The BondMachine toolkit: Enabling Machine Learning on FPGA

Mirko Mariotti[*] [ab], **Loriano Storchi** [bc], **Daniele Spiga** [b], **Davide Salomoni**[e], **Tommaso Boccali**[f], **Daniele Bonacorsi**[d]
[a]*Dipartimento di Fisica e Geologia, Universitá degli Studi di Perugia, Via Pascoli, 06123 Perugia, Italy*
[b]*INFN sezione di Perugia, Via Pascoli, 06123 Perugia, Italy*
[c]*Dipartimento di Farmacia, Universitá degli Studi G. D'Annunzio, Chieti, Italy*
[d]*Department of Physics and Astronomy, University of Bologna, Viale Berti Pichat 6/2, 40127 Bologna, Italy*
[e]*INFN CNAF , Viale Berti Pichat 6/2, 40127 Bologna, Italy*
[f]*INFN sezione di Pisa, largo Pontecorvo 3, 56127 Pisa, Italy*
E-mail: mirko.mariotti@unipg.it, loriano@storchi.org, daniele.spiga@pg.infn.it

The BondMachine (BM) is an innovative prototype software ecosystem aimed at creating facilities where both hardware and software are co-designed, guaranteeing a full exploitation of fabric capabilities (both in terms of concurrency and heterogeneity) with the smallest possible power dissipation. In the present paper we will provide a technical overview of the key aspects of the BondMachine toolkit, highlighting the advancements brought about by the porting of Go code in hardware. We will then show a cloud-based BM as a Service deployment. Finally, we will focus on TensorFlow, and in this context we will show how we plan to benchmark the system with a ML tracking reconstruction from pp collision at the LHC.

International Symposium on Grids and Clouds (ISGC) 2019 and Soundscape conference
31 March - 5 April 2019
Academia Sinica, Taipei, Taiwan

[*]Speaker.

1. Introduction

Future systems will be characterized by the presence of many computing cores in a single device, by heterogeneous architectures built to optimize power and "silicon" consumption as much as possible and by re-configurable hardware technologies. These concepts have been demonstrated, both in software programming and hardware evolution, by the multi-core [1], GPGPU [2], OpenCL [3] and re-programmable logic devices [4] which populate the spectrum from small devices up to large-scale data centers. A key to the success in the era of hybrid computing will be how coherently HW/SW systems will take all these components into account.

The BondMachine (BM) is an innovative prototype software ecosystem aimed at creating facilities where both hardware and software are co-designed, guaranteeing a full exploitation of fabric capabilities (both in terms of concurrency and heterogeneity) with the smallest possible power dissipation. The disruptive innovation of the BM is to provide a new kind of computer architecture, where hardware dynamically adapts to the specific computational problem, rather than being static and generic, as in standard CPUs synthesized in silicon.

In order to exploit the dynamic nature of the BM, its main goal is to create the described heterogeneous and flexible architectures on top of re-configurable technology devices (such as FPGAs) [4]. Moreover, the overall BM vision is based on the reduction of the number of hardware/software layers, which as a byproduct guarantees a simpler software development. This is precisely why the BM project has been thought as a complete re-configurable computing ecosystem, that starting from a high-level description of a computational task creates both the hardware and the software representing the optimal solution.

The BM uses Go [5] as main language for the codesign. Its concurrency primitives can be perfectly mapped to the BM architecture and they allow to write concurrent applications on FPGA with a small overhead compared to the HDL code. The flexibility of the BM makes possible the implementation of any computing system, ranging from networks of small agents, like IoT (Internet of Things) [6], to high performance devices for ML (Machine Learning) [7] or real time data analysis, and even systems that mix all these different characteristics together.

In the context of the HEP domain [8], we are developing new BM components to deploy complex AI systems on hardware, providing a high-level mechanism to translate into silicon Deep Learning networks, created via standard Tensorflow [9] and Keras toolkits [10]. Regarding the deployment of models, the BM provides several solutions, such as a standalone FPGA, accelerators coupled to workstations, as well as a BM as a Service running on hybrid Clouds [11].

2. BondMachine architectural overview

The aim of the BondMachine (BM) project is to implement a computing system to enable a real and full exploitation of the underlying hardware. This is a key to the success in the era of hybrid computing [12]. In order to achieve this objective the BM has been designed to create a heterogeneous and flexible architecture on top of FPGAs [4]. Moreover the overall vision is based on a reduction of the number of hardware/software layers which guarantees a simpler software development. That is the BM project has been thought as a complete re-configurable computing ecosystem, that starting from a high-level description of the task creates both the hardware and the software to optimally solve the computational problem.

The two architectural pillars of the BM are computing elements (processors) and non-computing elements (for example memories, channels, barriers). The latter are meant to be shared among processors. Finally thanks to a custom network protocol many BondMachines can be interconnected together, therefore building heterogeneous multi-core systems or even cluster of multi-cores.

The flexibility of the BM makes possible the implementation of any computing system ranging from networks of small agents, like IoT (Internet of Things) [6], to high performance devices for ML (Machine Learning) or real time data analysis, and even systems that mix all this different characteristics together.

The BM can interact with standard Linux workstations both as a special purpose hardware accelerator or as part of a computer/BM hybrid clusters. As a final and important remark we want to stress that, regardless of the scenario considered, the hardware/software generation always starts from a high level description of the problem.

2.1 Components

The present project is aimed to build a full re-configurable computing ecosystem made of several components. In the following we will report and describe all these components. The Architecture Description (AD), that is a full specification of the computing and non-computing shared objects which constitute the BondMachine (BM). The BM handling tools: a set of standard computer programs to manage any aspect of the BMs starting from their creation up to the HDL code production. The BM front end tools: a set of programs to use the BM, that is a set of API, a compiler and other fundamental tools such as a web interface, translators from Boolean expressions, mathematical expressions and so on.

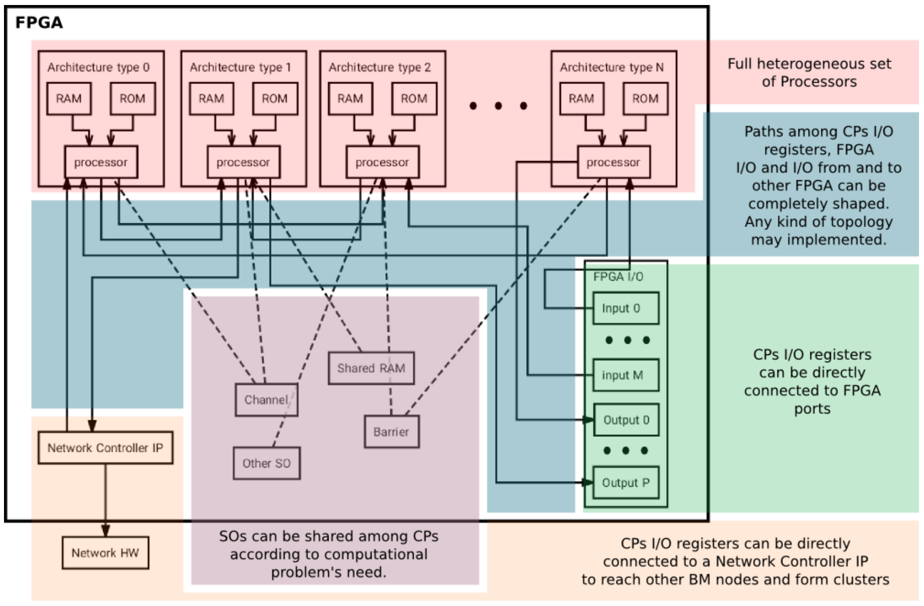

FPGA
Architecture type 0
Architecture type 1
Architecture type 2
Architecture type N
RAM
ROM
processor
Full heterogeneous set of Processors
Paths among CPs I/O registers, FPGA I/O and I/O from and to other FPGA can be completely shaped. Any kind of topology may implemented.
FPGA I/O
Input 0
input M
Output 0
Output P
CPs I/O registers can be directly connected to FPGA ports
Shared RAM
Channel
Barrier
Other SO
Network Controller IP
Network HW
SOs can be shared among CPs according to computational problem's need.
CPs I/O registers can be directly connected to a Network Controller IP to reach other BM nodes and form clusters

Figure 1: Template of the block diagram of a BondMachine design on an FPGA.

The image (Fig.1) shows a template of the block diagram of a BondMachine design on an FPGA. The BM architecture changes in order to satisfy the specific computational problem, so there is not a single FPGA design. Instead the tools that generate the architectures should be thought as firmware generators.

The BondMachine (BM) architecture consists of the full specification of the interconnections among Connecting Processors (CPs) and Shared Objects (SOs), being non-computing units, that can be shared among the various CPs. The main features of the BM architecture are the possibility to fully configure: i) the number and type of the processor cores, ii) the number of inputs and outputs, iii) the topology of the interconnections between processors and iv) the number and type of the Shared Objects used by each processor.

In the following we will detail about the cited components of the architecture.

2.1.1 Connecting Processors

The Connecting Processor is the computing core of the BondMachine. One of the main capability of a Connecting Processor, as the name suggest, is to be configured in such a way that can be easily connected to other processors and to any Shared Objects. CPs are as simple as possible, specialized and optimized to perform a single task. In fact any CP can be created with a different number of registers, different number of I/O registers and different instruction set (i.e. opcodes) with respect to the other ones.

The implemented opcodes are chosen among a rich set of possibilities, not all the possible opcodes are implemented on every CP. The implemented opcodes consist of the classical ones such as: set register **rset**, clear register **clr**, conditional jump **j**, **je**, **jz**, increment/decrement register **add**, **dec**, **inc**, copy register **cpy**, read input register **i2r**, write output register **r2o**. In addition we added some dedicated opcodes for controlling the Shared Objects such as operation related to the Internal and Shared Memory (**r2m**, **m2r**, **s2r**, **r2s**) as well as to Channel and Barrier management (**wrd**, **wwr**, **chc**, **chw**).

Registers within a CP are all general purpose and are named **r0 ... rR**, where R is not a constant but it can change between the various CPs. In addition, a CP has two types of specialized registers: the input registers, named **i0 ... iN**, that can only be read by the CP, while the output registers, **o0 ... oM** , can only be written. The registers are used to connect different CPs and an input register of a CP can only be connected to the output register of another CP. Moreover, an input register can also be used as BM input, and similarly an output register may be used as the BM output.

Finally it is important to report as every CP has two kind of memories. The ROM, that may contain only the instructions set of program to be executed by the processor, and the RAM, that is the local storage for the processor. Clearly the RAM may contain both the instructions set of the program as well as the application data.

2.1.2 Shared Objects

These are non-computing objects shared between all or some of the CPs. Several kind of objects can be implemented to increase the processing capability and functionality of the BMs improving the high-speed synchronization and communication between tasks running on separate CPs. Three kinds of objects have been currently implemented: Channels, Shared Memories and Barriers. Clearly other kind of components can be easily added in the future.

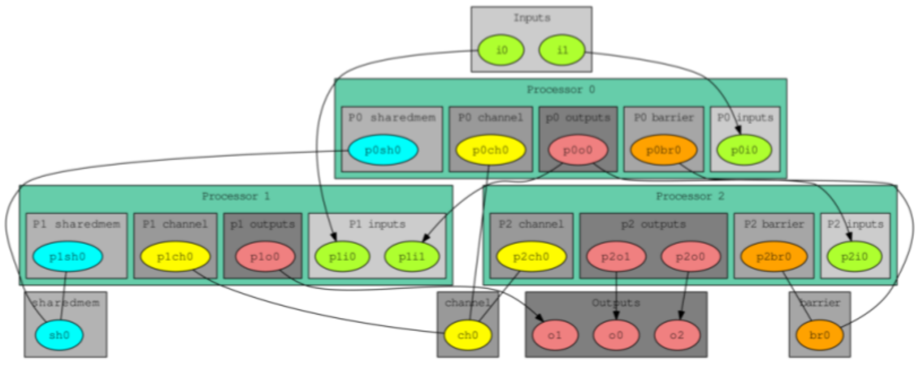


Figure 2: An example of layers in a BondMachine system

In the Figure 2 we are reporting a scheme of a complete example of the BondMachine architecture. Specifically it consists of two inputs and tree outputs interconnected between the input/output registers of the processors. One can easily see as the shared objects, such as memory, channels and barriers, are connected to the various processors.

Channels Channels are hardware message queues acting as conduits between two CPs. Following the concurrency model in Communicating Sequential Process (CSP) [13] each processor has a dedicated interface to the Channel and can send or receive data to and from it.

Shared Memory Shared Memory consists of RAM shared between one or more CPs. In a Shared Memory system, every processor has direct access with its dedicated interface (data input/output, address, write and enable signals), that is: it can directly load or store data to any memory address. The system is able to synchronize the read/write processors' requests without controlling the specified address and the conflicts. The depth of the RAM is calculated considering the implemented task of each CP and the value of allocated memory address. Clearly the depth of each Shared Memory object can be individually configured.

Barrier A Barrier is an object used to synchronize Concurrent Processors on a shared architecture. Upon reaching a barrier, a processor must wait until all the other processors reach the same barrier. The processors are stalled waiting for the others and during this time they are in a idle state. The implementation of the barrier mechanism uses a dedicated opcode named **hit** and a timeout counter to define a limit for the task completion.

2.1.3 Network Component

An interesting feature of the project is that several BondMachines can be interconnected via custom protocol, that is: a distributed cluster of heterogeneous multi-core machines can be built in such a way. To do so every BM joining a cluster has a network component within the FPGA that extends the same logic to other FPGAs within the cluster.

BMs may communicate using a native protocol called EtherBond. Its purpose is to replicate the electronic behavior of BMs registers and to extend it over the device boundaries. In other words clusters of BMs may be created and their behavior is driven by the same rules of separate BM devices. The main objective is then to handle devices and cluster in the same way. Interestingly the EtherBond protocol has been ported to Linux, so that BMs can now communicate also with a standard PC software.

3. Toolkit components

The complexity of the BondMachine architecture can be managed using a set of software tools. These tools allow to build a specific architecture as a function of the task one want to perform, to easily modify the architecture, to simulate the behavior and finally to check the functionality with the aim to generate the Hardware Description (HDL) code for a programmable device, i.e. a FPGA device.

The full set of tools can be subdivided in two different categories: i) the CP builder: that manages the configuration parameters of all the CPs (procbuilder), and ii) a BM builder: that manages the interconnection between the CPs and the SOs (bondmachine). Moreover, all the tools share the capability of using the generated BM architecture, so that the full architecture may be emulated directly on a workstation, using the so-called simbox framework, without the explicit need of a FPGA device.

3.1 The Bondgo compiler

Bondgo is the compiler, that starting from an high level language, in this case Go, produces the assembly code of the architecture. The generated assembly code may be finally assembled with the procbuilder tool, this will generate the binary code for a CP. Unlike other compilers, Bondgo may create the assembly code so that a given processor can directly run it, and also creates a specific CP optimized to run the code. That is, the Bondgo creates both the hardware and the software optimized for the hardware. The resulting architecture will be generated with the minimal needed resources and will be highly specialized.

Moreover Bondgo is able also to handle the concurrency of the source code by creating, if needed, a multi-cores BM, that is it can create a new CP in the BM every time a Go-routine is encountered (clearly each CP is optimized to run the code produced).

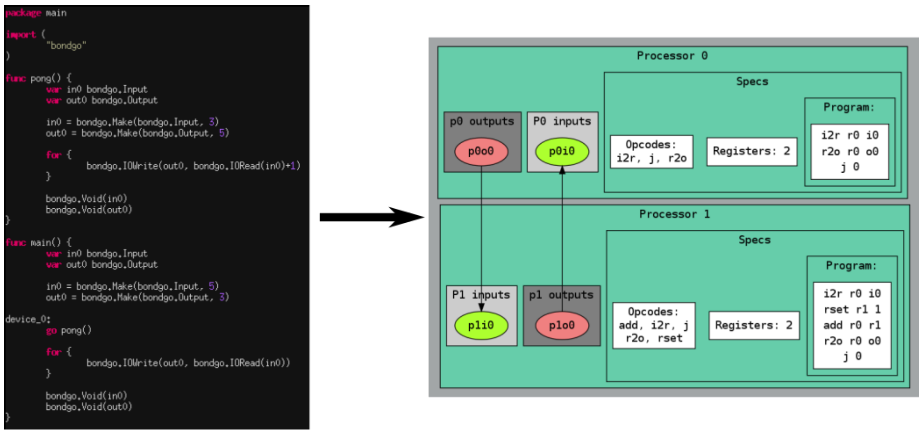
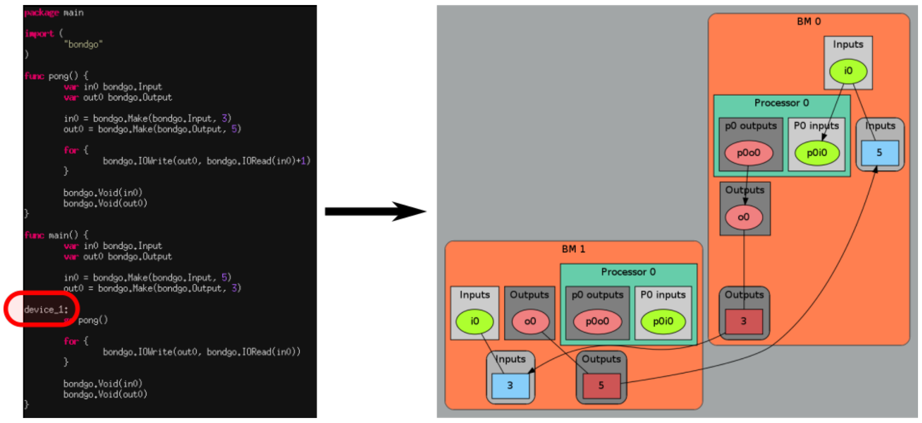

```
package main

import (
        "bondgo"
)

func pong() {
        var in0 bondgo.Input
        var out0 bondgo.Output

        in0 = bondgo.Make(bondgo.Input, 3)
        out0 = bondgo.Make(bondgo.Output, 5)

        for {
                bondgo.IOWrite(out0, bondgo.IORead(in0)+1)
        }

        bondgo.Void(in0)
        bondgo.Void(out0)
}

func main() {
        var in0 bondgo.Input
        var out0 bondgo.Output

        in0 = bondgo.Make(bondgo.Input, 5)
        out0 = bondgo.Make(bondgo.Output, 3)

device_0:
        go pong()

        for {
                bondgo.IOWrite(out0, bondgo.IORead(in0))
        }

        bondgo.Void(in0)
        bondgo.Void(out0)
}
```

Figure 3: A BM built starting from two Go-routines sending an uint8 data value back and forth through I/O registers created via bondgo.Make

In the following we report an example of the BondMachine. This is a trivial example yet it shows well the basic capabilities of the BondMachine architecture and ecosystem. Two Go-routines sending an **uint8** data value back and forth through I/O registers (created with bondgo.Make). The pong go-routine also increases the value by one before sending it back. Once the code has been compiled with Bondgo the result is a multi-core BondMachine as shown in the Figure 3.

Finally within Bondgo the developer can choose which routine (and thus which CP) runs on which FPGA, naturally allowing for the possibility of building autonomous clusters of multi-cores. Indeed starting from the previous example and just changing the device_0 label with device_1, the compiler is instructed to put the two go-routines on different BondMachines. Numerically the result will be the same but now we will use a cluster of two BondMachines connected via the EtherBond protocol. Interestingly, after a simple and quick change, a multi-core system is transmuted in a distributed system as shown in Figure 4:



```
package main

import (
        "bondgo"
)

func pong() {
        var in0 bondgo.Input
        var out0 bondgo.Output

        in0 = bondgo.Make(bondgo.Input, 3)
        out0 = bondgo.Make(bondgo.Output, 5)

        for {
                bondgo.IOWrite(out0, bondgo.IORead(in0)+1)
        }

        bondgo.Void(in0)
        bondgo.Void(out0)
}

func main() {
        var in0 bondgo.Input
        var out0 bondgo.Output

        in0 = bondgo.Make(bondgo.Input, 5)
        out0 = bondgo.Make(bondgo.Output, 3)

device_1:
        go pong()

        for {
                bondgo.IOWrite(out0, bondgo.IORead(in0))
        }

        bondgo.Void(in0)
        bondgo.Void(out0)
}
```

Figure 4: A BM built starting from two Go-routines sending an uint8 data value back and forth through I/O registers using a cluster of two BondMachines connected via the EtherBond protocol

4. Machine Learning with the BondMachine

The interconnected heterogeneous processors model, on which the BondMachine design is based, is clearly the perfect framework to run modern workloads like Machine Leaning (ML) and Computational Graph. To exploit this opportunity we developed several higher level tools. All of them share the capability to generate BondMachines starting from different sources, allowing the creation of trained graphs or neural networks in the form of HDL code. While the final product of all the cited tools is a BM architecture, the way each tool processes its own input to reach the final goal it is different and will be illustrated in the following,

4.1 Neuralbond

Neuralbond is an application that models a neural network [14] over BondMachines. The final result of the process in a device composed by many CPs acting as neuron-like computing units. The neuralbond approach it is API-based, so that its Go library can be used directly in the code to build a neural-like BondMachine architecture. An example of the library is reported in fig. 5.

```
layers := []int{2, 5, 2}
weights := make([]neuralbond.Weight, 0)

if *save_bondmachine != "" {
   if mymachine, ok :=
      neuralbond.Build_MLP(layers, weights); ok
      == nil {
      if _, err := os.Stat(*save_bondmachine);
      os.IsNotExist(err) {
         f, err := os.Create(*save_bondmachine)
         check(err)
         defer f.Close()
      }
   }
}
```

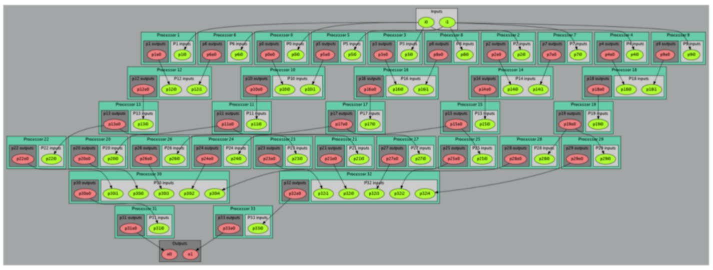


Figure 5: A Neuralbond usage example: The code on the left creates the multi-layer neural network shown on the right

4.2 Tensorflow Translator

TensorFlow [9] is a software library for data-flow from Google, it is used for machine learning applications such as neural networks. In the last few year it has become very popular and is a sort of standard de facto in ML. Computation in TensorFlow is performed by creating a computational graph that is a network of nodes, with each node known as an operation, running some function. The nodes of the graph are connected via tensors that are n-dimensional matrices. TensorFlow can save trained computational graphs in the form of file containing protocol buffers data. The developed translator may read these files and by descending the graph it creates one or more processor for every encountered node. Whenever a tensor is found, it is translated into a set of I/O registers among the respective node/processors.

4.3 NNEF composer

Neural Network Exchange Format (NNEF) [15] is a standard from the Khronos group that encapsulates a complete description of the structure, operations and parameters of a trained neural network. A NNEF description is independent from the training tools used to produce it and the inference engine used to execute it. Indeed, its main purpose is to enable the easy transfer of trained networks among frameworks, inference engines and devices.

The BondMachine NNEF Composer we developed is able to read a model file and to parse it to produce the related architecture accordingly, and finally to build a multi-core BondMachine. The adoption of NNEF has several advantages over the protocol buffer approach. Firstly every framework that is able to export a NNEF format file can be directly used to produce models workable into a BM. Equally obviously being a NNEF model a text file it is human readable, thus can be easily decoded and modified.

5. BondMachine computing accelerator

All the previously described ways of building BM architectures can be used equivalently on a single FPGA or on a clusters of FPGAs. In order to use a BM as a hardware accelerator attached to a Linux workstation we developed a software library to create the interface between the FPGA fabric and the Linux applications. Along with the developed C software library [16] a special-purpose hardware it is clearly needed. It can be either a chip that has a programmable logic (PL) subsystem alongside a standard processor (PS) [17], or a board with an FPGA that can be directly inserted in a computer bus (like for example a PCI express bus). In figure 6 an example of the first scenario is reported.

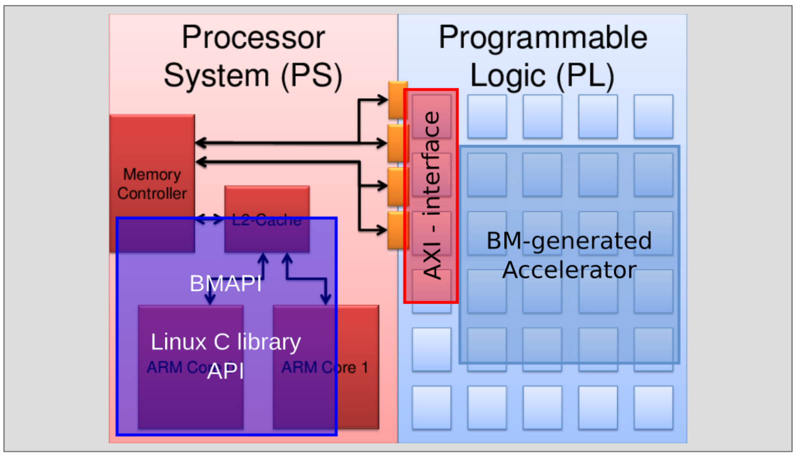


Figure 6: Hybrid chip accelerators: example of a Zynq chip with AXI4 protocol

Hence, as partially reported in figure 6, while a BM architecture is flashed on the PL part of the chip, on the processor (PS) part a standard application is linked against the BondMachine API C library developed. Clearly on the FPGA a firmware component we developed is responsible of managing the BM I/O registers dedicated to the accelerator interface. The interaction between PS and PL is performed within the chip itself and the BM API library can access memory-mapped components of the interface and send data to the accelerator back and forth.

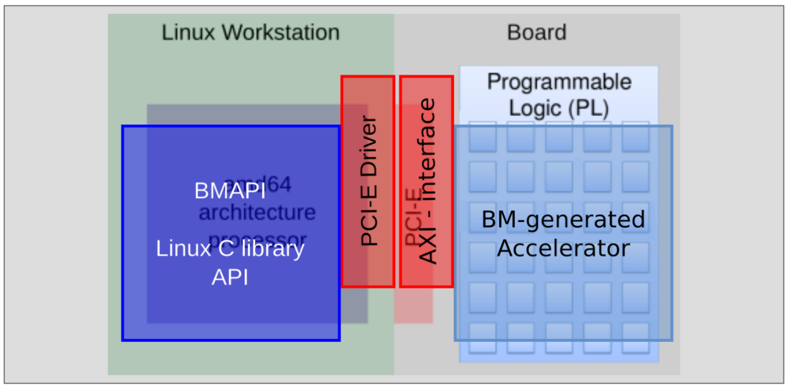


Figure 7: PCIe accelerator example

The second scenario we cited if the the FPGA on PCI express boards that is reported in figure 7. The scheme is similar to the previous one with an exception. Since there is no direct communication mechanism among the FPGA and the processor a PCI express driver is needed on the workstation kernel to interface it with the accelerator via the PCI bus.

5.1 Cloud accelerators

FPGA can also be used in a cloud context. It is either possible to buy FPGA computing resource from commercial cloud providers or to create a private cloud infrastructure [18] with FPGA enabled machines.

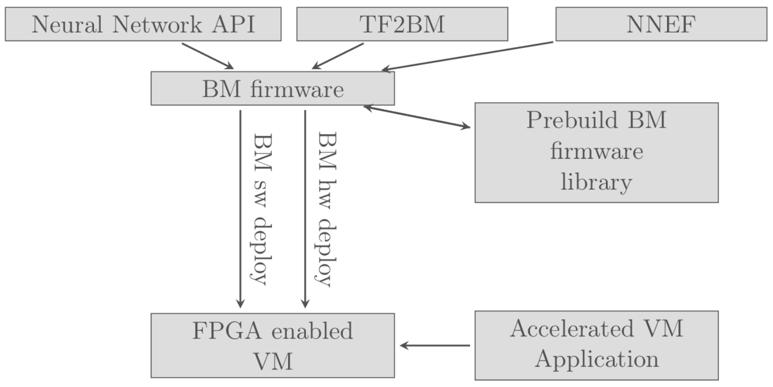


Figure 8: A BM Cloud accelerator workflow

In figure 8 we are reporting a workflow of a BM accelerated VM application starting from one the previously described methods to produce ML accelerators. Firstly a BM firmware is created using one of the described tools or it is taken from a library of pre-built accelerators. Once the firmware is available the FPGA VM can be started and the hardware deployed on it. At this point the VM will contains the BM architecture that is ready to run the BM software, after that the application is ready to be used.

Due to the dynamical nature of the BM architecture, it is possible to put the whole process in the form of a BM accelerator as a service (BMAaaS). Specifically, given an high level description of a computational problem, either in the form of graph or Go source code, a first cloud service builds the accelerator and a second one starts a FPGA enabled VM to run an accelerated application.

6. Future work and benchmarks

ML and DL have become a popular solution also in High Energy Physics [19]. The standard software stack used to train and exploit DL based systems consists of Tensorflow used via a Keras abstraction. While Keras presents a simplified interface able to hide most of the complexity, Tensorflow is the real engine, able to handle tens of thousands inputs. Tensorflow graphs are inherently not serial in processing, thus performing better than standard sequential algorithms for complex problems.

A case where DL is currently heavily tested is the reconstruction of tracks into complex pp collisions at LHC. Track reconstruction is the most powerful tool when probing complex event topologies, and is currently attempted by combining the signals from large silicon detectors, able to flag the passage of each charged particle on a series of thin silicon layers in the presence of a strong magnetic field. Since O(1000) different tracks are expected per event at HL-LHC (2026+), the problem is essentially combinatorial and its computational performance scale worse than linearly with the number of tracks.

A solution is currently being developed using CNNs implemented via Keras on a TensorFlow back-end, on large systems using GPGPU. While the results are promising, the utilization of the solution is limited at offline studies, since requires a large computing power (dissipating > 200 W per machine) and cannot be put close to the detectors. An interesting test would be to try and move the network to hardware with lower power dissipation, and able to survive closer to the detectors.

The test we want to attempt is to use the BondMachine ML capability in order to implement on FPGA a trained network, which can output track seeds. The approach with respect to GPGPUs is largely different in the power dissipation, which is important make the computation at the edge.

Collider experiments already use extensively FPGAs at trigger level, using the low power profile and higher radiation hardness. Using the same infrastructure, it is feasible to test BondMachine created networks on real / realistic data streams, without the overhead of programming trigger algorithms in xHDL code and instead profiting from DL techniques in such environments.

7. Conclusion

The BondMachine is a new kind of computing device made possible in practice only by the emerging of re-programmable hardware technologies such as FPGA. The main goal of our project is the construction of a computer architecture where the hardware is shaped by the problem one aims to solve. Clearly this approach bring to an increased computing power and flexibility yet keeping a standard way of programming it.

Following these reasons, the compiler is not anymore a software that translates an high level source code to a general purpose machine binary code, it becomes a software that creates the architecture, the binary code and the HDL code to run on FPGA devices.

Finally we want to underline as whenever high performances are required, we expect that the reduction of the number of hardware/software layers will surely lead to an increase of the overall performances. The fact we will use a register machine (processor), and not for example directly the FPGA, means that will be relatively easy to import well known and already used computer science techniques on a BM.

Respect to any other classic computational system in the BM project the optimization process is extended up to the hardware level. Indeed if the program does not expect the processor to implement a particular operation, the resulting hardware will not have that instruction. Thus we are able to obtain a software/hardware system completely shaped by the given Computational task one aims to solve.

Over the abstraction of the BM it is possible to create a full computing Ecosystem, ranging from small interconnected IoT devices to Machine Learning accelerators.

References

[1] D. M. Chitty, "Fast parallel genetic programming: multi-core cpu versus many-core gpu," *Soft Computing*, vol. 16, no. 10, pp. 1795–1814, 2012.

[2] D. Luebke, M. Harris, N. Govindaraju, A. Lefohn, M. Houston, J. Owens, M. Segal, M. Papakipos, and I. Buck, "Gpgpu: general-purpose computation on graphics hardware," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, p. 208, ACM, 2006.

[3] A. Munshi, "The opencl specification," in *2009 IEEE Hot Chips 21 Symposium (HCS)*, pp. 1–314, IEEE, 2009.

[4] D. Bacon, R. Rabbah, and S. Shukla, “Fpga programming for the masses,” *Queue*, vol. 11, pp. 40:40–40:52, Feb. 2013.

[5] A. A. Donovan and B. W. Kernighan, *The Go Programming Language*. Addison-Wesley Professional, 1st ed., 2015.

[6] K. Ashton *et al.*, “That ‘internet of things’ thing,” *RFID journal*, vol. 22, no. 7, pp. 97–114, 2009.

[7] D. Michie, D. J. Spiegelhalter, C. Taylor, *et al.*, “Machine learning,” *Neural and Statistical Classification*, vol. 13, 1994.

[8] P. Baldi, P. Sadowski, and D. Whiteson, “Searching for exotic particles in high-energy physics with deep learning,” *Nature communications*, vol. 5, p. 4308, 2014.

[9] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, “Tensorflow: A system for large-scale machine learning,” in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pp. 265–283, 2016.

[10] A. Gulli and S. Pal, *Deep Learning with Keras*. Packt Publishing Ltd, 2017.

[11] B. Sotomayor, R. S. Montero, I. M. Llorente, and I. Foster, “Virtual infrastructure management in private and hybrid clouds,” *IEEE Internet computing*, vol. 13, no. 5, pp. 14–22, 2009.

[12] G. Mateescu, W. Gentzsch, and C. J. Ribbens, “Hybrid computing—where hpc meets grid and cloud computing,” *Future Generation Computer Systems*, vol. 27, no. 5, pp. 440–453, 2011.

[13] C. A. R. Hoare, “Communicating sequential processes,” in *The origin of concurrent programming*, pp. 413–443, Springer, 1978.

[14] S. Haykin, *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994.

[15] B. Seo, M. Shin, Y. J. Mo, and J. Kim, “Top-down parsing for neural network exchange format (nnef) in tensorflow-based deep learning computation,” in *2018 International Conference on Information Networking (ICOIN)*, pp. 522–524, IEEE, 2018.

[16] “Bmapi repository.” https://github.com/lstorchi/bmapi. Accessed: 2019-04-01.

[17] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and R. W. Stewart, *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Strathclyde Academic Media, 2014.

[18] M. Ryan and F. Lucifredi, *AWS System Administration: Best Practices for Sysadmins in the Amazon Cloud*. " O’Reilly Media, Inc.", 2018.

[19] A. M. Sirunyan, A. Tumasyan, W. Adam, F. Ambrogi, E. Asilar, T. Bergauer, J. Brandstetter, M. Dragicevic, J. Erö, A. E. Del Valle, *et al.*, “Observation of higgs boson decay to bottom quarks,” *Physical review letters*, vol. 121, no. 12, p. 121801, 2018.