero padding P, used to expand the input volume
- the output "volume" of size $V_o = W_o \times H_o \times D_o$, where:
  - $W_o = (W_i + 2P - F)/S + 1$
  - $H_o = (H_i + 2P - F)/S + 1$
  - $D_o = K$
- the same parameters shared over all the receptive fields, introduces $F \times F \times D_i$ weights per filter; results the total of $(F \times F \times D_i) \times K$ weights and K biases.

An important characteristic of a convolutional layer is: the number of parameters requested is small compared with the input and output data, because

$$(F \times F \times D_i) \times K << W_i \times H_i \times D_i$$

The operation performed in a convolutional layer is applied to each receptive field (see Figure 1a) from the input volume. The receptive field is represented as a vector X of $r = F \times F \times D_i$ elements. The inner product (IP) between the vector of weights W of r components and X is submitted to the non-linear activation function f. The function f and the vector W is the same for all the receptive fields defined for the input of a convolutional layer. Thus, for a computational pipe associated to a CNN, the vector W loaded only once in the accelerator and it is used many times, in contrast with the weight matrix $\mathbf{W}$ which defines a fully connected NN layer, and which is loaded and used only once. Each receptive field is used to generate K elements in the output volume.

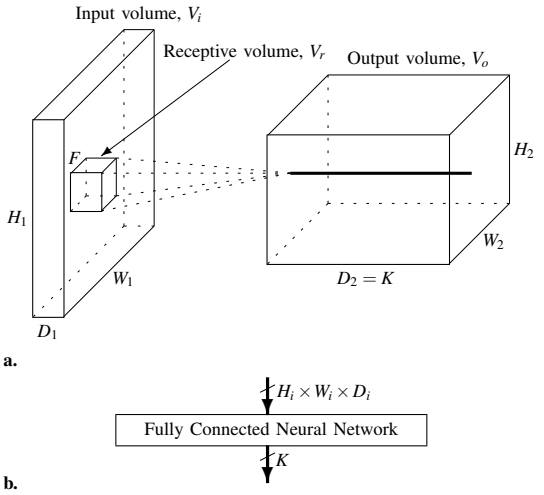

Fig. 1. **a.** Convolutional layer. Each receptive field from the input layer generates K elements of the output volume. **b.** Fully convolutional layer. For $F = H_i = W_i$ and $P = 0$ the convolutional layer becomes a fully connected layer.

If K filters are applied, then a matrix of $(F \times F \times D_i) \times K$ weights are used to compute K elements in the output volume. Then, the main operation is also the matrix-vector product, and the FNN layer of K neurons can be seen as a particular case of a receptive field with $F = D_i = H_i$ and $P = 0$ (see Figure 1b).

Let us conclude that the computation pattern in a DNN is the matrix-vector multiplication applied in different ways to arrange the data inside the accelerator. The different ways to load data in the accelerator depend on the size of the matrices involved and also on how many times a weight is used once loaded in the accelerator. Because there are many data configurations requested we need a very flexible mechanism for loading and rearranging data inside the accelerator.

IV. Map-Reduce Accelerator

The computation requested by CNN is dominated by the inner product (IP), the elementary operation in matrix-vector multiplication. The IP operation requests a two-level structure: the *map* level of multiplication and the *reduce* level of summation. Our proposal is based on this observation and on a previously implemented cellular engine [11] [7]. In [12] the Map-Reduce approach is proved to be based on a mathematical model of computation [4]. The cellular organization of the proposed Map-Reduce Accelerator (MRA) and its instruction set architecture (ISA) are described in the following.

A. Organization

The hybrid computing system we propose for efficient CNN training and inference has the following subsystems:

- HOST: a general purpose CPU for the complex part of the code and for accessing the set of functions accelerated by the associated co-machine

- MEMORY: the system memory for programs and data
- Interconnection Fabric: a data transfer unit able to transfer scalar or vectors of various size representing the programs executed by the co-machine and the data exchanged with MEMORY
- ***Map-Reduce Accelerator***: the co-machine to run the intense part of the code.

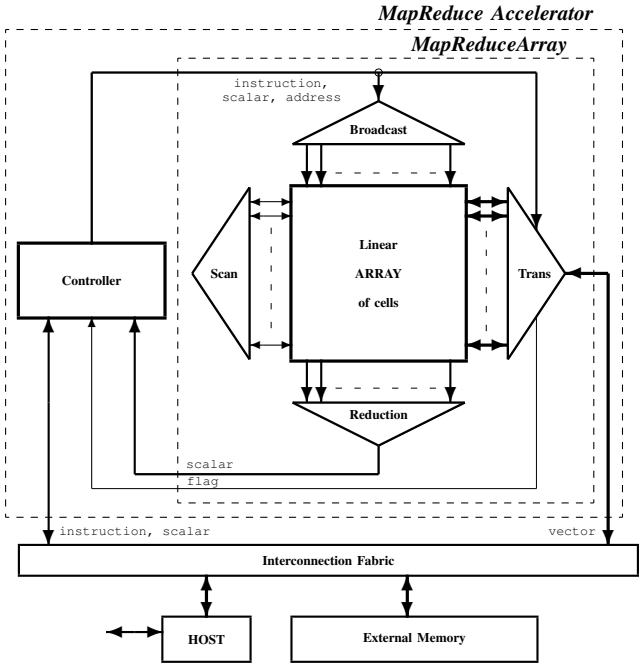

Fig. 2. The hybrid computing system.

The MRA has two sections:

- the scalar section: **Controller** – a processing element with the associated data memory and program memory (in each location stores a pair of instructions, one for Controller and another for ARRAY)
- the vector section: **Map-Reduce Array** – the core of the accelerator performing predicated vector operations.

The cellular system **Map-Reduce Array** consists of:

- the linear **ARRAY** of p cells, $CELL_0$, $CELL_1, \ldots,$ $CELL_{p-1}$, each containing:
  - MEM: the local data memory of m scalars
  - ENG: a scalar execution unit of n bits which executes, if the cell is active, the instruction broadcasted in each clock cycle by Controller
- the **Trans** unit: transfers vectors, of w scalars, between the ARRAY and MEMORY
- the **Broadcast** unit: a *log*-depth pipelined distribution network used to send in each clock cycle, from Control to ARRAY, the current instruction accompanied, if needed, by an address and a scalar
- the **Reduce** unit: a *log*-depth pipelined reduction network with arithmetic and logic operations (`add`, `max`, `or`, ...)
- the **Scan** unit: closes a global loop over the cells of ARRAY in order to provide in each cell global information.

In each clock cycle, in ARRAY can be executed p operations, in Reduce $p-1$ operations and in Control one operation, thus the peak performance of the MRA unit is $2pOPS/cycle$. There are few types of parallelism in MRA:

- inside of ARRAY
- inside of Reduce
- between ARRAY and Reduce
- between the *control* process in Control and the *computation* in ARRAY & Reduce

These multiple forms of parallelism allow the accelerator to provide sometimes super-linear accelerations.

B. Architecture

The Map-Reduce architecture of our accelerator is due to the ARRAY, able to execute operations *mapped* along its cells, and to the Reduce network able to *reduce* a vector to a scalar sent to the Control unit.

The accelerator operates on two data structures: scalars, stored in Controller's *scalar memory*, and vectors, stored in the *vector memory* distributed along the ARRAY's cells. Because the execution units in Control and in each cell are accumulator based the registers of the accelerator are:

- in Control: *pc*, *acc*, *cr*, *addr* used to address the local data memory
- in ARRAY:
  - $B = \langle b_0, b_1, \ldots, b_{p-1} \rangle$: a Boolean vector, used to activate the cells of the MAP array (the cell i is active only if $b_i = 1$, else cell i ignores the instruction issued in the current cycle by Control)
  - $IX = \langle 0, 1, \ldots, p-1 \rangle$: the constant vector `index`, used to identify each cell
  - $ACC = \langle acc_0, acc_1, \ldots, acc_{p-1} \rangle$: accumulator vector, used as operand and as destination for the result
  - $CR = \langle cr_0, cr_1, \ldots, cr_{p-1} \rangle$: carry vector
  - $ADDR = \langle addr_0, addr_1, \ldots, addr_{p-1} \rangle$: address vector, used to address in the local memories *mem*.

The local memory distributed along the cells is represented as a $(p \times m)$-component matrix M. Each line of M is a *horizontal vector*:

$$V_i = \langle s_{0i}, s_{1i}, \ldots, s_{(p-1)i} \rangle$$

for $i = 0, 1, \ldots, m-1$, while each column is a *vertical vector*:

$$W_j = \langle s_{j0}, s_{j1}, \ldots, s_{j(m-1)} \rangle$$

for $i = 0, 1, \ldots, p, -1$.

The ISA of MRA is the Cartesian product of two ISAs:

$$ISA_{MRA} = cISA \times aISA$$

where *cISA* is executed by Control, while *aISA* is executed in ARRAY.

The arithmetic and logic operations are the same in the two subsets. In *cISA* the operations are defined on scalars, while

in *aISA* are executed on vectors. The instructions are of the form:

$$acc <= acc\ OP\ operand$$

in CONTROL, and

$$acc_i <= b_i\ ?\ acc_i\ OP\ operand_i\ :\ acc_i$$

where *OP* represents an arithmetic or logic operation and *operand* and $operand_i$ are selected in seven modes. Let us exemplify them for the ADD operation in any execution unit in ARRAY's cells or in Control:

```
VADD(val)  : acc <= acc + val
ADD(val)   : acc <= acc + mem[val]
RADD(val)  : acc <= acc + mem[val + addr]
RIADD(val): acc <= acc + mem[val + addr]
             addr <= value + addr
CADD       : acc <= acc + coOp
CAADD      : acc <= acc + mem[coOp]
CRADD      : acc <= acc + mem[coOp + addr]
```

where: `val` is the immediate value, and `coOp` (co-operand) is `acc` for the cells from AARRAY, while for Control are the outputs of the reduction network Reduce selected by `val`:

- $redSum = \sum_0^{p-1} acc_i \times b_i$ for `val = 0`
- $redMax = MAX_0^{p-1} acc_i \times b_i$ for `val = 1`
- $redBool = \sum_0^{p-1} b_i$ for `val = 3`

The main differences between the two subsets are related to the control instructions. The control instructions for Control are the standard conditioned or unconditioned jumps and branches. In ARRAY, *aISA* provides a spatial control using predicated operations. It is based on operations applied on the Boolean vector *B*. The main spatial control operations are:

- `activate` : $b_i <= 1$, for $i = 0, 1, \ldots, p-1$
- `where (cond)` : $b_i <= (b_i\ \&\ cond_i)\ ?\ 1 : 0$
- `endwhere` : restore *B* to the previous value

```
/*****************************************************
Index vector is multiplied with the sum of its odd
components.
Number of cells: 512
*****************************************************/
      cNOP;          ACTIVATE;   // activate all cells
      cNOP;          IXLOAD;     // acc<=1; acc[i] <= i
      cNOP;          VAND(3);    // acc[i]<=acc[i] & 0...11
      cNOP;          VSUB(2);    // acc[i]<=acc[i] - 2
      cVLOAD(8);     WHEREZERO;  // acc<=8; where i=2(mod3)
LB(1) cBRNZDEC(1);NOP;           // wait-loop for latency
      cNOP;          ENDWHERE;   // reactivate all cells
      cCLOAD(0);     IXLOD;      // acc<=redAdd; acc[i]<=i
      cNOP;          CMULT;      // acc[i]<=acc[i] * acc
```

Fig. 3. Example of code executed by MRA. The left column contains instructions, prefixed with `c`, for CONTROL, while the right column contains instructions for the MAP array.

Let us take an example (see Figure 3) of a simple code which multiplies the index vector *IX* with the sum of its $2(mod3)$ components The line labeled with 1 is the wait-loop for the latency introduced by the *log*-depth Reduction section. We consider $p = 512$, then between the cycle when the odd accumulators of ARRAY are selected and the cycle when the reduction sum is loaded in Control's accumulator we allow a latency of 9 cycles.

The execution time, for $p = 512$, of the program just exemplified is: $T(p) = 7 + log_2 p = 16$. In this number of cycles are executed 512 VANDs, 512 VSUBs, 511 ADDs, and 512 MULTs, i.e., more than 256 arithmetic and logic operations per cycle.

C. Physical Design

The physical implementation is evaluated for $p = 2048$, $m = 4096$, $n = 32$ in standard cell $28nm$ technology. For $f_{clock} = 1GHz$ the resulting area is $92mm \times 92mm$ and the power consumption is plotted in Figure 4.

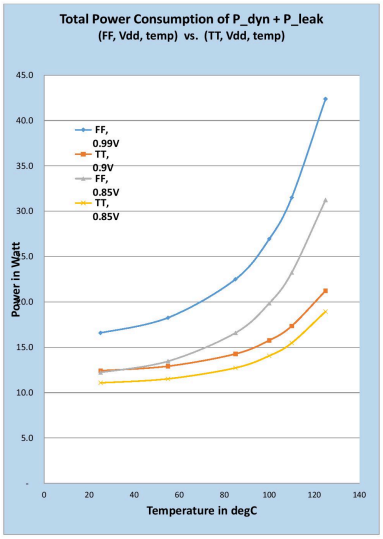

Fig. 4. Power consumption evaluated for out MRA.

The power consumption scales good enough with *p*. Therefore, at 100^oC we can select an implementation for our accelerator with $p = 512$ working at $< 5Watt$.

V. Algorithm Implementation

The implementation of the stereo match algorithm, published in [13] [14] and based on [6], has two versions: the accurate one and the fast one. Both versions process the pair of stereo frames taking the image in patches of 9×9 gray pixels. The patches are taken with a stride of 1 on both directions. Therefore, each frames of 1240×376 pixels, considered in the experiments of the authors, is divided in 1232×368 patches. Each pair of patches, one from left frame and another from the right frame, are processed by a CNN. The fast version of the algorithm contains only convolutional layers, while the accurate one is dominated by FNN.

A. Main operations

We describe in this subsection two main operations performed in MRA; one helps in loading data in the vector memory, while the other performs matrix-vector multiplication.

1) Matrix transpose: is applied on N full horizontal vectors, $V_i, V_{i+1}, \ldots, V_{i+N}$, organized in $\lfloor p/N \rfloor$ $N \times N$ matrices. The first matrix is loaded starting from $CELL_f$, with $f < N$, i.e., the element e_{11} of the first matrix is s_{if}. Let us call s_{if} the starting point of the transpose operation $T_N(if)$ which transpose $\lfloor p/N \rfloor$ matrices stored in N vectors starting with the vector V_i in $CELL_f$. The execution time for this operation is:

$$t_{transpose}(N) = N^2 + 29N - 7$$

which translates in

$$(N^2 + 29N - 7)/(N^2 \times \lfloor p/N \rfloor)\ cycles/element$$

For our 9×9 matrices and an ARRAY of $p = 1024$ cells, results, compared to a mono-core execution, an acceleration of 189.

2) Matrix-vector multiplication: of a $N \times M$ matrix with a M-component vector consists in three main operations performed in parallel in our MRA on distinct hardware resources:

- control, performed by the CONTROL unit
- multiplication, performed in the ARRAY section
- addition, performed in the REDUCE section

The main problem solved for optimizing the algorithm was to avoid the effect of the latency, of $O(log\ p)$, introduced by the REDUCE section. An additional p-stage of n-bit shift register, `shiftReg[[0:(1<<x)-1]][n-1:0]`, introduced in the organization of ARRAY, allows to insert back in ARRAY the output of REDUCE avoiding an explicit load in the CONTROL's accumulator. Thus, instead of providing each component of the resulting vector with a latency in $O(log\ p)$, only the result vector is provided with a $O(log\ p)$ latency in `shiftReg`.

The program in assembly language is listed in Figure 5, where the instruction `IP(255);` is special as it is executed in both, CONTROL section and ARRAY section, does the following:

```
acc[i] <= acc[i];
redReg[i] <= b[i] ? acc[i] x mem[val + addr[i]] : 0;
addr[i] <= addr[i] + val;
shiftReg[i] <= (i=0) ? redSum : shiftReg[i-1];
```

where, `redReg[i]` stores the input in the REDUCE section from each cell of the ARRAY. The instruction `cBRNZDEC('L)` introduces a delay according to p.

The execution time for matrix-vector multiplication is

$$T(N) = N + 2 + log_2 p \in O(N)$$

for $N \leq p$ and $M \leq m$.

Compared with a mono-core engine the ***acceleration is supra-linear***, because besides the parallelism offered by the many-cell structure of ARRAY, we benefit by the parallelism in REDUCE and by the control running on Controller.

The use of this function is considered for the case when the matrices defining an FNN are very big. Then the supra-linear acceleration will allow high accelerations of the application.

B. Loading Data in Accelerator

The size of the convolutional or fully connected layers used in both, accurate and fast versions of the algorithm allow us to use our MRA in a pure SIMD mode if data is appropriately distributed in cells. If each patch can be loaded and processed in one cell, then the degree of parallelism achieved in running CNN on MRA becomes very high.

```
/*****************************************************
FUNCTION NAME: Matrix-vector multiplication
The function multiplies a NxM matrix with a vector
Initial:
 addr[i] = M+1 : M is address of the last matrix's line
 acc[i] = V[i] : the vector
Final: acc[i] = result
*****************************************************/
//Parameters:
 `define N    13      // matrix edge size
 `define S    (x-1)   // latency size because p = 2^x
//Labels:
 `define M    1       // main loop label
 `define L    2       // latency loop label

        cVLOAD(`N);     NOP;       // acc <= N;
LB(`M); cBRNZDEC(`M); IP(255);    // loop control; IP
        cVLOAD(`S);     NOP;       // init latency loop
LB(`L); cBRNZDEC(`L); NOP;         // latency loop
        cNOP;           SRLOAD;    // result in acc[i]
```

Fig. 5. Code for matrix-vector multiplication. For big N the program is executed in $\sim 2N$ cycles.

In this subsection we show how the patches from the two frames with stride 1 on both dimensions can be loaded as pairs of vertical vectors in the vector memory of our MRA.

The loading process has two phases: the frame is loaded as it is in the vector memory (each line as a horizontal vector), and then the $F \times F$ patches of pixels are transformed in $(F \times F)$-component vertical vectors. Each patch is indexed by the pair of indexes of its first pixel (on the first line and the first column of the patch): P_{ij}.

The first phase is a simple data transfer from External Memory where the two frames are stored linearly, lines after lines or column after columns. Results a $W \times H$ matrix stored in H horizontal vectors:
$V_1 = \langle s_{11}, s_{21}, \ldots, s_{W1} \rangle$
$V_2 = \langle s_{12}, s_{22}, \ldots, s_{W2} \rangle$
...
$V_H = \langle s_{1W}, s_{2W}, \ldots, s_{WH} \rangle$

For the second phase, because we consider $S = 1$ and $P = 0$, there are a number of $(W - F + 1) \times (H - F + 1)$ patches to be converted in vertical vectors.

In order to cover vertically the patches of the two frames, the following sequence of sequences of the transpose operations must be applied $\lfloor H/F \rfloor$:

$$T_N((i+1)1), T_N((i+2)1), \ldots, T_N((i+F)1)$$

for $i = 0, F, 2F, \ldots, (\lfloor H/F \rfloor - 1)$ For $i = 1$ the operation restore the patches from one frame
$P_{11}, P_{12}, \ldots, P_{1F}$
$P_{(F+1)1}, P_{(F+1)2}, \ldots, P_{(F+1)F}$
$P_{(2F+1)1}, P_{(2F+1)2}, \ldots, P_{(2F+1)F}$
...
as vertical vectors in

$CELL_0, CELL_1, \ldots, CELL_{F-1}$
$CELL_F, CELL_{F+1}, \ldots, CELL_{2F-1}$
$CELL_{2F}, CELL_{2F+1}, \ldots, CELL_{3F-1}$
...
and so on for the next values of i.

In order to cover horizontally the patches of the two frames, the vertical cover, defined by

$$T_N((i+1)j), T_N((i+2)j), \ldots, T_N((i+F)j)$$

must be repeated F times, as follows for $J = 1, 2, \ldots, F-1$.

C. Computation

After each application of the transpositions

$$T_N((i+1)j), T_N((i+2)j), \ldots, T_N((i+F)j)$$

for each frame, resulting two 81-component vertical vectors in each cell, the content of the vector memory of ARRAY is prepared for the CNN computation in a pure SIMD mode.

D. Accurate Architecture

Following the system proposed by Žbontar and LeCun, we evaluated the accurate architecture for the stereo matching cost. The CNN considered by the authors has one convolutional layer followed by 7 FNN layers (see Figure 2 in [13]).

The load of pixels and the rearrangement, using the transpose operation, takes around 400 cycles for a pair of patches, while the associated computation is 1.6 GOPs. The 600,000 weights used to define all the 8 layers of the CNN can be used in parallel for all $\lfloor p/F \rfloor \times F$ pairs of patches involved simultaneously in the computation. Thus, for this application of the CNN configuration we obtain a computational process which is not I/O bounded.

For all 1232×368 pairs of patches running an application which requests 10 frames/sec processing, the total amount of computation is $7.25\,GOPs/sec$. Therefore, the MRA system described in Subsection IV C is able to perform this computation in less than 11 sec with an actual performance of 0.33 from the peak performance of a circuit powered with $\sim 12\,Watt$. In [13] are reported 100 sec for experiments running on Nvidia GeForce GTX Titan GPU.

E. Fast Architecture

Following the suggestion of J. Žbontar and Y. LeCun from (see Figure 2 in [14]) we evaluated the computation of a CNN with 4 convolutional layers ended with the computation of the cosine similarity between the resulting two 64-component vectors. The computation starts from the same two frames and the same number of 1232×368 pairs of patches. The convolutional layers are, by turn, defined by $7 \times 7 \times 10$, $5 \times 5 \times 16$, $3 \times 3 \times 32$, and $1 \times 1 \times 64$ weights. Results 10 GOPs per pairs of frames. For 10 pairs of frames per second a reasonable 100 GOPs/sec computational power is requested.

VI. Evaluation

Based on the evaluation made for the physical design and on the simulation in Vivado environment we are able to conclude about our proposal. Let us consider that a real application for stereo vision, using the fast architecture, requests a computational power higher than 100 GOPs/sec, let say 300 GOPs/sec. With an actual performance of 0.3 from the peak performance, a version of 1024-cell of our MRA, before described, will solve the problem working at around 6 Watt.

VII. Conclusion

The accurate architecture is, for the time being, far from the performance achieved by low power market product. But the fast architecture, implemented using our MRA, qualifies for mass production and for high temperature environment (such as the interior of a car in the middle of a desert).

Compared with Google's TPU, a custom ASIC that normally beats, for a specific application, a general purpose processing unit, our MRA, a general purpose programmable accelerator, performs good enough, in the same range. $GOPs/sec/mm^2$ is 140 for TPU and 92 for MRU, while $GOPs/sec/Watt$ is 1230 for TPU and 670 for MRU. Even if the numbers are a little smaller, the advantage of programmability and universality can not be neglected.

References

[1] D. C. Cireşan, U. Meier, J. Masci, L. M. Gambardella, and J. Schmidhuber, "Flexible, high performance convolutional neural networks for image classification" in *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence*, pp. 1237–1242, 2011.

[2] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, et al., "In-Datacenter Performance Analysis of a Tensor Processing Unit^{TM}", To appear at the *44th International Symposium on Computer Architecture (ISCA)*, Toronto, Canada, June 26, 2017. [Online]. Available: https://drive.google.com/file/d/0Bx4hafXDDq2EMzRNcy1vSUxtcEk/view

[3] A. Karpathy, "Cs231n: Convolutional neural networks for visual recognition". [Online]. Available: http://cs231n.github.io/

[4] S. Kleene, "General recursive functions of natural numbers", iin *Mathematische Annalen*, vol. 112, no. 1, pp. 727-742, 1936.

[5] H. T. Kung, "Why Systolic Architecture", in *Computer*, 1982, pp 37-46.

[6] Y. LeCun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-Based Learning Applied to Document Recognition", in *Proc. of the IEEE*, Nov. 1998, pp. 1-46.

[7] M. Maliţa, G. M. Ştefan, and D. Thiébaut, "Not multi-, but many-core: Designing integral parallel architectures for embedded computations" in *ACM SIGARCH Computer Architecture News*, 35(5):32–38, Dec. 2007.

[8] K. Ovtcharov, O. Ruwase, J. Kim, J. Fowers, K. Strauss and E. Chung, "Accelerating Deep Convolutional Neural Networks Using Specialized Hardware", *Microsoft Research*, February 23, 2015.

[9] G. Raina, "Deep Convolutional Netvork evaluation on the Xeon Phi: Where Subword Parallelism meets Many-Core". *Eindhoven University of Technology*, 2016. [Online]. Available: http://repository.tue.nl/844256

[10] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection", in *Cornell Univ. Library*, 2016.

[11] G. M. Ştefan, A. Sheel, B. Mîţu, T. Thomson, and D. Tomescu, "The CA1024: A Fully Programmable System-On-Chip for Cost-Effective HDTV Media Processing" in *Stanford University: Hot Chips: A Symposium on High Performance Chips*, August 2006. [Online]. Available: `https://youtu.be/HMLT4EpKBAw` at 35:00.

[12] G. M. Ştefan and M. Maliţa, "Can one-chip parallel computing be liberated from ad hoc solutions? a computation model based approach and its implementation" in *18th Inter. Conf. on Ciruits, Systems, Communications and Computers*, Santorini, Greece, pages 582–597, July 2015.

[13] J. Žbontar and Y. LeCun, "Computing the stereo matching cost with a convolutional neural network", in *Cornell University Library*, 2015.

[14] J. Žbontar and Y. LeCun, "Stereo Matching by Training a Convolutional Neural Network to Compare Image Patches", in *Journal of Machine Learning Research*, 17 (2016) pp. 1-32.