ROMANIAN JOURNAL OF INFORMATION SCIENCE AND TECHNOLOGY
Volume 20, Number 3, 2017, 186–197

Hybrid Accelerator with MapReduce Architecture for Convolutional Neural Networks

David Mihăiţă[1] and Gheorghe M. Ştefan[1]

[1]Politehnica University of Bucharest
Email: gheorghe.stefan@upb.ro

Abstract. The current hybrid architectures used to train and implement Convolutional Neural Networks (CNN) are based on Nvidia GPU or Intel MIC accelerators. They are marked by limitations due to their too general and *ad hoc* structure and architecture. We propose an accelerator with a Map-Reduce architecture. The FPGA version of our proposal is considered for accelerating the training process, while the ASIC versions, based on experiments done in the FPGA environment, are targeted for low-energy consumption mass product applications. The paper emphasizes a specific set of functions for the CNN used in Machine Learning (ML) applications, and presents the resulting structural and architectural requirements. The main stages used in a pipe of functions destined to ML are: padding, convolution, pooling and fully connected neural network. The actual applications use these functions in a big variety of configurations. Thus, it worths to define, for training and running a CNN, a programmable accelerator. The paper describes the organization and the architecture of a hybrid system based on Map-Reduce architecture. The energy consumption is estimated, by simulation, for the ASIC version. We conclude that the Map-Reduce approach provides an appropriate solution for accelerating various ML applications.

1. Introduction

The new look of Artificial Intelligence (AI) is Machine Learning (ML), and the preferred embodiment is CNN. Because the spectrum of applications and of solutions is very wide, and the required performance is very high, we have to implement the CNNs on very flexible computational machines designed as hybrid systems putting together a general purpose CPU and a parallel accelerator.

The initial steps, in this process of using AI as a ubiquitous function, are done using computational devices that where developed in a different conceptual context. Using a graphics accelerator or a general purpose many-core engine provides some acceleration, but the current investigations show that the acceleration is too small and the energy involved is to high, i.e., the training time for CNN is too long and the power consumption unacceptable for CNN-based products destined for the big market.

Our proposal is to use an accelerator organized as a Map-Reduce system with an architecture optimized for linear algebra applications. The resulting hybrid system performs very well on matrix-vector operations. Because the accelerator is implemented in FPGA technology, its organization and architecture is parameterized and is configurable. In this initial stage of the project, the accelerator is used for the training process and then, if a mass production is considered, as experimental prototype for the ASIC version of the hybrid system.

The main target of our approach is to provide a many-cell system able to work with a high degree of parallelism in the envisaged application domain.

The second section shows the main results published about the use of hybrid systems in training and running CNNs. Then, in the next section, the main functionality involved in CNN implementation is reviewed. The fourth section describes the proposed accelerator. The next section shows how the system is used to implement the main functionality required by CNN applications. The evaluation of the performance in using our Map-Reduce accelerator is provided in the sixth section. Final comments conclude the paper.

2. State of the Art

The currently used accelerators for CNN applications are based on Nvidia or Intel's Xeon Phi parallel engines. Recently published results emphasize a very low use of their big peak performance. From few TFLOP/s only a small fraction is activated.

Real time object detection with Titan X GPU, for 40-90 fps, uses maximum 63 GFLOPs/sec [1] [2] form the peak performance of 6 TFLOPs/sec, while from Intel's i7 CPU with 112 GFLOPs/sec a much bigger fraction, 32%, is used in running the same application.

With a Xeon Phi [3] accelerator, having 57 cores and peak performance of 2 TFLOPs/sec, the same application uses only 0.48 GFLOPs/sec from each core which is able to provide 35.2 GFLOPs/sec.

Why from 32% use of the peak performance for CPU we go around 1% use for the accelerators available on the shelf? We suppose it is about an architectural inadequacy.

3. Convolutional Neural Networks

A CNN consists of a number of convolutional layers followed by few fully connected neural network layers. The size, the number of layers and the weight of the convolutional layers involved in the organization of a CNN differ very much from an application to another. In contrast to the standard neural layer, characterized by a two-dimension matrix, a convolutional layer has a more complex structure. The parameters of a convolutional layer can be summarized [4] as follows:

- the input of a convolutional layer receives a volume of size $W_1 \times H_1 \times D_1$ values
- the definition requires four hyper-parameters:
  - number of filters K,
  - their spatial extent of the receptive field, F – with $F << W_1$ and $F << H_1$ –, specifies the size of the edge of the squared region used to explore the input volume
  - the stride, S – with $S \leq F$ –, used to explore the input

  - the amount of zero padding, P, used to expand the input volume

- the output volume of size $W_2 \times H_2 \times D_2$, where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$
  - $D_2 = K$
- with the same parameter sharing over the receptive fields, it introduces $F \times F \times D_1$ weights per filter, for a total of $(F \times F \times D_1) \times K$ 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

$$(F \times F \times D_1) \times K << W_1 \times H_1 \times D_1.$$

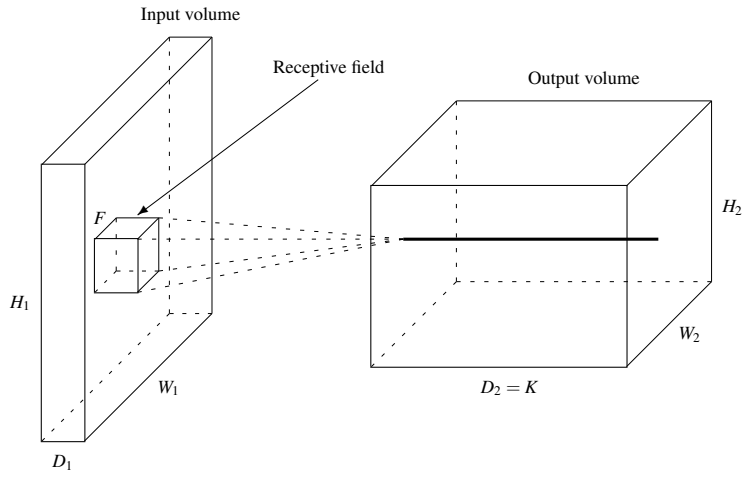

Fig. 1. Convolutional layer. Each receptive field from the input layer generates K elements of the output volume.

The operation performed in a convolutional layer is applied to each receptive field (see Figure 1) from the input volume represented as a vector X of $r = F \times F \times D_1$ 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 are 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 is loaded only once in accelerator and it is used many times, in contrast with the weight matrix $\mathbf{W}$ which define a fully connected NN layer, which is loaded and used only once. Each receptive field is used to generate K elements in the output volume.

In the last few years different organizations of CNN were investigated for real applications [5] [6] [7] [8]. The lesson provided by all the current approaches is the very strong dependence of the organization of CNN by the actual applications.

4. Map-Reduce Organization and Architecture

The computation requested by CNN is dominated by inner product (IP) operation in both, the convolutional and the fully connected layers. The IP operation consists on two levels: 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 [9]. In [10] [11] we tried to prove that our cellular structure has an integral parallel architecture with a wide spectrum of applications. In [10] the Map-Reduce approach is based on a mathematical model of computation [13].

4.1. Organization

A hybrid organization, according to the saying *"10% of the code runs 90% of the time"*, consists of two parts: the CPU – for running the complex code (the size of code in the same range with the executions time) – and the Accelerator – for running the intense code (the size of code much smaller than the execution time).

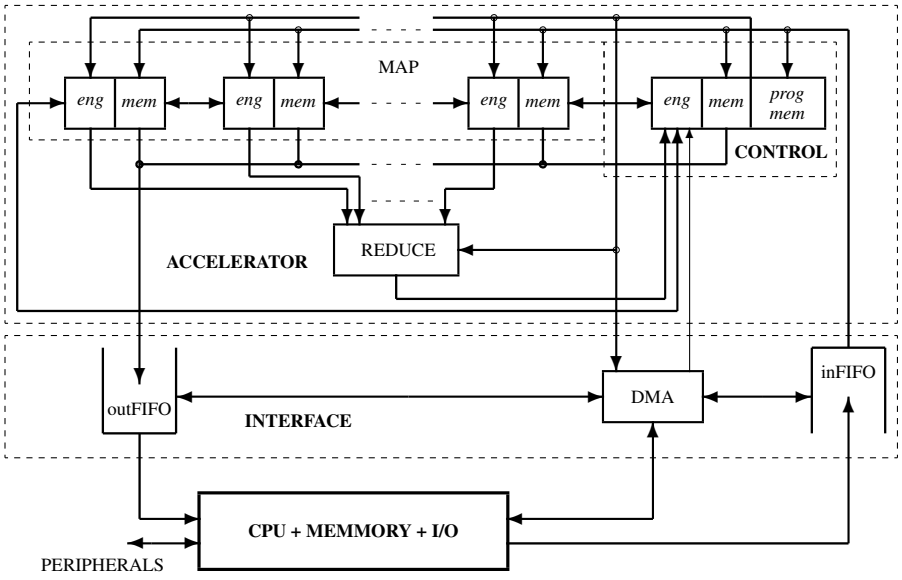

Fig. 2. Hybrid system based on a Map-Reduce accelerator.

In Figure 2 is represented the organization of our hybrid system with the Map-Reduce Accelerator (MRA). The accelerator has three main parts:

- the MAP array of p identical cells, each with its own local memory (*mem*) and the associated engine (*eng*)
- the REDUCE *log*-depth tree-like net
- the CONTROL unit used to issue in each cycle an instruction to be executed in the MAP array.

The interface of MRA loads the program memory of the controller and transfers data between the system memory and the locally distributed memory along the cells.

There are four main processes which can run simultaneously in MRA:

- control and sequential computation
- massive parallel computation
- reduction computation
- data transfer

The host computer, loads the program in CONTROL and sees the accelerator as a library of functions accessed from the program it runs.

4.2. Architecture

The basic data structure inside the MAP section is the vector of scalars. The entire content of the local memory distributed along the cells is represented by a $p \times m$ matrix M. Each line in M is a *horizontal vector*:

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

for $i = 0, 1, \ldots, m-1$. Each column in M is a *vertical vector*:

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

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

There are few specific horizontal vectors distributed along the array of *eng* units:

- $IX = \langle 0, 1, \ldots, p-1 \rangle$: the constant vector `index`, used to identify each cell
- $B = \langle b_0, b_1, \ldots, b_{p-1} \rangle$: a Boolean vector, used to activate the cells of the MAP array (the cell i is active only if $b_i = 1$, else the cell i ignores the instruction issued in the current cycle by CONTROL)
- $ACC = \langle acc_0, acc_1, \ldots, acc_{p-1} \rangle$: 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*.

Correspondingly, in CONTROL there are the scalar resources: *acc*, *cr*, *addr*. The option to start with an accumulator centered architecture is justified by the initial stage of the development of the MRA architecture.

The instruction set architecture (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 the array of cells.

The arithmetic and logic operations are the same in the two sets. In *cISA* these operations are defined on scalars, while in *aISA* are executed on vectors. The instructions are of form:

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

in CONTROL, and

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

where *OP* represent an arithmetic or logic operation and $operand$ and $operand_i$ are selected in seven modes. For example, ADD operation in any *eng* (from MAP or from CONTROL) is performed in one of the following ways:

```
VADD(value) : immediate value ADD
    acc <= acc + value
ADD(value) : direct ADD
    acc <= acc + mem[value]
RADD(value) : indirect ADD
    acc <= acc + mem[value + addr]
RIADD(value) : indirect ADD with increment
    acc <= acc + mem[value + addr]
    addr <= value + addr
CADD : co-operand ADD
    acc <= acc + coOperand
CAADD : co-operand absolute ADD
    acc <= acc + mem[coOperand]
CRADD : co-operand relative ADD
    acc <= acc + mem[coOperand + addr]
```

where `coOperand` is `acc` for the cells from MAP, while for CONTROL are the outputs of the reduction network REDUCE:

- $redSum = \sum_0^{p-1} acc_i \times b_i$
- $redMax = MAX_0^{p-1} acc_i \times b_i$
- $redBool = \sum_0^{p-1} b_i$

The main differences are in the control instructions subsets. The control instructions for CONTROL are the standard conditioned or unconditioned jumps and branches. In MAP, *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

Let us take an example (see Figure 3) of a simple code which multiplies the index vector *IX* with the sum of its odd components The line labeled with 1 is the wait loop for the latency introduced by the *log*-depth reduction network. In this example we consider $p = 512$, then

```
/*********************************************************
Index vector is multiplied with the sum of its odd
components.

Number of cells: 512
*********************************************************/
       cNOP;           ACTIVATE;   // activate all cells
       cVLOAD(1);      IXLOAD;     // acc<=1; load index
       cNOP;           CAND;       // acc[i]<=acc[i] & acc
       cVLOAD(8);      WHEREZERO;  // acc<=4; only even cells
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] * redAdd
```

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.

between the cycle when the odd accumulators of MAP are selected and the cycle when the reduction sum is loaded in CONTROL's accumulator we allow a latency of 9 cycles (8 cycles looping on the line labeled with `LB(1)` and one on the line `cNOP; ENDWHERE;`).

The execution time, for $p = 512$, of the program just exemplified is: $T(p) = 6 + log_2 p = 15$. In this number of cycles are executed 512 ANDs, 511 ADDs, and 512 MULTs, i.e., more than 100 arithmetic and logic operations per cycle.

5. CNN Functionality on Map-Reduce Accelerator

The main operation involved in CNN applications is the inner product operation (IP) used for matrix-vector multiplication. However, there is a big difference between the way we use matrix-vector multiplication in the convolutional levels or in the fully connected levels.

5.1. Matrix-vector multiplication on the convolutional layers

If each receptive fields in the input volume is considered a vector $\mathbf{V_i}$ of $F \times F \times D_1$ input components and the K filters are also vectors of the same number of parameters (weights), then we must multiply the weights matrix $\mathbf{M}$ of $(F \times F \times D_1) \times K$ size with the input vector associated to the receptive field. Results a K-component vector. The function f is applied to its components and results the K-component vector in the output volume (see the line in output volume in Figure 1). The matrix $\mathbf{M}$ is unique for a convolutional layer. Therefore, it is loaded only once for the computation of one convolutional layer. This is the easy problem. The hard problem is how to load the input volume in order to maximize the degree of parallelism and to minimize the multiplication of data.

There are three ways to load the input volume and the parameters of the K filters. Depending on the actual sizes of input volume we have three possibilities:

1. as D_1 matrices of $W \times H$ components, where $W \leq W_1$ and $H \leq H_1$; the parameters of the K filters are loaded as many $F \times (F \times K)$ matrices in each group of F cells
2. as maximum $(W_1F + 2P)/S + 1 + (H_1F + 2P)/S + 1$ vertical vectors of $F \times F \times D_1$ components; the parameters of the K filters are loaded only in the data memory of CONTROL
3. as maximum $(W_1F + 2P)/S + 1 + (H_1F + 2P)/S + 1$ horizontal vectors of $F \times F \times D_1$ components; the parameters of the K filters are loaded as horizontal vectors

In the first case the integral or fragmented load of the input volume in the distributed memory of the MAP section is performed by stridded bursts from MEMORY. For each receptive field are allocated F cells in MAP. The associated data is stored in $F \times K$ memory locations in each cell. Thus, in the p cells of MAP are loaded in $F \times K$ horizontal vectors p/F receptive fields from the input volume. The load is performed by stridded bursts from MEMORY. The stride is W_1 and the burst is $\lfloor p/F \rfloor \times F$. The load of the parameters for the K filters is done by distributing them from the CONTROL's data memory. The degree of parallelism in performing the computation, π, is diminished by the fact that $F-1$ final additions and the function f are performed only in $\lfloor p/F \rfloor$ cells. But, the majority of $F \times K$ multiplications and $F \times K - 1$ additions are done in parallel for p receptive fields. Results:

$$\pi \simeq \frac{2FK + (F-1)/F}{2FK + F}$$

Because the data transfer between MEMORY and MAP is transparent to the computation, if the system is featured with enough local data memory in each cell, then the data movement does not significantly affect the performance. If the local memory in MAP can not support this approach, then we must select one of the other two.

In the second case the degree of parallelism in the computational section of the algorithm is 1, but the input volume is loaded in a way which depends on the stride S. For $S = F$ we are in the most favourable situation, while for $S = 1$ we have the worst case. Besides the strided burst, used to load each receptive fields as a number of horizontal vectors, the matrix transpose operation is required to transform the horizontal vectors in vertical vectors. Then the MAP section can work, very efficiently, in pure SIMD[1] mode.

In the third case we are in the best situation from the computational point of view, because the matrix-vector multiplication will activate at maximum our organization: the REDUCE unit is used to perform the additions in parallel with the multiplications performed in the MAP section. The data is loaded in stridded bursts and the parameters for the K filters are loaded as vectors only once.

The selection of one of the above described three cases, for a MAP section of p cells and m-word cell's data memory, is done according to the sizes of the input volume, the size of the receptive field and the number of the filters involved in the convolutional layer. The main advantage of the computation in the convolutional layer is given by the fact that a big input volume can be easy fragmented to fit the limited size of the accelerator parameters p and m.

5.2. Matrix-vector multiplication on the fully connected layers

A fully connected layer of a CNN has the advantage of being modeled by the simple and consecrated operation of matrix-vector multiplication. The only implementation problem we are

[1] Single Instruction Multiple Data, according to Flynn's taxonomy.

faced for this part of the computational pipe is the big size of the weight matrices involved in some applications. For example, in [14] is investigated a CNN with the last three layers fully connected. Almost all of the total 60,000,000 parameters involved are associated to these layers, while in [8] the application involves small matrices of weights which can be kept inside the accelerator's memory. In the first case, if the full pipeline is run completely for each input volume, then the computation is I/O bounded and the parallel accelerator has a small effect in accelerating the computation, while in the second the performance is not limited by the data transfer.

There are many approaches in the first case. One is to increase the bandwidth between the MAP array and MEMORY. But, for $p > 64$ it is hard to get rid of I/O limitation. Another solution is to run the convolutional part of the CNN pipe n times and then to use in the last fully connected layers n times the weights loaded only once. The price paid for the first solution is power consumption, and for the second the latency.

The more radical solutions for the I/O bottleneck are two: to increase the size of the local data memory in cells and to increase the number of fully connected layers in CNN. Also, we must remember that the concept of deep neural network occurred triggered by the necessity to reduce the size of the layers in neural networks with a small number of layers. Then, we must work on the architecture of CNN in order to define fully connected layers with reduced number of neurons.

6. Evaluation

In the previous section we emphasized the following frequently used functions which must be accelerated for CNN applications:

- matrix-vector multiplication
- matrix transpose
- stridded burst data transfer between the MAP array an MEMORY

6.1. Matrix-vector multiplication

The algorithm for multiplication of $N \times M$ matrix with a M-component vector consists in three main operations:

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

All these three operations are performed in parallel on distinct hardware resources. 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 shift register introduced in the organization of MAP allows to insert 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. The program in assembly language is listed in Figure 4, where the instruction `cCPUSHL(0);` pushes the inner product

```
/*************************************************************
FUNCTION NAME: Matrix-vector multiplication

The function multiplies a NxN matrix with a vector
Initial:
    addr[i] = M : address of the last line in matrix
    acc[i] = V[i] : the vector
Final:
    acc[i] = result
*************************************************************/
//Parameters:
 `define N     13        // matrix edge size
 `define W     0         // working space: to save vector
 `define S     (x-2)     // latency size because p = 2^x
//Labels:
 `define M     1         // main loop label
 `define L     2         // latency loop label

        cNOP;           STORE(`W);   // mem[i][W]<=acc[i]
        cVLOAD(`N);     RLOAD(0);    // acc <= N;
                                     // acc[i]<= last line
        cVSUB(1);       MULT(`W);    // acc <= N-1;
                                     // acc[i]<=acc[i]*mem[i][W]
LB(`M); cCPUSHL(0);     RILOAD(255); // push redSum;
                                     // acc[i]<= previous line
        cBRNZDEC(`M);   MULT(`W);    // loop control;
                                     // acc[i]<=acc[i]*mem[i][W]
        cVLOAD(`S);     NOP;         // init latency loop
LB(`L); cBRNZDEC(`L);   NOP;         // latency loop
        cNOP;           SRLOAD;      // result in acc[i]
```

Fig. 4. The program for matrix-vector multiplication. For big N the program is executed in $\sim 2N$ cycles.

provided by REDUCE in the mentioned shift register, and after N runs of the two steps loop the instruction `cBRNZDEC(`L)` introduces a delay according to the size, p, of the MAP array.

The execution time for matrix-vector multiplication is

$$T(N) = 2N + 4 + log_2 p \in O(N)$$

for $N \leq p$.

Compared with a mono-core engine the acceleration is supra-linear, because besides the parallelism offered by the many-cell structure of MAP, we benefit by the parallelism in REDUCE and by the control running on a different physical resource, the controller.

6.2. Matrix transpose

The matrix transpose operation is used in conjunction with the data transfer operation. It is necessary to rearrange data loaded from MEMORY where the arrays of data are streams of vectors representing the lines of different matrices.

The execution time, measured on our simulator, for $N \times N$ matrix transpose operation is

$$T(N) = N^2 + 29N - 7 \in O(N^2)$$

If $N \leq p/2$, then more than one matrix can be transposed in parallel. The mono-core execution time of this operation is also in $O(N^2)$. Thus, the process is significantly accelerated only for matrices with $N \leq p/2$, which is the case in many applications. But, even if $N = p$, the theoretical performance is not diminished because the transfer time is also in $O(N^2)$ for a $N \times N$ matrix.

6.3. Stridded burst transfer

The linear representation of the two-dimension data in MEMORY must be accessed in CNN applications at both, convolutional and fully connected layers, as a stream of small or big two-dimension patches. In MEMORY, the information belonging to a small patch is distributed stridded with the distance between the beginning of two lines. Then, in order to load a $F \times F$ patch from a $W \times H$ frame we must order F bursts of F components stridded at W. Thus, a patch of $F \times F$ components is loaded as a vector of length $F \times F$ distributed along $F \times F$ cells in the MAP array. If $(F \times F) \leq p/2$, then more than one patch can be loaded on a horizontal vector. If needed, the $F \times F$ horizontal vectors can be rearranged as vertical vectors transposing $(F \times F) \times (F \times F)$ matrices.

7. Final Comments

Our proposal is a development environment for AI applications implemented on CNN as ML tools. The domain is an emergent one and appropriate tools are needed for experimenting, training and designing. Hybrid computing seems to be a very good computational environment in this stage of development. It looks like the currently used accelerators do not fit well for this application domain because their huge computational power expressed in TFLOPs/sec can not be fully put to work.

Our proposal – Map-Reduce Accelerator – solves very well the computational aspects, like matrix-vector multiplication, offers a good environment for stridded data transfer in bursts and for transpose operations, but has problems, to be addressed by future research, related with I/O bounded operations.

Acknowledgements

The authors got a lot of support from the main technical contributors to the development of the ConnexArray^{TM} technology, the CA1024 chip, the associated language, and its first application: Emanuele Altieri, Frank Ho, Mihaela Maliţa, Bogdan Mîţu, Marius Stoian, Dominique Thiebaut, Tom Thomson, Dan Tomescu. The comments received from anonymous reviewers helped a lot to improve this paper.

References

[1] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, *You only look once: Unified, real-time object detection*. [Online]. Available: `https://pjreddie.com/media/files/papers/yolo.pdf`

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

[3] 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`

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

[5] M. D. Zeiler and R. Fergus, *Visualizing and understanding convolutional networks*, Cornell University Library, 2013.

[6] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, *Going deeper with convolutions*, Cornell University Library, 2014.

[7] K. Simonyan and A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, Cornell University Library, 2015.

[8] J. Zbontar and Y. LeCun, *Computing the stereo matching cost with a convolutional neural network*, Cornell University Library, 2015.

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

[10] M. Malita, G. M. Stefan, and D. Thiebaut, *Not multi-, but many-core: Designing integral parallel architectures for embedded computation*, ACM SIGARCH Computer Architecture News, vol. 35, no. 5, pp. 32–38, 2007.

[11] R. Andonie and M. Malita, *The connex array as a neural network accelerator*, Proceeding CI '07 Proceedings of the Third IASTED International Conference on Computational Intelligence, pp. 163–167, 2007.

[12] G. M. Stefan and M. Malita, Can *one-chip parallel computing* be liberated from *ad hoc solutions?* a *computation model based approach* and its *implementation*, 18th International Conference on Ciruits, Systems, Communications and Computers (CSCC 2014), Santorini Island, Greece, pp. 582–597, 2007.

[13] S. Kleene, *General recursive functions of natural numbers*, Mathematische Annalen, vol. 112, no. 1, pp. 727–742, 1936.

[14] A. Krizhevsky and I. S. end Geoffrey E. Hinton, *Imagenet classification with deep convolutional neural networks*, Advances in Neural Information Processing Systems 25 (NIPS 2012), 2012.