Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

A highly parameterizable framework for Conditional Restricted Boltzmann Machine based workloads accelerated with FPGAs and OpenCL

Zoran Jakšić [a,*], Nicola Cadenelli [a,b], David Buchaca Prats [a,b], Jordà Polo [a], Josep Lluís Berral Garcia [a], David Carrera Perez [a,b]

[a] Barcelona Supercomputing Center (BSC), C. Jordi Girona 1-3, 08034, Barcelona, Spain
[b] Universitat Politècnica de Catalunya (UPC) - BarcelonaTECH, Spain

A R T I C L E I N F O

Article history:
Received 27 May 2019
Received in revised form 21 August 2019
Accepted 27 October 2019
Available online 1 November 2019

Keywords:
CRBM
FPGA
OpenCL
Time-series
ANN
GEMM

A B S T R A C T

Conditional Restricted Boltzmann Machine (CRBM) is a promising candidate for a multidimensional system modeling that can learn a probability distribution over a set of data. It is a specific type of an artificial neural network with one input (visible) and one output (hidden) layer. Recently published works demonstrate that CRBM is a suitable mechanism for modeling multidimensional time series such as human motion, workload characterization, city traffic analysis. The process of learning and inference of these systems relies on linear algebra functions like matrix–matrix multiplication, and for higher data sets, they are very compute-intensive.

In this paper, we present a configurable framework for CRBM based workloads for arbitrary large models. We show how to accelerate the learning process of CRBM with FPGAs and OpenCL, and we conduct an extensive scalability study for different model sizes and system configurations. We show significant improvement in performance/Watt for large models and batch sizes (from 1.51x up to 5.71x depending on the host configuration) when we use FPGA and OpenCL for the acceleration, and limited benefits for small models comparing to the state-of-the-art CPU solution.

1. Introduction

Conditional Restricted Boltzmann Machine (CRBM) [1–3] is a Machine Learning (ML) mechanism that attracts significant attention in recent years. It can learn a probability distribution over a set of data, and its application in system modeling and prediction is under research primarily in the area of unsupervised learning. They are a specific type of an artificial neural network (ANN) with two layers. Usually, authors name the input "visible layer," and the output "hidden layer". Recent publications demonstrate that CRBM is a right candidate for modeling of multidimensional time-series such as human motion [2], workload characterization [3], road traffic analysis/predictions, etc.

In essence, the learning process of CRBMs, as in other Deep Neural Networks (DNN), uses a gradient descent algorithm where weights are updated continuously in an iterative process. Implementation of this algorithm relies on linear algebra functions like vector–matrix and matrix–matrix multiplication. Previous research [3] proved that the learning process of a (C)RBM is much longer and dominates over inference time. Thus, the acceleration of learning is essential.

For large datasets and model dimensions, these applications are very compute-intensive. Because of that, nowadays many proposals try to leverage technologies such as General Purpose Graphical Processor Units (GPGPUs) or Field Programmable Gate Arrays (FPGAs). To improve the performance of these algorithms or to reduce the overall energy consumption. These accelerators provide a higher number of GFLOP/s when compared to CPU, and potentially any ML (DL) application that based on General Matrix Multiplication (GEMM) could benefit from using it.

A big issue, on the other hand, for the usage of the heterogeneous architectures in the workload acceleration, is the communication between the different parts of the system. More precisely, the data transfer between the Host application (that typically runs on a general-purpose CPU) and an accelerator has to be taken into account as well. Otherwise, the benefit of code acceleration would be significantly lower due to non-optimized communication between a host and an accelerator, i.e., an accelerator or a host can be idle a significant percentage of time due to inadequate data transfer mechanism. Because of that, to

* Corresponding author.
E-mail addresses: zoran.jaksic@bsc.es (Z. Jakšić), nicola.cadenelli@bsc.es (N. Cadenelli), david.buchaca@bsc.es (D.B. Prats), jorda.polo@bsc.es (J. Polo), josep.berral@bsc.es (J.L. Berral Garcia), david.carrera@bsc.es (D.C. Perez).

https://doi.org/10.1016/j.future.2019.10.025

exploit the advantage of an accelerator over a CPU-only state-of-the-art solution, a holistic approach in analyzing the application is necessary.

Although FPGAs have existed for decades, their application in the areas of High-Performance Computing (HPC) and Deep Learning (DL) is relatively new. Traditionally, people have used some Hardware Description Languages (VHDL or Verilog) to implement FPGA design. Programming FPGA with these languages allows fine-tuning of the logic, and in conjunction with placement and timing constraints, achieves optimal performance. However, the design and verification of those projects are very demanding, and they consume a significant amount of time. Because of that, some alternative approaches as High-Level Synthesis (HLS) emerged to offer a faster design time at the cost of some performance penalty.

OpenCL [4] is one approach for designing HPC and DL applications when we use FPGAs for their acceleration. In essence, it is a set of C libraries that provide two different things:

1. Implementation of FPGA kernels with C language. At first, HLS compiler takes the C code and transfer it to Verilog RTL design, and in the second step, it takes Verilog code and compiles it for specific FPGA.
2. Set of runtime functions that host program can use for interaction with an accelerator card (FPGA, GPU).

Although much effort has been invested in the development of OpenCL compilers to make FPGA technology more accessible to people with a little background on it, many improvements are still necessary. The compilation of OpenCL kernels for FPGA guarantees the correct execution of the code, but achieving a maximal performance is not a trivial task at the current stage of development. Also, orchestrating their execution in the manner that the whole system gets utilized as much as possible is a separate problem but equally important when designing an application.

Nowadays, the research on the OpenCL for FPGA development is mostly focused on the acceleration of (Convolutional) Deep Neural Networks (DNN), especially their inference part [5–7]. Although some interesting works that leverage these techniques for the acceleration of HPC applications appear from time to time (like in [8–10]); more effort should be put in this direction in order to understand the real potential of OpenCL for FPGA. Application characterizations are critical since they offer valuable insights to the Cloud/Data-Center/Supercomputer architects about the system requirements at the early stage before they get deployed. Knowing the real requirements of workloads is especially important when they should work in an environment with hard power or timing constraints (e.g., environment on the edge [11]).

In this work, we present a parametrizable framework for implementation of CRBM based applications accelerated with FPGA and OpenCL. For the acceleration part, we improve the performance of the learning part of the CRBM as we observed that it dominates over inference.

At first, we implement matrix multiplication kernels with OpenCL and compile it for Arria 10 FPGA board, and then we show how to optimize the host code of a CRBM based application to support FPGA accelerated version.

In summary, our contributions are:

1. A parametrizable framework for CRBM applications based on OpenCL for a heterogeneous environment Xeon CPU + Discrete FPGA. The framework supports arbitrary large models that are not limited by the size FPGA on-chip memory;

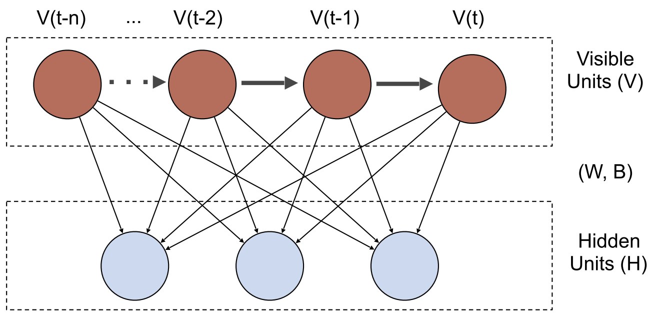

Fig. 1. Conditional Restricted Boltzmann Machine (CRBM) block diagram.

2. Implementation of GEMM on FPGA in two versions. In one version, we use Host Pipes, a new OpenCL mechanism that we use to reduce communication overhead between FPGA and CPU. As far as we know, this is the first paper that investigates Host Pipe benefits in a DL application based on GEMM;
3. Optimization of the CPU (Host) code to support usage of FPGA GEMM designs for CRBM acceleration;
4. An extensive scalability study of the application concerning performance and energy consumption, for different models and batch sizes and system configurations.

As far as we know, this is the first paper that describes an implementation of (Conditional) Restricted Boltzmann Machines using FPGA and OpenCL. The framework that we present here is highly scalable and easily portable to any newer version of the FPGA accelerator board that supports OpenCL. We perform in-depth scalability study in terms of batch size, number of Gibbs samples and model size, as well as the number of threads.

Moreover, some conclusions that we draw here overcome the scope of CRBM acceleration, and they are relevant for any application based on GEMM that uses discrete FPGA for acceleration, especially when there is data dependency between input and output.

For large models, we observe performance improvements from 1.51x to 5.51x when we compare results with the state-of-the-art CPU implementations compiled for a different number of threads. Reduction in energy consumption is even higher and goes from 1.61x to 5.71x for the same baselines and configurations. However, for small CRBM models and batch sizes, we observe a significant loss in the performance because the data transfer between the CPU and FPGA dominates execution time. Because of the data dependency, FPGA stays idle for a considerable portion of the time, which increases the overall execution time of the application.

The rest of the paper is organized as follows. In Section 2 we give a theoretical background of Conditional Restricted Boltzmann Machine. In Section 3 we present related work. In Section 4 we describe our system solution and in Section 5 we present performance and energy numbers. Finally, in Section 6 we draw conclusions.

2. Conditional restricted Boltzmann machine

Restricted Boltzmann Machine (RBM) is a type of artificial neural network that can learn a probability distribution over its set of inputs. It is a form of a bipartite graph where nodes of one part are input, usually referred to as a visible layer, and nodes of another part are output, commonly called hidden layer (Fig. 1).

The output of a hidden unit is defined according to Eq. (1), where: σ is the activation function (usually sigmoid Eq. (2)), $w_{i,j}$

are weights, and b_j is a threshold (bias). Hereinafter we refer to all $w_{i,j}$ values as matrix W and to all b_j values as vector B.

$$P(h_j = 1 \mid v) = \sigma(b_j + \sum_{i=1}^{m} w_{i,j} v_i) \quad (1)$$

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad (2)$$

W and B are obtained during the learning phase in the iterative process that uses the gradient descent technique presented in Eqs. (3) and (4).

$$W(i+1) = W(i) + \alpha \frac{\partial W}{\partial t} \quad (3)$$

$$B(i+1) = B(i) + \alpha \frac{\partial B}{\partial t} \quad (4)$$

Lastly, the method calculates the gradient of W and B according to Eqs. (5) and (6). Here, matrices $\hat{v}^T_{(k)}$ and $\hat{h}_{(k)}$ are the values of the visible and hidden layer after k steps of Gibbs sampling.

$$\frac{\partial W}{\partial t} \approx h_{(1)} \cdot v^T_{(1)} - \hat{h}_{(k)} \cdot \hat{v}^T_{(k)} \quad (5)$$

$$\frac{\partial B}{\partial t} \approx h_{(1)} - \hat{h}_{(k)} \quad (6)$$

Conditional Restricted Boltzmann Machine is, in essence, Restricted Boltzmann Machine with some special connections that are used to model temporal dependencies. Fig. 1 presents CRBM for one dimensional input. Visible units $V(t-n) \dots V(t-1)$ are the history of the input and value $V(t)$ is the current state of the system. Some authors separate the history part from the last state and accordingly divide matrices W and B in submatrices [3].

The learning process of a CRBM can be generalized in the following five steps:

1. From the input time-series data, we generate data slices with the length of the units in the input layer.
2. Perform Gibbs sampling over a set (batch) of the input slices. The number of Gibbs samples can range from 2 to some predefined number N. This is the most time-consuming part of the process.
3. Calculate gradients according to Eq. (6) and update matrices W and B.
4. Repeat steps 2 and 3 for all the batches.
5. Repeat steps 2–4 for some predefined number of epochs, or until the difference between two consecutive updates of the matrices W and B becomes less than some predefined value.

As in any other DL applications, we define different algorithms based on their batch size:

- Batch gradient descent: learning algorithm that processes all slices at once. It calculates the gradient and updates matrices B and W in one step.
- Mini-batch gradient descent: learning algorithm that processes the slices in subsets, called *mini-batch*. It calculates the gradient and updates matrices B and W iteratively, once per each mini-batch until all slices are processed.
- Stochastic gradient descent: learning algorithm that processes only one slice at the time. It is the extreme case the Mini-batch gradient with mini-batch of one single slice.

3. Related work

Research on accelerating of HPC and DL workloads with FPGAs and GPUs is a very actual topic. A very good survey paper [12] summarizes the current state-of-the-art in the domain of DL and AI. Mentioning all these papers goes beyond the scope of this work, so here we cite just the most significant for us.

We split these works into two categories. In the first category, we put the most recent research works that propose an efficient solution for matrix multiplication on FPGA.

In [13], the authors propose some optimization for sparse matrices, but since there is not much data about the potential sparsity of CRBM models, we did not focus our work in that direction.

The authors of [14] claim the performance of almost 0.9 TFLOPs on the Arria 10 1150 FPGA, the same one that we used in this paper. However, it is not clear if they used OpenCL for the implementation, or they use some RTL language to optimize the design and achieve these performance numbers. Besides, they do not state the matrix dimensions for their benchmarks.

In [15] authors present a framework for matrix-multiply for Intel HARPv2 platform. They state a lower performance than [14], possibly because of smaller resources available on FPGA for this platform. However, differently from our work, they show results with smaller data types (16 and 8 bits), and they use the approach to accelerate the inference part of some convolutional neural networks. Also, in our work, we are more interested to see the benefit in the learning process of some DL mechanism rather than inference what was the case in [15].

Since the systolic architectures map well for FPGA, most of these works adopt that approach for matrix multiplication implementation.

In the second category, we put the papers whose primary objective is the implementation of (C)RBM with FPGA.

Most of the works that explore (C)RBM implementation on FPGAs evaluate the design for smaller models (with matrix W sizes up to 512×512) [16–22].

Although there is a paper whose primary focus is the acceleration of the inference part of CRBM [18], the majority try to speed up the learning process. In [19,20] authors prove that for the learning phase of CRBM, the precision of 16-bit is enough, and investigation in that direction is promising. However, since we also tested larger models in our paper, in this paper, we only evaluated 32-bit solutions. In [20] authors reached similar conclusions to ours: the FPGA has limited capability to speed up these models.

The most recent paper that we found on this topic is [22]. The architecture that the authors propose tries to minimize the effects of data dependency when they partially compute the gradient as data arrive. They state that the solution achieves a 4x improvement over the previous for the same FPGA and an 87x improvement over software solution. However, these numbers should be considered cautiously, because their baseline software solution is set very low (they compare results against one thread software implementation that use double-precision data against 32-bit integer precision implementation on FPGA). Also, their solution is limited to the models that can fit on the FPGA (maximal W size that they use 512×512 for 32-bit data and 512×1024 for 18-bit).

Unlike other papers, we focused our work on four major points:

- To show how to use CPU and FPGA in conjunction for the execution of CRBM based application without passing an extensive process of RTL implementation and evaluation of an FPGA design. Besides it takes less time to develop, this is very useful because the framework is easily portable to any newer FPGA Board that has support for OpenCL;
- To support arbitrary large models that are not limited by the size FPGA on-chip memory;

- To show how to tune CPU code to optimize the application execution, and also to show how the application scales for the different number of CPU threads;
- Finally, we propose the usage of a new OpenCL mechanism, *Host Pipe*, to mitigate the problem of CPU–FPGA communication as we identified it as a big challenge for application optimization. As far as we know, this is the first paper that demonstrates usage of *Host Pipe* for a GEMM based application and evaluates benefits of this implementation.

As far as we know, this is the first paper that describes an implementation of a parametrizable CRBM framework with some HLS tool (such as OpenCL). It presents an in-depth study of the performance of a CRBM based application that utilizes CPU + FPGA. We believe that this is a crucial task because it shows how the application scales for different model sizes and configurations. As such, it is not just useful to Machine Learning engineers/researches, but also data-center or HPC centers architects, too.

4. System implementation

The Gibbs sampling of CRBM, described in Section 2, is the most time-consuming part of CRBM learning process [3]. Thus, our initial idea is to offload this sampling to FPGA. To do this, we first divide it into two steps: A *Forward* step that calculates the values of the hidden layer and a *Backward* pass that computes the numbers of the visible layer. These calculations comprise two essential components: a General matrix–matrix Multiplication (GEMM) that we execute in both *Forward* and *Backward* pass, and the computation of the activation function in the *Forward* step [2,3].

In this section, we first present an implementation of GEMM multiplication on FPGA, and then we describe the whole CRBM application. We implement the entire design with OpenCL, i.e., we used OpenCL for two purposes: (1) to perform GEMM on FPGA and (2) as an interface between CPU and FPGA accelerator.

4.1. General Matrix Multiplication (GEMM) on FPGA

Since we implemented the learning part of a CRBM, data precision is fundamental. In this work, we use 32-bit floating-point presentation of the data. Although in [19,20] authors show that 16-bit floating-point data could be enough for CRBM learning, those works investigate performance on smaller models. In this work, we want to evaluate the application for different model sizes. It is a known fact that for larger models, to achieve the desired accuracy, we need a higher precision of the data. Since there is not much data how many bits are enough for specific model size, we have taken a "skeptical" approach in these implementations and evaluations, and we use 32-bit floating-point data. Obviously, the usage of 16-bit floating-point data in this implementation would leave much more resources for GEMM on FPGA. Accordingly, we could achieve a higher number of operations per second in FPGA, which would (without any doubt) benefit the total execution time of the FPGA solution over the corresponding CPU version.

Fig. 2 presents a block scheme of the solution. We implement a systolic algorithm for GEMM. We use the standard technique of dividing matrices into the blocks (tiles) to improve utilization of adjacent data, which maximizes the performance. In the text that comes, we use the terminology according to Fig. 3.

The main building block of the systolic GEMM algorithm is a kernel Processing Element (PE) that performs multiplication and accumulation. It receives a block of input A and a block of input B. The dimensions of a block are 32 × 32. Each *PE* has a local memory that holds 32 × 32 = 1024 intermediate results. It performs 32 multiplications and additions in parallel and final summation with the previous result. When each *PE* finishes computation over one pair of blocks, it receives a new block from the row of blocks of input A and a new block from the column of blocks of the input B.

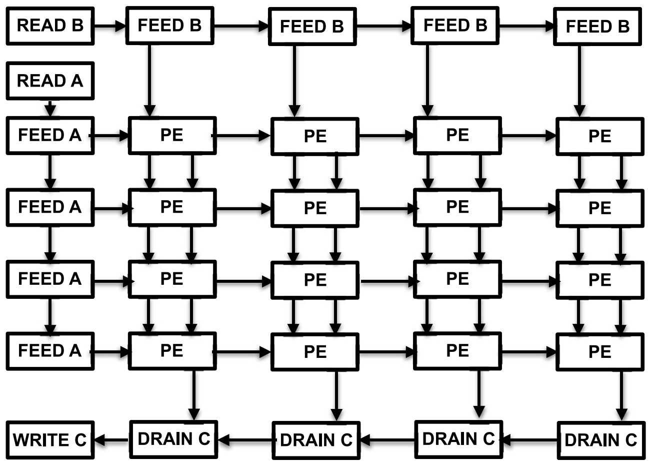

Fig. 2. Systolic matrix multiplication.

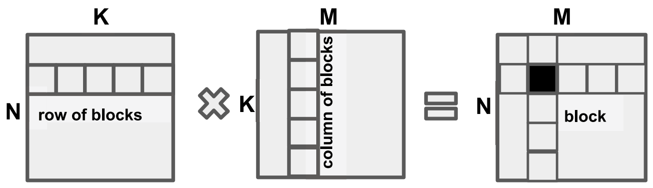

Fig. 3. Blocking of matrices.

A *PE* also sends processed input data to the adjacent *PE*s (to the right *PE* which receives a block from the input A, and to the bottom *PE* which receives the data block of the input B). The border *PE*s receive data from *FEEDA* and *FEEDB* kernels, and the first *FEEDA* and *FEEDB* kernels collect data from the *LOADA* and *LOADB* modules. All these kernels exchange data over separate channels, an OpenCL mechanism which is, in essence, a FIFO buffer. To increase the performance of *PE*s, we implement double buffering (ping-pong) at the input and the output, so multiplication and accumulation happen in parallel with data transfer.

All the blocks that we process on every *PE*s form a superblock which dimension depends on the number of *PE*s. In our configuration, all *PE*s form a 5 × 8 matrix. We implement the design on the Arria 10 1150 FPGA, which provides enough resources for this structure size. Since we use 32-bit floating-point data, the limitation comes from the number of the multipliers that Arria 10 FPGA has. For more straightforward implementation, we fix the inner dimension of the superblock to 256 elements. So when the data pipeline is full, the system receives and process a superblock of the input matrix A which sizes are 160 × 256 and a superblock of the input matrix B which sizes are 256 × 256.

When the multiplication of one row of superblocks of matrix A and one column of superblocks of matrix B finishes, the results are transferred to *DRAINC* kernels. Finally, *WRITEC* kernel receives the data from the last *DRAINC* kernel, and it writes the output superblock to the memory. Its dimensions are 160 × 256.

Lastly, we implement the kernel *WRITEC* in two flavors:

- The first one uses a classical approach. We accept data from *DRAINC* kernel and write data to the card DDR memory.
- The second one uses host pipes to transfer data directly to CPU as they arrive.

Table 1
FPGA Resource utilization.

Resource	Utilization
Logic utilization (ALMs)	231,386/427,200 (54%)
Block memory bits	13,572,648/55,562,240 (24%)
DSP Blocks	1280/1518 (84%)
Registers	417 895

Host pipe is a relatively new thing that Intel introduced in their SDK from version 17.1. Board Support Package of the Intel A10 Dev Kit that we used in the implementation has support for host pipe. Instead to write the whole data to the board local memory and after the full processing finishes, transfer all the data traditionally with "clEnqueueReadBuffer" function, *WRITEC* sends the result directly to the CPU as the data arrives via host pipe. As far as we know, this is the first application of the host pipe mechanism in the FPGA acceleration of some GEMM based workload.

Since we want to achieve a maximal number of GFLOPs of GEMM on FPGA, to relax the compiler, and reduce the on-chip memory needed for an efficient implementation, we perform blocking part on CPU, i.e., CPU takes the input data and transform the matrices so that FPGA can process the data as they arrive sequentially.

Table 1 presents resource utilization for the FPGA that we use in our implementation. The critical limiting resource is the number of DSP Blocks (84% used). We tried to compile FPGA design with the configuration for a higher number of *PE*s (we thought that compiler might use ALMs for the implementation of multipliers and adders). However, the compilation crashes at some point. Accordingly, we accepted this configuration as optimal.

4.2. CRBM learning algorithm

In the *Forward* step of Gibbs sampling, we make a batch of slices already formed from the input data by placing them in the rows of a matrix. We perform multiplication of the Batch matrix (Ba) with W, and we calculate the activation function (sigmoid) to obtain values of the hidden layer (H). In the *Backward* step, we perform multiplication of the obtained H with W^T, and we get the numbers of the visible layer (V). This process is one Gibbs sample. The process repeats until an equilibrium is reached, i.e., the difference between two consecutive values of the visible layer is below some predefined value. Alternatively, the process happens N times, where we define N at the beginning.

Implementing sigmoid in FPGA would consume its hardware resources. Since sigmoid is calculated only once at the output of the *Forward* step, we execute this function on the CPU.

Algorithm 1 shows application details. Although the algorithm is simple, some problems are hard to solve. The portion of execution time that goes on communication between CPU and FPGA card is the primary issue. Because of data dependency between the *Forward* and the *Backward* Gibbs sample, double buffering is not an option. In theory, we could solve this dependency by processing the row of blocks when it is ready ([22] presents an architecture that leverages this approach). However, that would mean transferring smaller chunks between FPGA and CPU. Moving smaller chunks of data over PCIe leads to the lower PCIe bandwidth utilization, and potential benefits of that implementation get mitigated due to the slower transfer of data, so in our system, this is not a very good solution.

In any case, some improvements are possible. Fig. 4 shows a block diagram of the Gibbs sampling process. The functions

```
initializeW, B;
for i ← 1 to epoch_num do
    for t ← 1 to batch_num do
        load(W);
        transform_to_blocks(W);
        copy(W, cpu2fpga);
        transpose(W);
        copy($W^T$, cpu2fpga);
        for k ← 1 to gibbs_num do
            load(Ba[t]);
            transform_to_blocks(Ba[t]);
            copy(Ba[t], cpu2fpga);
            $H = Ba[t] * W$ (on fpga);
            copy(H, fpga2cpu);
            $H = sigmoid(H + B)$ (on cpu);
            copy(H, cpu2fpga);
            $V = W^T * H$ (on fpga);
            copy(H, fpga2cpu);
            if $k == 1$ then
                calc_grad_first_part(W) (on cpu);
                calc_grad_first_part(B) (on cpu);
            end
        end
        compute $\frac{\partial W}{\partial t}$ (on cpu);
        compute $\frac{\partial B}{\partial t}$ (on cpu);
        update(W) (on cpu);
        update(B) (on cpu);
    end
end
```

Algorithm 1: CRBM Learning Algorithm

that we execute on the CPU (blocking, sigmoid, transposition) are parallelized for a certain number of threads using OpenMPv3.0 [23] programming model. Besides, we use a separate thread for copying data between CPU and the FPGA board and to start GEMM execution on FPGA to run in parallel when there is no data dependency.

The function *copy*(W, *cpu2fpga*) executes when W is ready, and this is done in parallel with preparation of matrix $B[t]$. The first *Forward* GEMM on the FPGA executes in parallel with transposing of W. Calculation of sigmoid on CPU is done in parallel with *copy*(W^T, *cpu2fpga*). Similarly, the second step of Gibbs sampling starts while we compute the first part of the gradients of W ($H \cdot V^T$) and B.

Although there are no data dependencies between functions that execute on CPU (i.e., blocking of W and $B[t]$ and transposing W), we have already parallelized these function for the number of threads; thus running them in parallel on the same resources would lead in performance loss. So, the basic rule is that we use CPU for matrix processing (plus some simple (fast) function like, start FPGA execution or initialize communication over PCIe).

To speed up the process of communication between FPGA to CPU, we use a host pipe. Host pipe allows data transfer from FPGA to CPU as results arrive. In theory, this reduces the execution time of the application because no separate data transfer of the output is necessary, i.e., the FPGA execution runs in parallel with output data transfer. However, reading the data on the CPU side at slower rate could backpressure FPGA execution. Overall benefits of this approach we evaluate in Section 5.

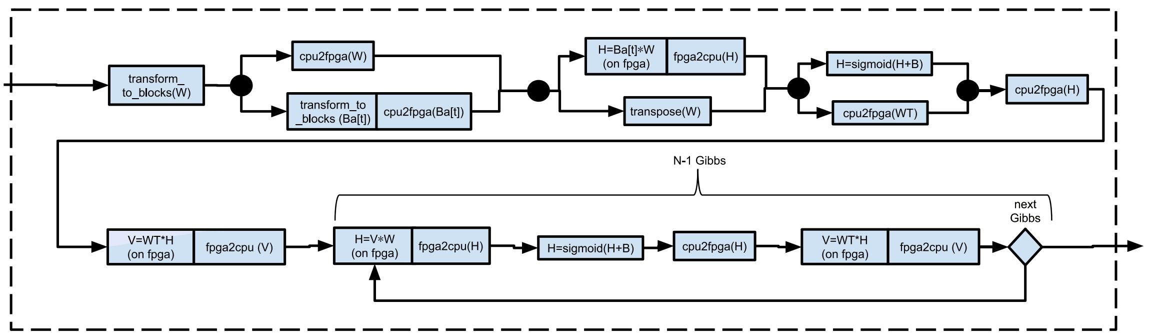

Fig. 4. Gibbs sampling application execution block scheme.

4.3. Considerations on using this approach with other type of accelerators

OpenCL is a programming model that we use for writing programs for heterogeneous platforms that consist CPU (usually called a Host) and accelerator (which can be GPU, DSP or FPGA). Although the goal of this paper is not to compare different technologies and to discuss which one is better for our CRBM application, we present some essential notes about generality and applicability of our solution to the other architectures.

It has been already shown that writing efficient OpenCL code for a device that would map well both for FPGA or GPU is hard. For instance, authors in [8] discuss usage of FPGA and GPU for an acceleration of data-intensive memory-bounded genomic application. They achieve similar results in terms of execution time and energy consumption, but more importantly, they showed that in order to efficiently use FPGAs, the GPU OpenCL code needs to be rewritten entirely.

When we consider our device code, in general, we should mention a couple of things. Although matrix multiplication can be implemented on GPU in a systolic manner, it is a well-known fact that this architecture maps better on FPGA due to their nature [24]. Our implementation heavily uses channel (pipe) primitives, which is a feature that maps very well on FPGA devices. For the implementation of the code we used some specific functions for Intel FPGAs (e.g., *write_channel_intel*, *read_channel_intel*), so just compiling the code for GPU (or even Xilinx FPGA) will not work. Also, we have chosen the parameters that are optimal for our FPGA device (Arria 10 1150). To efficiently port our framework to another architecture, a designer would have to choose these parameters according to the available resources offered by the target architecture.

The host code is pretty much general. Besides the part that is responsible for starting and monitoring the kernel execution, the rest of the code (copying data to the device, updating weights, etc.) should work without any change.

At the moment of writing this paper, we were not aware of any GPU or DSP device that offers the feature of Host Pipes. Of course, there is a possibility that some will appear in the future. If that happens, our Host code version that uses Host Pipe feature would demand minimal change (if any) to use GPU accelerator.

5. System evaluation

For the evaluation of the proposal, we used a Workstation with one discrete FPGA – the Arria 10 Development kit. The most critical parameters of the system we present in Table 2. We compiled the application and accelerator design, with Intel OpenCL compiler for FPGA version 18.0.

We evaluate the training of the models of different sizes. According to the size of the model, we named them from the smallest to the largest: "tiny, small, medium, large and huge". Table 3 shows the configuration of every model.

Table 2
System configuration.

CPU	Intel Xeon E5-2609 v3, 6 Cores, AVX-256 b Fmax 1.90 GHz
DRAM	64GB DDR4 2133 MHz
FPGA board	Arria 10 (1150) Dev Kit 2 GB DDR PCIev3 x8 Intel SDK for FPGA OpenCL v.18.0
OS	CentOS 7.4 kernel v3.10

Table 3
CRBM model configurations.

Model name	Visible units	Hidden units
Tiny	1024	512
Small	2048	1024
Medium	4096	1024
Big	8192	2048
Huge	8192	4096

For the evaluation of the application performance of every model, we generated a synthetic dataset. After that, we created a set of 25 600 vectors that we call slices. A slice is a subset of input data with a predefined length. The length of a slice is equal to the number of visible units of the specific model, e.g., the length of a slice for the "small" model is 2048. We split the whole set of slices into batches. To perform one epoch of training, we have to process $Batch\_Num = 25\,600/Batch\_Size$.

5.1. GEMM performance

On Fig. 5, we show the performance of GEMM on FPGA. We show the results for different configurations and different batch size. We see that performance of the Matrix Multiplication goes from very poor 130 GFLOPS for "tiny" model and batch size 160 up to 650 GFLOPs for "huge" model and 2560 batch size. These results are reasonable because the pipeline of the systolic structure has to be warmed up at the beginning. In other words, for "tiny" model the last PEs of the matrix multiplier does not start working before we load the whole matrices. However, as the size increases, the effects of pipeline warming up disappears, and performance improves. It reaches saturation at 650 GFLOPs for "large" inputs.

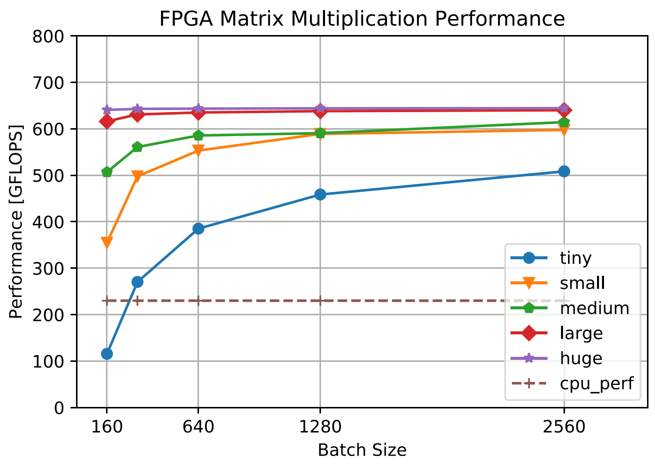

Fig. 5. FPGA matrix multiplication performance.

Dashed lines show the performance of GEMM when executed on a 6-core CPU that we have in our workstation when we use highly optimized state-of-the-art Intel Library for Linear Algebra (MKL). The number is almost constant, and it is about 220 GFLOPS, which is close to the theoretical maximum that this CPU can achieve. By limiting the execution for a smaller number of cores, performance drops proportionally. It is essential to show what are the maximal performance that can be achieved by CPU and FPGA for GEMM before anything since this comparison gives a critical insight what level of performance improvement we can expect for the application.

After the synthesis, the compilation report shows the kernel frequency of around 299 MHz (for both versions, the traditional one and the one with Host Pipes). The maximal clock frequency that Arria 10 FPGA can achieve is 450 MHz, according to Intel's documentation. However, this number is tough to reach, principally when we use some HLS tool as OpenCL. Maybe we could achieve a slightly higher frequency if we set different compilation seed or a different number of processing elements for GEMM structure. However, testing all the configurations for a various seed to find the solution that can improve performance couple of percents, we do not see very useful, especially if we know that the whole process of compilation for this design can last up to 10 h.

5.2. CRBM application results

In the following Figures, we present performance and energy results for different use cases of CRBM. For comparison reasons, we show the numbers for the solution when it is implemented on CPU when we use state-of-the-art Math Kernel Library (MKL) for computations. We believe that this code is the baseline for comparisons since these libraries are known as highly optimized for these kinds of algorithms. We present result for accelerated version when we use traditional approach (*FPGA_ACCEL*) and when we use host pipes for the application (*FPGA_ACCEL_HP*).

Before we started performance analysis, we verified the correctness of our results by comparing them with baseline CPU implementation. Although we use 32-bit floating-point data to implement GEMM on FPGA as well as we use it on CPU, due to "relaxed" implementation on FPGA (rounding to the floor, the order of adders, etc.) there is a possibility that error accumulates. We were comparing results between CPU version when we use state-of-the-art Math Kernel Library (MKL) for implementation (*CPU_MKL*) with accelerated versions; the classical one (*FPGA_ACCEL*) and the one that uses Host Pipes (*FPGA_ACCEL_HP*). We compared results after calculating data in "Forward" step, "Backward" step of Gibbs sampling process and after calculating the gradient, and the difference was never higher than 1% for every element of the corresponding matrices.

We measured execution time and total energy of the system for the different configurations, and model sizes. For energy measurement, we used "Wattsup" power meter [25] to collect real power data of the system. We used a simple script that performs sampling of the power consumption when the application is running. Sampling interval was 1 s.

Fig. 6 shows the Execution Time of the CPU version of the CRBM application and FPGA accelerated version for different models and batch size. It should be clear that the algorithm convergence, in general, depends on the batch size, and it might need more epoch to reach the final equilibrium when the batch size is smaller. In this analysis, we do not include this effect in our performance numbers because we are not interested in some specific use-case of CRBM. Instead, we want to show how the application behaves for different configurations (CPU version and FPGA accelerated one) for the same model size and system configuration.

As expected, the FPGA acceleration favors larger models and larger batch sizes. On Fig. 6 we can see that for the configuration for five threads we see performance degradation of 1.5x (for *FPGA_ACCEL* version) for "tiny" model and a batch size of 320, but these numbers reach *CPU_MKL* version as the batch size increases. Having in mind what performance we can achieve from FPGA for GEMM for these sizes, and if we take in the account the time for data transfer, matrix preparation, and sigmoid calculation, these results are logical.

FPGA_ACCEL_HP version performs better than the *FPGA_ACCEL* version for larger batches than for the smaller. The reason for this is the following. Because matrix sizes are low data transfer over PCIe gets slower, and host pipe actually cannot transfer data at the rate they arrive. In other words, it back pressures FPGA kernels that perform GEMM, which reduces its performance. As a result of that, we observe an increase in the execution time that even the absence of separate data transfer function from FPGA to CPU cannot compensate.

On the other hand, increasing the batch (or model) size, the data transfer gets faster. As a consequence, configurations *FPGA_ACCEL* and *FPGA_ACCEL_HP* perform better than the *CPU_MKL* version, and *FPGA_ACCEL_HP* performs better than *FPGA_ACCEL*. Benefits of using host pipes reduce as the model size becomes too big ("large" and "huge" models) because the overall execution time of data transfer becomes much smaller than the task of matrix multiplication.

Power measurements showed that *FPGA_ACCEL* solution consumes 15% less power on the average when compared to *CPU_MKL* when CPU is fully loaded (6 thread version). On the other hand, *FPGA_ACCEL_HP* version consumes the same amount of power as *CPU_MKL*. This result was a little unexpected for us, but after entering in the analysis of the solution, we concluded that the reason for this is probably permanent checking of the host pipe from the CPU side if there is data which consumes power.

Combining power and time numbers in the overall energy consumption, we obtained plots presented on Fig. 7.

We perform scalability analysis of the solution, and we present how the application behaves when it uses only a limited number of cores. This kind of analysis is useful if someone wants to use just a part of the CPU cores for CRBM learning and another piece for another application. So we configured the application to run for a different number of threads on CPU, and we present results on the following figures.

Fig. 8 shows the results when the batch size is 1280, but the numbers are similar for other batch sizes. When compared

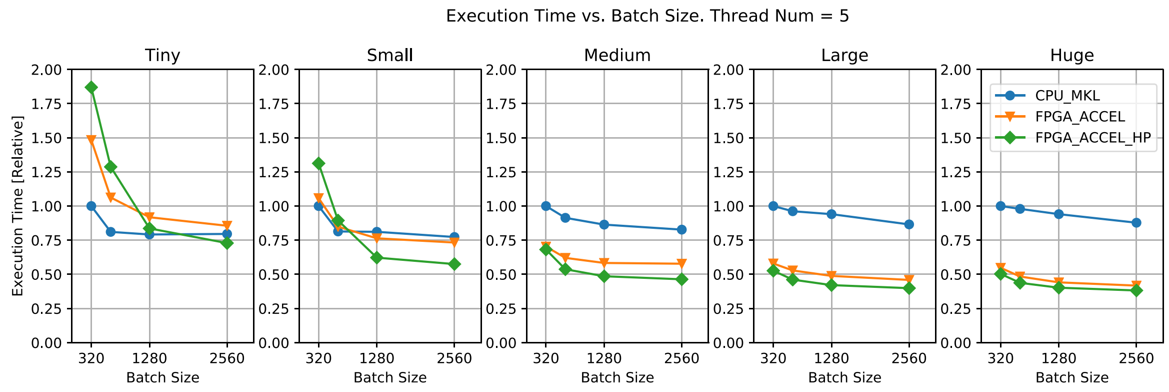

Fig. 6. Execution time vs. batch size for different models. Thread number = 5. Gibbs samples = 4. Epoch number = 10. Smaller is better.

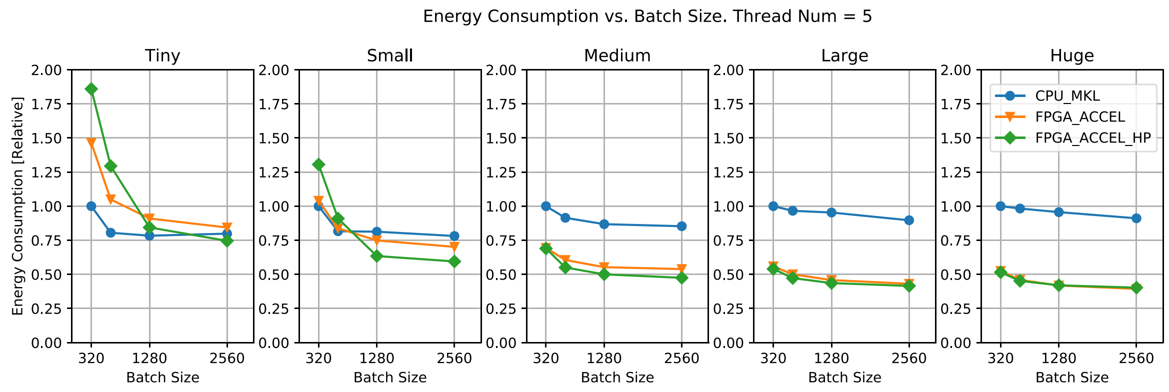

Fig. 7. Energy vs. batch size for different models. Thread number = 5. Gibbs samples = 4. Epoch number = 10. Smaller is batterer.

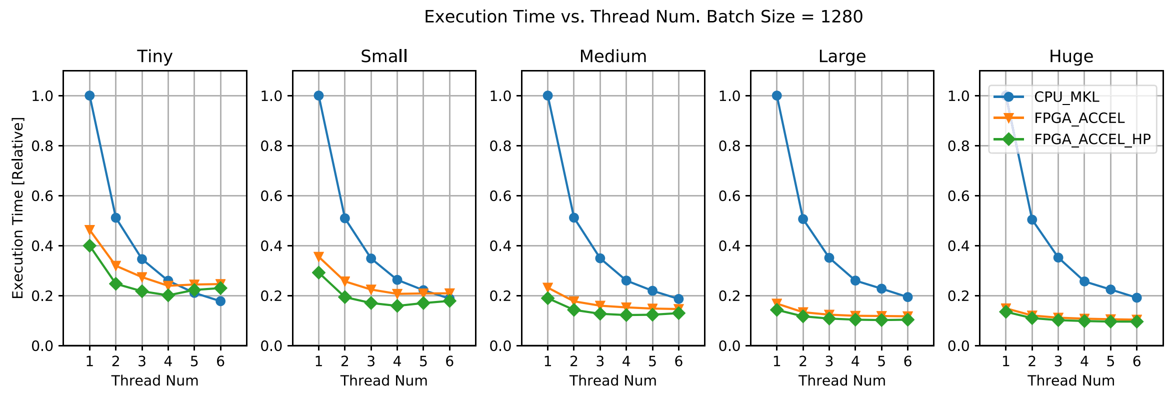

Fig. 8. Execution time vs. CPU thread number for different models. Batch size = 1280. Gibbs samples = 4. Epoch number = 10. Smaller is better.

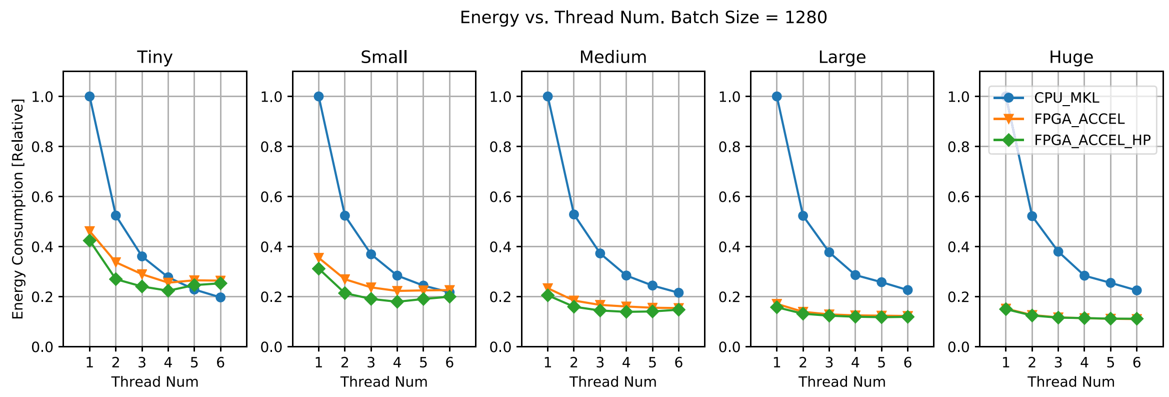

Fig. 9. Energy vs. CPU thread number for different models. Batch size = 1280. Gibbs samples = 4. Epoch number = 10. Smaller is better.

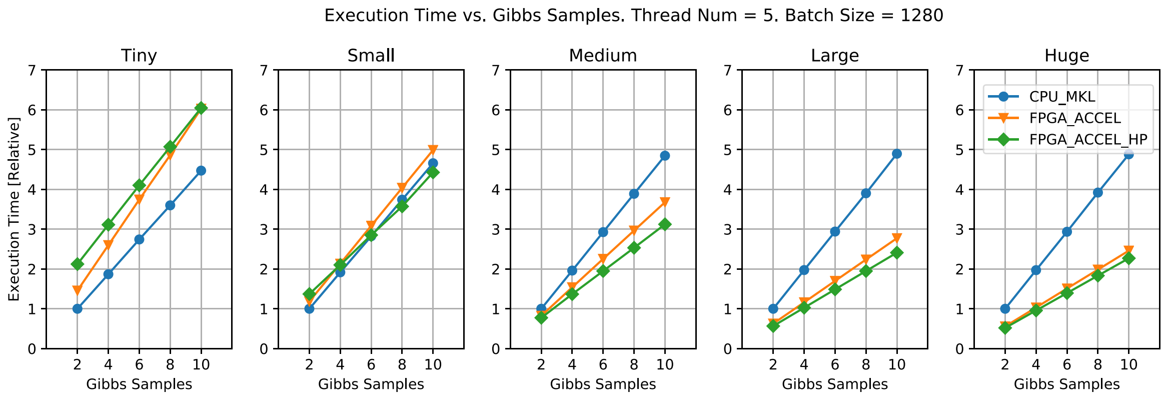

Fig. 10. Execution time vs. Gibbs samples. Batch size = 1280. Thread number = 5. Epoch number = 10. Smaller is better.

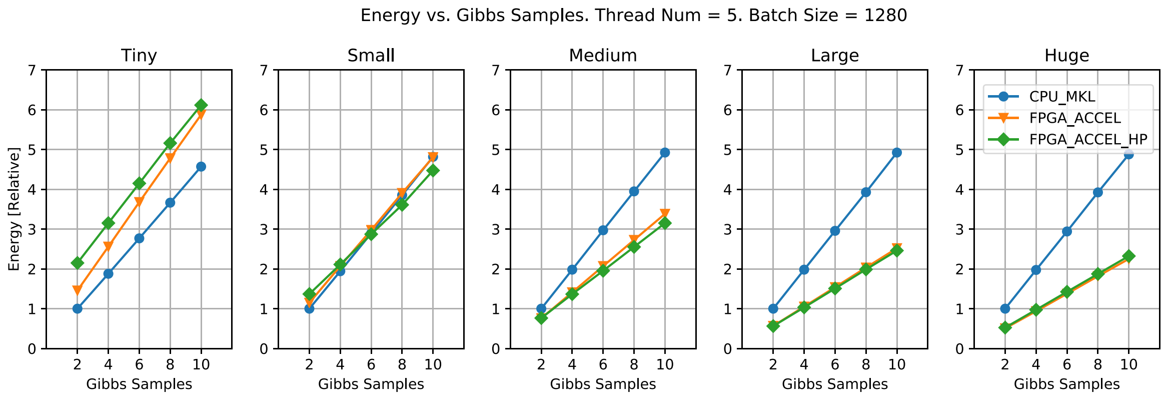

Fig. 11. Energy vs. Gibbs samples. Batch size = 1280. Thread number = 5. Epoch number = 10. Smaller is better.

with one-thread application improvements in FPGA goes up to 5.5x for "huge" model. Even for the "tiny" model for one thread configuration, the FPGA version shows better results than CPU only version, but for "tiny" model those improvements disappear very fast as the number of thread increase to 3.

Power measurements show that CPU-version, consumes the same power as *FPGA_ACCEL* version when it is configured up to 3 threads, and configuration with a higher number of threads benefit *FPGA_ACCEL* version which consumes up to 15% less than CPU for six thread version. Again *FPGA_ACCEL_HP* version consumes the most power. When we translate these numbers into energy results, we obtain significant energy savings according to Fig. 9.

Accelerated versions show better performance metrics when we apply a higher number of Gibbs samples per batch in gradient calculation. The reason is that the model and its transposed version is prepared and transferred only once to the acceleration card. Also, batch preparation (blocking) is done only once at the beginning of the Gibbs sampling process. So for any additional step of Gibbs sampling, the percentage of data moving over overall execution time is smaller, and the pure computation power of FPGA is more expressed. The effect is more observable for larger models. Figs. 10 and 11 present performance and energy numbers for the different number of Gibbs samples, and they confirm the previously stated thesis. Again, *FPGA_ACCEL_HP* scales a bit better for "small", "medium" and "large" models regarding execution time but this goes on the cost of the energy.

6. Conclusion

In this paper, we presented a parametrizable framework for implementation CRBM based workloads accelerated with FPGA and OpenCL. We implemented kernels that perform GEMM on FPGA, and we showed how to optimize the host application that runs on CPU to speed up the process of Gibbs sampling, which is a dominant part of the learning process. The paper proposes the usage of host pipes to reduce data transfer overhead between CPU and FPGA when there is a data dependency (as in the case of CRBM learning process). As far as we know, this is the first paper that proposes and evaluate all this. We performed an in-depth scalability study of the application for different configurations (number of threads that runs on CPU, batch sizes, Gibbs samples, and model sizes).

The proposed solution has better performance than the state-of-the-art CPU-MKL implementation for large models. For instance, when compared to the one-thread state-of-the-art CPU-MKL implementation, we achieve 5.6x improvement in the execution time. This number reduces when we make comparisons with the CPU version with the higher number of threads (e.g., for six CPU threads version improvements reach 1.55x in the execution time and 1.61x in the energy).

Smaller models and batch sizes favor CPU-MKL version because of the two reasons. First, the total number of GFLOPS that FPGA achieves is lower when matrix dimensions are smaller. Second FPGA and CPU are stalled a significant percentage of the time because of CPU–FPGA communication and data dependency between steps of Gibbs sampling process.

Up to 15% in the reduction of the execution time, we achieved when we used host pipes for data transfer between FPGA and CPU. This technique reduces communication overhead. However, these improvements go on the cost of overall energy consumption. Also, increasing the number of Gibbs samples for gradient computation scales better with FPGA versions in terms of energy and execution time.

Lastly, using OpenCL for FPGA besides it shortens the design time, it opens the possibility of using the code that is applicable for different architectures. Although more efforts have to be put in order to make a fully portable solution for CRBM for different accelerators (GPU, DSP), this work is a step forward in that direction.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreements No 639595); the Ministry of Economy of Spain under contract TIN2015-65316-P and Generalitat de Catalunya, Spain under contract 2014SGR1051; the ICREA, Spain Academia program; the BSC-CNS Severo Ochoa program, Spain (SEV-2015-0493) and Intel Corporation, United States

References

[1] V. Mnih, H. Larochelle, G.E. Hinton, Conditional restricted boltzmann machines for structured output prediction, CoRR abs/1202.3748. arXiv:1202.3748. URL http://arxiv.org/abs/1202.3748.

[2] G. Taylor, G. Hinton, S. Roweis, Two distributed-state models for generating high-dimensional time series, J. Mach. Learn. Res. 98 (1) (2011) 1025–1068.

[3] D.B. Prats, J.L. Berral, D. Carrera, Automatic generation of workload profiles using unsupervised learning pipelines, IEEE Trans. Netw. Serv. Manag. 15 (1) (2018) 142–155, http://dx.doi.org/10.1109/TNSM.2017.2786047.

[4] https://www.khronos.org/opencl/, in: Khronos Organisation Webpage, 2018.

[5] K. Guo, S. Zeng, J. Yu, Y. Wang, H. Yang, A survey of FPGA based neural network accelerator, CoRR abs/1712.08934. arXiv:1712.08934. URL http://arxiv.org/abs/1712.08934.

[6] U. Aydonat, S. O'Connell, D. Capalija, A.C. Ling, G.R. Chiu, An opencl™deep learning accelerator on arria 10, in: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, in: FPGA '17, ACM, New York, NY, USA, 2017, pp. 55–64, http://dx.doi.org/10.1145/3020078.3021738, URL http://doi.acm.org/10.1145/3020078.3021738.

[7] J. Zhang, J. Li, Improving the performance of opencl-based fpga accelerator for convolutional neural network, in: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, in: FPGA '17, ACM, New York, NY, USA, 2017, pp. 25–34, http://dx.doi.org/10.1145/3020078.3021698, URL http://doi.acm.org/10.1145/3020078.3021698.

[8] N. Cadenelli, Z. Jaksic, J. Polo, D. Carrera, Considerations in using opencl on gpus and fpgas for throughput-oriented genomics workloads, Future Gener. Comput. Syst. 94 (2019) 148–159, http://dx.doi.org/10.1016/j.future.2018.11.028, URL http://www.sciencedirect.com/science/article/pii/S0167739X18314183.

[9] D. Castells-Rufas, J. Carrabina, Opencl-based FPGA accelerator for disparity map generation with stereoscopic event cameras, CoRR abs/1903.03509. arXiv:1903.03509. URL http://arxiv.org/abs/1903.03509.

[10] S. Sridharan, Evaluation of 'opencl for FPGA' for data acquisition and acceleration in high energy physic, J. Phys. Conf. Ser. 664 (9) (2015) 092023, http://dx.doi.org/10.1088/1742-6596/664/9/092023, URL https://doi.org/10.1088%2F1742-6596%2F664%2F9%2F092023.

[11] S. Biookaghazadeh, F. Ren, M. Zhao, Are fpgas suitable for edge computing?, CoRR abs/1804.06404. arXiv:1804.06404. URL http://arxiv.org/abs/1804.06404.

[12] T. Ben-Nun, T. Hoefler, Demystifying parallel and distributed deep learning: An in-depth concurrency analysis, CoRR abs/1802.09941. arXiv:1802.09941. URL http://arxiv.org/abs/1802.09941.

[13] J. Yinger, E. Nurvitadhi, D. Capalija, A. Ling, D. Marr, S. Krishnan, D. Moss, S. Subhaschandra, Customizable fpga opencl matrix multiply design template for deep neural networks, in: 2017 International Conference on Field Programmable Technology (ICFPT), 2017, pp. 259–262, http://dx.doi.org/10.1109/FPT.2017.8280155.

[14] A. Vishwanath, Enabling High-Performance Floating-Point Designs, Intel Whitepaper, 2018.

[15] D.J. Moss, S. Krishnan, E. Nurvitadhi, P. Ratuszniak, C. Johnson, J. Sim, A. Mishra, D. Marr, S. Subhaschandra, P.H. Leong, A customizable matrix multiplication framework for the intel harpv2 xeon+fpga platform: A deep learning case study, in: Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, in: FPGA '18, ACM, New York, NY, USA, 2018, pp. 107–116, http://dx.doi.org/10.1145/3174243.3174258, URL http://doi.acm.org/10.1145/3174243.3174258.

[16] D. Ly, P. Chow, A high-performance fpga architecture for restricted boltzmann machines, in: Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, in: FPGA '09, ACM, New York, NY, USA, 2009, pp. 73–82, http://dx.doi.org/10.1145/1508128.1508140, URL http://doi.acm.org/10.1145/1508128.1508140.

[17] B. Li, M.H. Najafi, D.J. Lilja, An fpga implementation of a restricted boltzmann machine classifier using stochastic bit streams, in: 2015 IEEE 26th International Conference on Application-Specific Systems, Architectures and Processors (ASAP), 2015, pp. 68–69, http://dx.doi.org/10.1109/ASAP.2015.7245709.

[18] K. Ueyoshi, T. Marukame, T. Asai, M. Motomura, A. Schmid, Fpga implementation of a scalable and highly parallel architecture for restricted boltzmann machines, Circuits Syst. 07 (2016) 2132–2141.

[19] S.K. Kim, L.C. McAfee, P.L. McMahon, K. Olukotun, A highly scalable restricted boltzmann machine fpga implementation, in: 2009 International Conference on Field Programmable Logic and Applications, 2009, pp. 367–372, http://dx.doi.org/10.1109/FPL.2009.5272262.

[20] C. Lo, An fpga implementation of large restricted boltzmann machines, in: Master Thesis, 2010.

[21] T. Xia, Fpga implementation of a restricted boltzmann machine for handwriting recognition, in: Master Thesis, 2015.

[22] L. Kim, Deepx: Deep learning accelerator for restricted boltzmann machine artificial neural networks, IEEE Trans. Neural Netw. Learn. Syst. 29 (5) (2018) 1441–1453, http://dx.doi.org/10.1109/TNNLS.2017.2665555.

[23] https://www.openmp.org/, in: OpenMP Programming Model, 2018.

[24] J. Cong, Z. Fang, M. Lo, H. Wang, J. Xu, S. Zhang, Understanding performance differences of fpgas and gpus, in: 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2018, pp. 93–96, http://dx.doi.org/10.1109/FCCM.2018.00023.

[25] https://www.wattsupmeters.com, in: wattsup power meters, 2018.

Zoran Jakšić is a postdoctoral researcher in Barcelona Supercomputing Center (BSC). His primary research interest is the acceleration of compute-intensive workloads with FPGAs and GPUs. Before joining BSC, he was with Broadcom Networks where he worked as an RTL verification engineer for a year. He obtained a Ph.D. from Universitat Politecnica de Catalunya in 2015, and for that research, he was awarded by Intel E.U. Doctoral Student Honor Programme.

Nicola Cadenelli received the MS degree at the Università degli Studi di Brescia (UniBS), Italy in 2014. During his master studies, he spent one year as a visiting student at the University of Applied Sciences of Leipzig, Germany in 2012, and one semester at the Jülich Supercomputing Center, Germany in 2014.

Currently, he is a Ph.D. Student at the Technical University of Catalonia (UPC), Spain and part of the "DataCentric Computing" research group at the Barcelona Supercomputing Center (BSC), Spain.

In 2018, he was a summer visiting student at the Massachusetts Institute of Technology (MIT), USA.

His research revolve around the scalability, both vertical and horizontal, of real-world data-intensive workloads.

David Buchaca Prats received the degree in mathematics from University of Barcelona in 2012 and the M.Sc. degree in artificial intelligence from BarcelonaTech-UPC in 2014. He is currently pursuing the Ph.D. degree with the Data-Centric Computing, Barcelona Supercomputing Center. He is an applied mathematician, working in applications of artificial neural networks.

Dr. Jordà Polo received his bachelor's degree in Computer Science from Universitat Politècnica de Catalunya in 2009.

He then started his graduate work with Professors David Carrera and Yolanda Becerra at the Barcelona Supercomputing Center (BSC), completing his Ph.D. in 2014.

His research focused on how to manage and model the performance of data-intensive workloads.

He is currently working as a Postdoc in the same institution, leading the research in software-defined infrastructures and data-centric architectures for genomics workloads.

Josep Lluís Berral received his degree in Informatics (2007), M.Sc in Computer Architecture (2008), and Ph.D. at BarcelonaTech-UPC (2013). He is a data scientist, working in applications of data mining and machine learning on data-center and cloud environments at the Barcelona Supercomputing Center (BSC) within the "Data-Centric Computing" research line. He has worked at the High Performance Computing group at the Computer Architecture Department-UPC, also at the Relational Algorithms, Complexity and Learning group at the Computer Science Department-UPC. He received in 2017 a Juan de la Ciervaresearch fellowship by the Spanish Ministry of Economy. He is an IEEE and ACM member.

David Carrera received the MS degree at the Technical University of Catalonia (UPC) in 2002 and his PhD from the same university in 2008. He is an associate professor at the Computer Architecture Department of the UPC. He is also the Head of the "DataCentric Computing" research group at the Barcelona Supercomputing Center (BSC). His research interests are focused on the performance management of data center workloads.

In 2015 he was awarded an ERC Starting Grant for the project HiEST (1.5M€, 2015–2020), and ICREA Academia award (2015–2020) and an ERC Proof of Concept grant ('Hi-OMICS') in 2017 to explore the commercialization of an SDI orchestrator for genomics workloads. He has participated in several EU-funded projects and has led the team at BSC that has developed the Aloja project (aloja.bsc.es) and the servIoTicy platform (servioticy.com). He is the PI for several industrial projects and collaborations with IBM, Microsoft and Cisco among others.

He was a summer intern at IBM Watson (Hawthorne, NY) in 2006, and a Visiting Research Scholar at IBM Watson (Yorktown, NY) in 2012. He received an IBM Faculty Award in 2010. He is an IEEE and ACM member.