UNIVERSIDADE D
COIMBRA

Faculdade de Ciências e Tecnologia
Departamento de Engenharia Electrotécnica e de Computadores

Bayesian inference for artificial perception using OpenCL on FPGAs and GPUs

Rodrigo de Oliveira Lourenço Lopes

Supervisor:
Prof. Doutor Jorge Nuno de Almeida e Sousa Almada Lobo

Juri:
Prof. Doutor Rui Paulo Pinto da Rocha
Prof. Doutor Jorge Nuno de Almeida e Sousa Almada Lobo
Prof. Doutor Gabriel Falcão Paiva Fernandes

Dissertação de Mestrado Integrado de Engenharia Electrotécnica e de Computadores

Coimbra, 2019

Abstract

This dissertation project addresses the implementation of Bayesian inference on FPGAs and GPUs, following a top-down approach and using OpenCL. The target application of this Bayesian inference algorithms is artificial perception in robotics. The aim is to improve the power efficiency of Bayesian inference computations. Previous work at our university in the scope of an European project followed a bottom-up approach and developed a toolchain that enabled having custom circuits for Bayesian inference on reconfigurable logic. These had better power efficiency than desktop solutions, but require more design effort. In this work the goal is to use already available vendor tools, namely the OpenCL support from Intel (formerly Altera), to explore the design space in search of low power efficient solutions. To achieve this, the same benchmark problem used in previous works is going to be applied, tested in various dimensions in order to study scaling challenges. The main metrics analysed are nominal power, energy consumed, latency and result's precision. As expected the results show a great gain in power efficiency in relation to desktop solutions, and comparable performance with previous works developed in the context of the project BAMBI, but with gain in point precision, integration and usability.

Resumo

Este projecto de dissertação aborda a implementação de um algoritmo de inferência Bayesiana em FPGAs e GPUs seguindo uma abordagem "top-down" e usando OpenCL. Este algoritmo de inferência Bayesiana tem como foco em aplicações de percepção artificial para robótica. O objectivo é melhorar a eficiência energética de computações de inferência Bayesiana. O trabalho previamente desenvolvido na nossa universidade no âmbito de um projecto europeu seguiu uma abordagem "bottom-up" e desenvolveu uma "toolchain" capaz de ter circuitos personalizados para inferência Bayesiana em lógica reconfigurável. Estes tinham maior eficiência energética do que soluções implantadas tipicamente em "desktops", porém requeriam significativamente maior esforço em design. Neste trabalho, a ideia é usar ferramentas comerciais já disponíveis, nomeadamente OpenCL suportado actualmente pela Intel (antes pela a Altera), para explorar todo o espaço de design de modo a encontrar soluções de baixo custos energéticos. Para o fazer, é usado o mesmo problema de "benchmark" utilizado em trabalhos anteriores, o qual será testado em várias dimensões de modo a poder estudar os problemas de escalonamento. As principais métricas usadas em análise são potência nominal, energia consumida, latência, e a precisão de resultados. Como esperado os resultados mostram um ganho de eficiência energética em relação a soluções de "desktop", e performance comparável a trabalhos anteriores desenvolvidos no âmbito do projecto BAMBI, mas com ganhos em precisão, integração e na usabilidade.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Objectives and Key Contributions 4

1.3 Related Work 5

1.4 Overview 6

2 Background on Bayesian Inference and Computation Hardware **7**

2.1 Performing Bayesian Inference 7

2.1.1 Bayesian Inference 7

2.1.2 The BAMBI Project 9

2.1.3 ProBT Software 11

2.2 Hardware for Computation of the Bayesian Inference 12

2.2.1 GPUs and FPGAs 12

2.2.2 OpenCL 15

2.2.3 FPGA Workflow 17

3 Our Implementation of Bayesian Inference on OpenCL **19**

3.1 Work Methods 19

3.2 Benchmark Model 20

3.2.1 Boat Localization Inference Problem 20

3.2.2 Generating the Bayesian Model 21

3.3 Information Flow 26

3.3.1 Host Side Control 26

3.3.2 Bayesian Inference Kernel 27

3.4 Double vs Single Precision Floating-Point Numbers 29

3.5 Evaluation Metrics 31

3.5.1 Precision 31
3.5.2 CPU Power Estimation 32
3.5.3 GPU Power Estimation 32
3.5.4 FPGA Power Estimation 33
3.6 Framework and Hardware 34

4 Experimental Results **35**
4.1 Finding the Position of the Boat 35
4.2 Scalability and Resource Usage 36
4.3 Results Precision 37
4.4 Execution Times 38
4.5 Power Consumption 41
4.5.1 GPU Energy Consumption 41
4.5.2 FPGA Energy Consumption 41
4.5.3 CPU Energy Consumption 41
4.6 Computing with Double Precision vs Single Precision 42
4.7 Comparison With Previous Works 45

5 Conclusion **48**
5.1 Result's Discussion 48
5.2 Hardware and Software Limitations 49
5.3 Future Work 50

List of Figures

2.1 Schema of BAMBI organization. [9]. 10
2.2 Island-style global FPGA architecture. A unit tile consists of Configurable Logic Block (CLB), Connect Box (CB) and Switch Box(SB). 12
2.3 Zoom-out of a general circuit logic present on a GPU card. 13
2.4 FPGA Workflow 17

3.1 Boat Localization Problem [15] 20
3.2 Likelihood of the boat location for the first landmark on a 64 by 64 model, where (a) for the distance sensor model when the distance is 32, and (b) for the angle sensor model when the angle is 0^o degrees [10]. 21
3.3 Likelihoods LUT Access 22
3.4 Boat location problem formal specification [15]. 25
3.5 Single and double bit arrangement. 29

4.1 Finding the Positions of the Boat through Bayesian Inference. 35
4.2 FPGA Resource utilization for 16x16 grid dimensions. 36
4.3 FPGA Resource utilization for 32x32, 64x64, and 128x128 grid dimensions. 36
4.4 FPGA and GPU Execution Times for one Bayesian Inference Iteration on Device. 40
4.5 FPGA and GPU Power Consumption for one Bayesian Inference Iteration on Device. 42
4.6 KL divergence results of boat example for different grid sizes and increasing energy consumption of the BM1 (background), OpenCL on FPGA (blue dotted) and GPU (red dotted), and finally our improved version of the CPU algorithm (pink dotted). 45

List of Tables

3.1 Information captured on the GPU hardware during the Bayesian inference algorithm execution 33

4.1 Kullback-Leibler divergence for each grid size 37
4.2 Posterior Probability Matrix Normalization Time on CPU 39
4.3 Bayesian Inference in GPU 39
4.4 Bayesian Inference in a single FPGA 39
4.5 Bayesian Inference in CPU using ProBT version for one iteration 40
4.6 Bayesian Inference in CPU using Optimized C++ version 40
4.7 Avarege Energy input on Devices 41
4.8 Power Consuption of One Iteration Bayesian Inference on GPU 41
4.9 Power Consuption of One Iteration Bayesian Inference on FPGA 41
4.10 One Iteration Bayesian Inference in CPU using Optimized C++ version . . 41
4.11 Average Transfer Times to GPU 43
4.12 Using Single 43
4.13 Ratio Between single and Double Transfer Times 44
4.14 Avarege Execution Times on GPU 44

Acronyms

ASIC Application-Specific Integrated Circuit

API Application Programming Interface

AI Artificial Intelligence

BAMBI Bottom-up Approaches to Machines dedicated to Bayesian Inference

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

DRAM Dynamic Random Access Memory

FPGA Field Programmable Gate Array

GPU Graphical Processing Unit

LUT Look Up Table

OpenCL Open Computing Language

RTL Register-Transfer Level

Chapter 1

Introduction

1.1 Motivation

One of the main challenges in the path to full robotic automation is perception. Perception is the organization, identification, and interpretation of sensory information in order to represent and understand the presented information, or the environment [1].

Perception can be split into two processes [2], the processing of the sensory input, which transforms low-level information to higher-level information, and the processing connected with a person's concepts and expectations, restorative and selective mechanisms that influence perception.

In the context of living beings, biological intelligence [3] comes from an amazing duality of arriving at conclusions based on perception of patterns of our inputs (senses), and the contrary, conclusions based on very structured and rational decisions. Both forms are different, but ultimately complement each other. Machine-based intelligence also comes in two forms: Deep learning based Artificial Intelligence [4] interprets patterns in data to arrive at conclusions mimicking the perception based intelligence of our brain, and standard instruction-by-instruction computing.

As is explicit in the title of the dissertation, we focus on artificial perception which deals with the capability of a computer system to interpret data in a manner that is similar to the way humans use their senses to relate to the world around them [5].

When developing algorithms capable of deriving perception meaning from sensors, sometimes we stumble upon problems in which the complexity exponentially increases with the adding of inputs.

A probabilistic algorithm is a method where the result and/or the way the result is obtained depend on chance. In some applications the use of probabilistic algorithms is natural, e.g. simulating the behaviour of some existing or planned system over time. In this case the result by nature is stochastic. In some cases the problem to be solved is deterministic but can be be transformed into a stochastic one and solved by applying a probabilistic algorithm (eg. numerical integration, optimization). For these numerical applications the result obtained is always approximate, but its expected precision improves as the time available to use the algorithm increases [6][7].

Another way to improve precision is to use Bayesian inference that derives the posterior probability as a consequence of two antecedents: a prior probability and a "likelihood function" derived from a statistical model for the observed data [8].

However, with increasing cardinality of Bayesian Inference computations, even simple perception problems can easily overload traditional CPU (sequential based) implementations.

In this context, adapting the hardware to the specific constraints of probabilistic computation, using parallelism via multicore architectures, it is shown to achieve several orders of magnitude in gains.

Such probabilistic computations can be highly independent for the individual candidate solutions, making them excellent candidates for cluster computations because they require little communication and synchronization. It is possible to specify a common parallel control structure as a generic algorithm for probabilistic cluster computations. Such a generic parallel algorithm can be "glued" together with domain-specific sequential algorithms in order to derive approximate parallel solutions for different intractable problems.

The BAMBI EU FET project, developed new computing machines largely inspired by biology with a main focus on those to perform probabilistic inferences efficiently. And in a long-term view, this might give rise to completely new hardware based on these principles [9].

The numerous simulations that were performed during the BAMBI project showed that these new highly parallel stochastic architectures could solve very efficiently several inference problems, including sensor fusion, Bayesian filters, parameter learning and approximation of highly intractable inference problems [9].

In this context previous works in the field used reconfigurable logic (FPGA) to emulate Bayesian gates and investigate the parallelization of probabilistic calculus on assemblies of gates. This dissertation takes another direction and performs exact Bayesian Inference, instead of approximate using a top-down approach with heterogeneous programming, namely using OpenCL on FPGAs and GPUs, with the intention to compare results. The main goal is to investigate the trade-off between energetic consumption and precision in relation to previous works in the field. Having a working model of the artificial perception hardware using Bayesian inference enables its use in robotics applications, further advancing towards autonomous robots able to evolve in uncertain and ambiguous real-world situations.

1.2 Objectives and Key Contributions

Objectives

This dissertation explores the solution space in search of low power efficient method on the application of Bayesian inference orientated towards artificial perception in robotics.

In previous works a toolchain was used that enabled having custom circuits for approximate Bayesian inference [10], this engine (known as BM1) presented better power efficiency than desktop solutions, but requires more design effort.

In this work OpenCL implementation of exact Bayesian inference is used running on FPGA and GPU. An analysis and comparison is going to be conducted between this method and previous works as well as a discussion of best possible applications for each implementation.

Also a comparison between single vs double floating point number program approaches was pursued in order to further explore the solutions space for Bayesian inference.

Key Contributions

The exact Bayesian inference implementation on OpenCL using GPU presents significantly less latency, total energy consumption, and development times, but considerable more nominal power required in relation to the FPGA approach. The OpenCL implementation (both GPU and FPGA) has a much higher result precision and lower design times in relation to the BM1 machine but with higher energetic costs and nominal power, as was initially predicted.

On the comparison between single vs double floating point number program approaches, both presented similar computation times of the Bayesian inference in both hardware, but single floating point number program presented lower the transfer times between host and device and lower memory occupied in device. The loss in precision from the double floating point number to single in the problem used as benchmark was nearly negligible.

1.3 Related Work

During the development of this dissertation some works were used either as theoretical base, or for orientation due to some overlapping topics addressed, namely:

Ferreira et al. demonstrate tractable implementation of exact Bayesian inferece with a real-time OpenCL vision-based model to estimate gaze direction in a human-robot interaction (HRI) using GPUs [11]. On the other hand, on a Bottoms-Up Approach [12], unconventional hardware architectures are explored to perform the required inference. This was the course of action of BAMBI project: a generic toolchain was developed to generate unconventional hardware for probabilistic inference from stochastic building blocks, allowing bit level parallelism [13]. During this project, a Boat Location problem was used as a case study [14], this is used in this dissertation as a performance benchmark. The results obtained in this project are used as comparison to the ones in this dissertation, accompanied with trade-off analysis. Later, Mira continued the work of the BAMBI project, integrating ROS and adding a PCIe interface to extract the full capabilities of the FPGA mini-cluster [15], this same setup was used in this dissertation.

1.4 Overview

• In chapter 1 we describe the work that its being proposed as well the motivations behind it. We give a context of fields relating to the project and review of the work already conducted on the subject.

• On chapter 2 the background presents the theoretical foundations of this dissertation briefly described.

• In chapter 3 we discuss the work methods used to approach the problems faced. In this chapter it's also addressed the framework/tools used during the development.

• In chapter 4 we present the results obtained, accompanied with a critical analysis discussing their relevance.

• In chapter 5 a brief conclusion is presented with final thoughts about the thesis, and some suggestions about future work.

Chapter 2

Background on Bayesian Inference and Computation Hardware

2.1 Performing Bayesian Inference

2.1.1 Bayesian Inference

Bayesian inference is a method of statistical inference in which Bayes' theorem is used to update the probability for a hypothesis as more evidence or information becomes available. Bayesian inference is an important technique in statistics, and especially in mathematical statistics. It has found application in a wide range of activities, including science, engineering, philosophy, medicine, sport, and law. In the philosophy of decision theory, Bayesian inference is closely related to subjective probability, often called "Bayesian probability" [16].

In statistical inference, there are two broad categories of interpretations of probability: Bayesian inference and frequentist inference. These views often differ with each other on the fundamental nature of probability. Simply put Frequentist inference defines probability as the limit of an event's relative frequency in a large number of trials, and only in the context of experiments that are random and well-defined. Bayesian inference, on the other hand, can assign probabilities to any statement, even when a random process is not involved. In Bayesian inference, probability is a way to represent an individual's degree of belief in a statement or given evidence.

Bayesian inference is a method of statistical inference in which Bayes' theorem is used to update the probability for a hypothesis as more evidence or information becomes available.

Formally put Bayesian inference derives the posterior probability as a consequence of two antecedents, a prior probability and a "likelihood function" derived from a statistical model for the observed data. Bayesian inference computes the posterior probability according to the Bayes' theorem:

$$P(H \mid E) = \frac{P(E \mid H) \times P(H)}{P(E)} \tag{2.1}$$

However, Bayesian inference in some applications has been limited by the computational requirements for real-time implementations. These computational limitations come sometimes from the large amounts of data being handled, this might lead to tractability problems.

In a straightforward way, a tractable problem is one that with the linear increase of inputs the solving time of the problem increases in near linear way as well.

The intractability issues come from the fact that Bayesian programs generally include a large number of inter-dependent random variables, that even though these have been discretized, the complexity of the Bayesian models results in an exponential increment in solving times with the linear increment of variables.

There are two types of inference techniques: exact inference and approximate inference. Exact inference algorithms calculate the exact value of probability $P(X \mid Y)$. Algorithms in this class include the elimination algorithm, the message-passing algorithm, and the junction tree algorithms. The time complexity of Bayesian inference is NP-hard [**?**] that stands for "non-deterministic polynomial acceptable problems" which informally means that the problem cannot be solved in a polynomial time in relation the inputs. To tackle this situation where some exact inference problems have unreasonable solving

times, approximate inference techniques are used because they require less computations time with the cost of reduced accuracy. Some Approximate inference algorithms include stochastic simulation and sampling methods, Markov chain Monte Carlo methods, and variational algorithms.

2.1.2 The BAMBI Project

The BAMBI project [9], had ambition to lay the foundations of new computing machines, largely inspired by biology, and specially designed to efficiently perform probabilistic inferences. For that, their main effort focused on three research axes: to develop a new theory of probabilistic computation, to investigate how simple living organisms process information and perform basic inferences, to build completely new hardware based on these principles.

In the first axis, Bayesian algebra is developed and also proposed the Bayesian gates that can be seen as the extensions of the Boolean algebra and logical gates, the essential components of current computers. Then, new architectures capable to solve probabilistic inference problems of various complexity were detailed and simulated. A key feature of these architectures is to rely on stochastic computation and on components that behave randomly. The numerous simulations that were performed during the project showed that these new architectures could solve very efficiently several inference problems, including sensor fusion, Bayesian filters, parameter learning and approximation of highly intractable inference problems.

In the second axis, the intention was to establish the link between some general principal governing biological signal processing at various space-time scales and the basic principles of probabilistic computation. This enabled to successfully developed probabilistic inference model for biochemical kinetics cast in the form of a nonlinear finite state machine which is massively parallel.

In the third axis, two tracks were followed. The first track is for short-term, it is based on the use of existing fully integrated circuits, namely the reconfigurable array of logical elements (FPGA), which is the area we will delve the most in the development

of the dissertation. The long-term track is based on existing components, the super paramagnetic tunnel junctions (SMTJs). The short-term track was very important, since it allowed to produce real circuits implementing the new architectures simulated in the first axis. The long-term track is very promising since, as previously shown, the energy efficiency will be several orders of magnitude better than one could expect with existing hardware.

The figue 2.1 represents the three main axes (theory of probabilistic computation, probabilistic inference in biology, and hardware implementation) which are oriented according to the bottom-up approach.

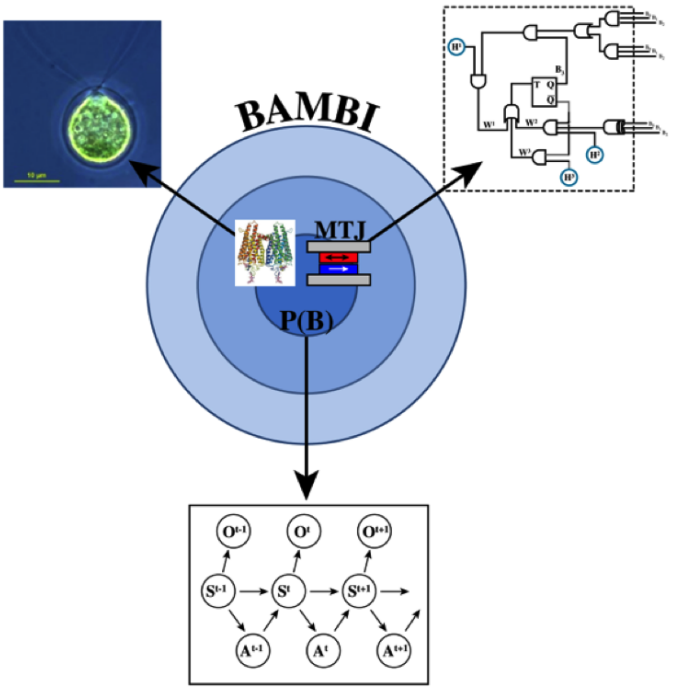


Figure (2.1) Schema of BAMBI organization. [9].

2.1.3 ProBT Software

ProBT is a C++ library for developing efficient Bayesian software. This library consists of two main components, a friendly Application Program Interface (API) for building Bayesian models and a high-performance Bayesian inference and learning engine allowing execution of the probability calculus in exact or approximate ways. ProBT aims to provide a programming tool that facilitates the creation of Bayesian models and their reusability. Its main idea is to use "probability expressions" as basic bricks to build more complex probabilistic models. The numerical evaluation of these expressions is accomplished just-in-time: computation is done when numerical representations of the corresponding target distributions are required. This property allows designing advanced features such as submodel reuse and distributed inference. Therefore, constructing symbolic representations of expressions is a central issue in ProBT [17].

The model adopted in this dissertation used the ProBT software to build the inference sensor fusion model of our boat localization problem from which the inference likelihoods table is created, along with one iteration of an exact inference vector output of the model that will be used as a precision benchmark to compare with the inference employed in our work. More of this subject will be discussed in the methodology chapter.

2.2 Hardware for Computation of the Bayesian Inference

2.2.1 GPUs and FPGAs

Whether you're talking about real-time stock trading, online searches, artificial perception, faster results are among one of the top priorities. In this search for computational speed, a debate on the best accelerators has emerged. A great portion of times this debate has focused on FPGA's vs GPU's as these two are probably the most common hardware solutions more widely used for task acceleration.

The FPGA stands for Field Programmable Gate Array and it is a type of device that is widely used in electronic circuits. As represented in the figure 2.2 FPGAs are semiconductor devices that contain programmable logic blocks and interconnection circuits. It can be programmed or reprogrammed to the required functionality after manufacturing, hence the term "field-programmable". The FPGA configuration is generally specified using a hardware description language, similar to that used for an application-specific integrated circuit (ASIC), which is a very design-intensive process.

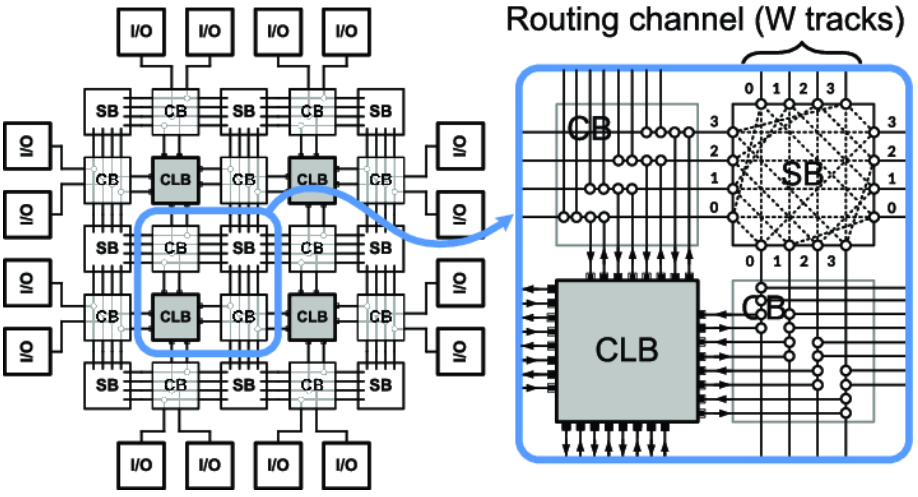


Figure (2.2) Island-style global FPGA architecture. A unit tile consists of Configurable Logic Block (CLB), Connect Box (CB) and Switch Box(SB).

A GPU is a heterogeneous chip multi-processor that is designed in most cases to be highly tuned for graphics. Although presenting significantly lower clock frequencies

than CPUs, their highly parallelized structure makes them more efficient in relation to general-purpose CPUs in algorithms that are possible to unfold in several parallel threads in the various cores or CU (compute units) as is shown in thhe figure 2.3 The GPU's offer higher floating-point throughput, data parallelism orientated architecture and higher memory bandwidth than processors, characteristics that make it a prime candidate to be used as accelerators in heterogeneous programming.

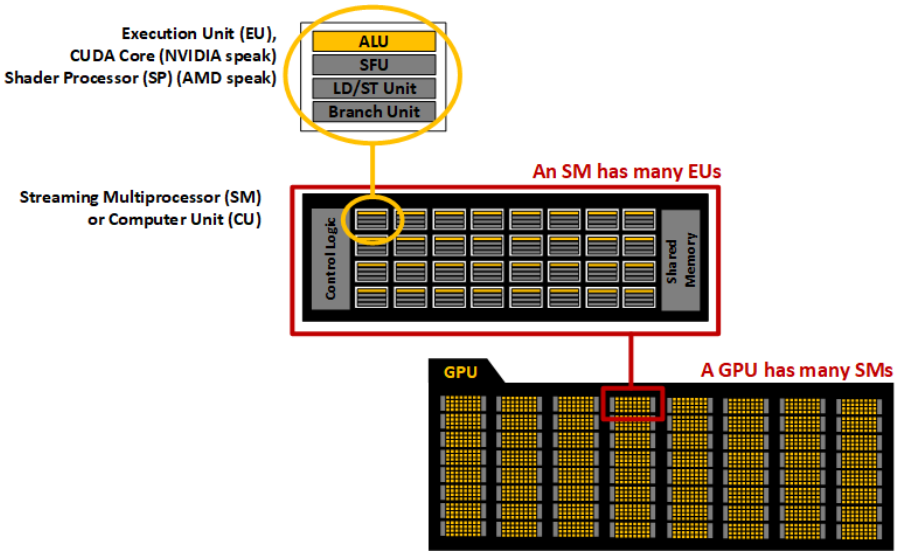


Figure (2.3) Zoom-out of a general circuit logic present on a GPU card.

FPGA and GPU manufacturers continuously compare against CPUs, sometimes making it sound like they can take the place of CPUs. But it's important to note that both FPGA's and GPU's do not function on their own without a server and neither can replace the server's CPU with the exception of FPGA's with system on chip, that has its own CPU and operating system, allowing it to function as a "server + accelerator" alone.

Depending on how fast you want or need your programs to run, the acceleration comes at a price. After the cost of acquisition, the final price also includes the amount of heat generated, the power required, and the huge increase in development times of applications to take full advantage of the new hardware [18].

The FPGA offers some clear advantages when it comes heterogeneous. These are the ability to tweak and tune the underlying hardware architecture and use software-defined

processing which allows FPGA-based platforms to deploy super problem-specific solutions to each problem.

Also, the FPGA's low latency offers unique advantages for mission-critical applications, such as autonomous vehicles and manufacturing operations. This comes from the fact that the data flow pattern in these applications may be in streaming form, requiring pipelined-oriented processing.

Power efficiency is another factor to have in great consideration in cases such as long term use, or energy-critical systems (eg. Devices with batteries). The FPGA's can have low power requirements and high performance per watt, since the logic has been tailored for specific applications and workloads, and thus extremely efficient in executing said application. For example in an FPGA the control structure is hardwired, and there is no need to fetch and decode instructions.

The modern high-performance GPU's have very high-bandwidth DRAM interfaces, which typically outperform those of FPGAs implemented using comparable technologies. However, external DRAM access consumes a significant amount of power. This means that the data organization for FPGA-bound implementations must be carefully optimized to use on-chip resources, even for kernel-to-kernel communication.

2.2.2 OpenCL

With the increase of the need for computational power, so the usage of accelerators has exploded in recent years. A typical debility of these systems has been the much larger effort required to program than conventional CPU based routines, as these have separate memories that require additional buffers and memory transfers, different ways to launch code, the need to specify details that do not exist in CPUs, and an overall higher level of overhead. Another problem related to most tools available to heterogeneous programming is that these are tied to a specific family of devices (like CUDA in the case of Nvidia GPUs), wich severely restricts the portability of applications between systems, resulting in the rapid obsolescence of the code built on them, and higher design times for cross-platform programs.

OpenCL is an answer to this latter problem as it is a framework for writing programs that execute across heterogeneous platforms consisting of central processing units, such as GPUs, FPGAs, DSPs and other processors or accelerators. OpenCL is a parallel programing language developed by Khronos group built upon C/C++ that provides a standard interface for parallel computing using task- and data-based parallelism [19]. The portability in OpenCL gives it a hand up on the other accelerating frameworks such as CUDA, which can only be used to create programs for NVIDIA GPU's.

Although presenting all these benefits, one issue is that OpenCL does not offer automatic optimal performance portability across multiple devices. This means that even though a specific OpenCL code can be executed on multiple devices from different vendors, it's not assured by the framework that the code will be compiled to the absolute best specifications of each hardware, and so different performances are to be expected. Some tweaking may be required to do on an OpenCL code when transitioning devices to take advantage of each architecture to the fullest.

During the development of this work, we noticed that the biggest difference between optimizing code for a GPU and for a FPGA lies in the different targeted architectures. In the case of a GPU, the programmer tries to achieve the best mapping of a problem into the fixed restrains of each hardware architecture. On the other hand for an FPGA, the

task is to guide the compiler (using pragmas for example) to generate optimized memory and computing architectures for each kernel in the application.

Even though the optimization of the code for FPGA tends to require more effort, since the architecture must be adapted to the code and vice-versa, we will argue in this thesis that the advantages gained in reduced energy consumption justify the additional work.

2.2.3 FPGA Workflow

The figure 2.4 shows the general FPGA workflow, to which usually the first step in is to convert the intended algorithm and divide it in the processes that are possible to be parallelized, these ones if long enough to be worth it to transfer them to an accelerator should be written in kernel in the OpenCL language, while the others should be maintained on the host side.

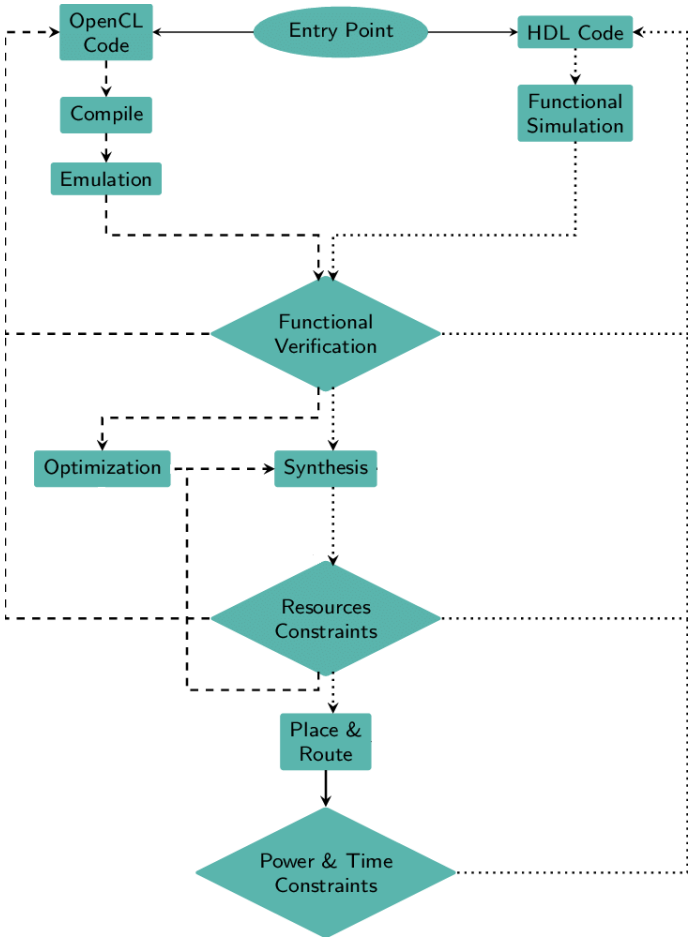


Figure (2.4) FPGA Workflow

After this process is completed we should take the OpenCl code and verify its functional correctness and build a software-based emulation. This emulations take significantly less time to build than the synthesis of the code to be deployed on the FPGA, helping us ensure that the outputs produced by the simulation are correct before going forward

on the development. It is also common in this stage to add test-benches to the host code which provides the inputs to and checks the outputs from the algorithm.

Once the functionality criteria is met, we can do a gross estimation of the performance of each individual kernel, compute unite, resource usage, etc and thus by taking also into account the targeted hardware specifications we can get an early idea of the final performance gains or losses.

After this, we generate the hardware design of the OpenCL kernels. Numerous compute units can be called and driven by the host code, in order to boost performance. Each workgroup can execute its tasks either in pipelined fashion, or by unrolling them, or both. In most cases pipelining provides the best cost/performance trade-off, but sometimes either the loops over the work items or some of the loops nested within them can be further unrolled to improve performance. After this, we proceed to hardware emulation (also known as RTL simulation) that is used to certify that the hardware is behaving correctly, verify if there is any resource limitation, and have a more precise notion of the overall performance. The physical design of each compute unit needs to be generated in this hardware emulation so this process is several times slower than the software emulation.

The last step consists on deploying the hardware design on the FPGA, running it and verify the real metrics such as time needed to run code, resources used, and the result's fidelity.

Chapter 3

Our Implementation of Bayesian Inference on OpenCL

3.1 Work Methods

One of this thesis objectives is the comparison of the OpenCL Bayesian Inference implementation of a well-known Boat Localization problem with a conventional CPU approach and with a heterogeneous computing platform running Bayesian Framework previously developed by BAMBI.

The OpenCL SDK libraries of Altera (now Intel FPGA) will allow us to focus on the program in a higher level of abstraction than if we were to use low-level hardware descriptive languages such as VHDL. This factor reduces design times and costs, but also the probable outcome of some losses in performance due to the general purpose of the language.

To enable the benchmarking of the tests performed since the beginning a Boat Localization problem was used as a performance measure (same used in previous works, namely in the project BAMBI EU FET).

3.2 Benchmark Model

3.2.1 Boat Localization Inference Problem

One of this thesis objectives is to test the performance of the OpenCL Bayesian Inference with a conventional CPU approach and with the Bayesian Machines previously developed during BAMBI project. This test is not representative of all Bayesian inference problems, but it was the only one used in the BAMBI project where an energy performance test was conducted.

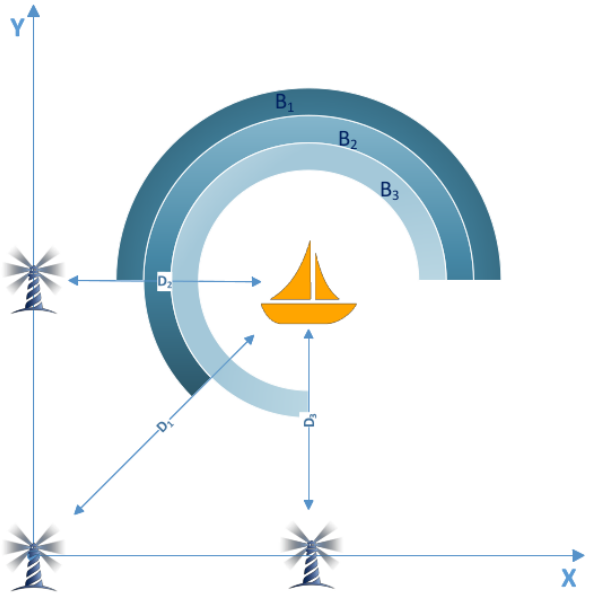
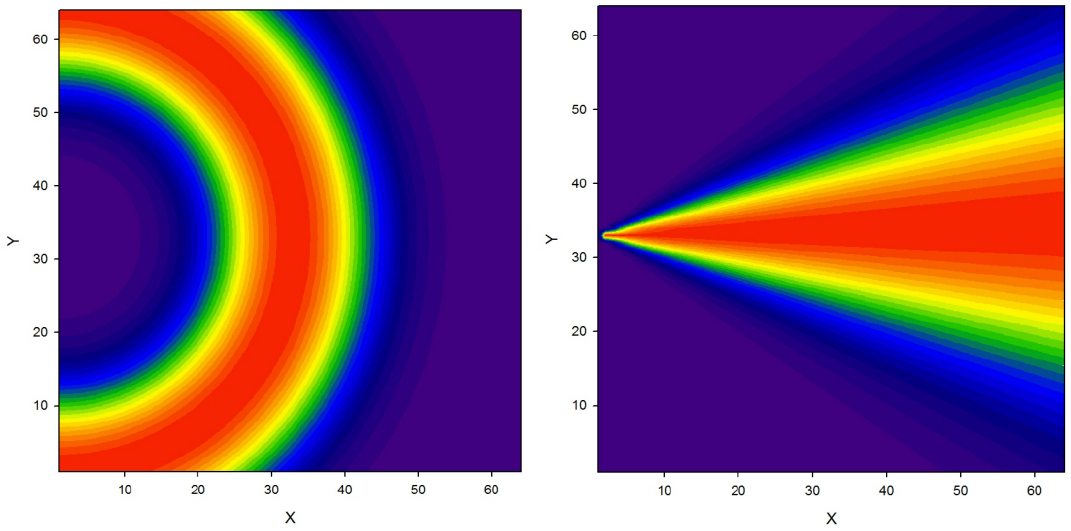


Figure (3.1) Boat Localization Problem [15]

To benchmark the tests performed a boat localization problem with variable complexity will be used. This problem consists in determining the position of a boat in a discrete square grid of variable size. The user provides the boat coordinates (x, y) as input to a program working concurrently with the OpenCL host program, this data is pre-processed by a routine and mathematically converted into sensor values. These new data are fed to the inference engine for processing. As shown in the figure 3.1, we consider the existence of three landmarks from which the boat is able to detect the distance (D 1 , D 2 , D 3) and bearing (B 1 , B 2 , B 3) of each.



(a) Distance sensor.

(b) Bearing sensor.

Figure (3.2) Likelihood of the boat location for the first landmark on a 64 by 64 model, where (a) for the distance sensor model when the distance is 32, and (b) for the angle sensor model when the angle is 0^{o} degrees [10].

The figure 3.2 illustrates the likelihood of the boast's position (in the middle of the grid) captured by a distance and bearing sensor respectively. The final likelihood of the boat's position corresponds to the product of likelihood map of all sensors. The boat problem will be scaled to several dimensions to test out multiple inference engine sizes.

3.2.2 Generating the Bayesian Model

Before we can run the Bayesian inference algorithm we first have to generate the fusion likelihoods.

To compute this file, the ProBT library is used on a program that computes the posterior probability distribution on the searched variable S (from the sensors), using the knowledge of the prior distribution P(S) and the set of conditional distributions P(K i

—S), where Z is a normalization constant:

$$P([S = s_j])K_1, ..., Kn) = \frac{1}{Z}P(S = s_j)) \prod_{i=a}^{N} P(K_i[S = s_j]) \tag{3.1}$$

After this operation the computation tree is extracted and a set of probability distribution Look-up Tables (LUTs) is generated. After this, the LUT, along with other control variables are preloaded onto the device, next the inference is performed and the results stored on the host machine.

As illustrated in figure 3.3 the fusion likelihoods file holds for every possible value of each of the six sensors (3 distance sensors, 3 bearing sensors). This file holds the likelihood associated with each combination of cell, sensor, and sensor's value (discrete). The file itself is structured so it can be easily accessed as Lookup-table (LUT), with cell number, sensor number, and sensor value, functioning as the keys.

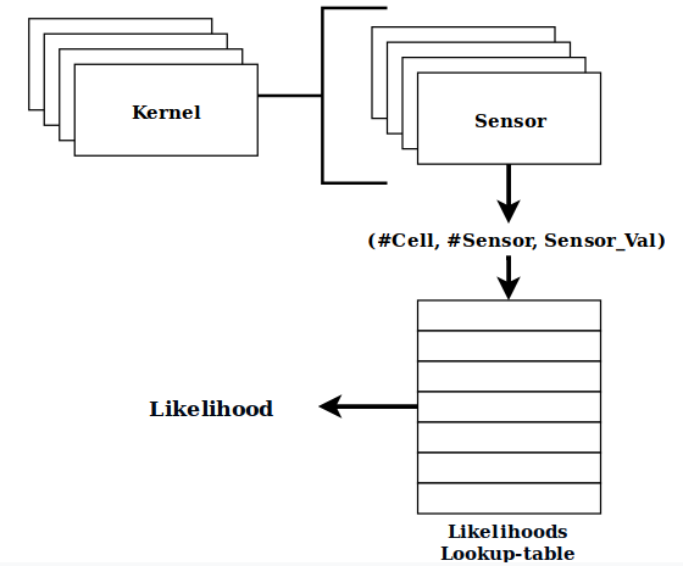


Figure (3.3) Likelihoods LUT Access

While the fusion priors represent the current likelihood of the presence of the boat in each of the cells on the grid. These two files are loaded on the device's memory during the execution of the Bayesian inference engine, with the fusion priors being updated as the outcome of the inference algorithm.

To generate both these files we adopt an algorithm in python that uses the ProBT libraries in which the user introduces the size of the model in the number of bits for de lateral dimensions (8x8, 16x16, 32x32, 64x64, etc.). We also generate the exact inference distribution given some x and y coordinates that later are used for benchmark precisions comparations with the approximate inferences tested on the other hardware.

From a high level of the abstraction, this code performs the following steps:

- From the values introduced by the user, we define the dimensions of the grid in which the model will be created.
- Define the probabilistic variables associated with location.
- Define the probabilistic variables associated with range and bearing.
- Define the function necessary to compute the mean and the standard deviation of the distance to each beacon knowing the position X and Y.
- Define the function necessary to of the bearing to each beacon knowing the position X and Y.
- Build the specification, and define the distributions related to distances and bearings of the model.
- Generate the exact inference distribution.

The Inference problem corresponds to computing the probability distribution over the X and Y coordinates, knowing the outputs of the six sensors. Therefore, considering:

- M_m = Position cell of index m
- E_k = Value of Sensor k
- $P(M_m)$ =Set of prior probabilities for each cell of index m
- $Posterior = \frac{Likelihood \times Prior}{Evidence} \Longleftrightarrow$
- $P(model \mid data) = \frac{P(data|model) \times P(model)}{P(data)} \Longleftrightarrow$

From Bayes Theorem, we gather that: $P(M \mid E) = \frac{P(E|M)P(M)}{P(E)}$. As the partition M_m is finite, we can use the Law of Total Probability to define $P(E) = \sum_m P(E \mid M_m)P(M_m)$. From the above, we can define our new Posterior Ratio as:

$$P(M \mid E) = \frac{P(E \mid M) \times P(M)}{\sum_m P(E \mid M_m) \times P(M_m)} \tag{3.2}$$

Considering that E is in fact a set of k multiple independent sensors E_k , we have:

$$y_m = \frac{x_m \times a_m}{z} \iff P(M \mid E) = \frac{\prod_k P(E_k \mid M) \times P(M)}{\sum_m \prod_k P(E_k \mid M_m) \times P(M_m)}$$

With:

- y_m : Posterior ratio.
- a_m : Prior ratio.
- x_m :Likelihood.
- z :Normalization factor, scalar independent of of both m and k.

We can formalize this problem as a Bayesian Program in the ProBT language. The ProBT API is then used to generate a simplified computation tree. In addition, the probability distribution across all variables is calculated offline and stored in Lookup Tables (LUTs) in floating point variables.

Boat Locator

Description

Specifications

$InputVariables$

$D_1, D_2, D_3, B_1, B_2, B_3$

$Decomposition$

$P(D_1 D_2 D_3 B_1 B_2 B_3 \mid XY) =$

$P(D_1 \mid XY)P(D_2 \mid XY)P(D_3 \mid XY)P(B_1 \mid XY)P(B_2 \mid XY)P(B_3 \mid XY)$

$ParametricForm:$

$P(D_1 \mid XY) = \mathcal{N}(d_1, (5 + \frac{d_1}{10})^2)$

$P(D_2 \mid XY) = \mathcal{N}(d_2, (5 + \frac{d_2}{10})^2)$

$P(D_3 \mid XY) = \mathcal{N}(d_3, (5 + \frac{d_3}{10})^2)$

$P(B_1 \mid XY) = \mathcal{N}(b_1, (5 + \frac{b_1}{10})^2)$

$P(B_2 \mid XY) = \mathcal{N}(b_2, (5 + \frac{b_2}{10})^2)$

$P(B_3 \mid XY) = \mathcal{N}(b_3, (5 + \frac{b_3}{10})^2)$

$Identification$

$Alltablesprovidedbytheuser$

$Question:$

$P(X \wedge Y \mid [D_1 = d_1] \wedge [D_2 = d_2] \wedge [D_3 = d_3] \wedge [B_1 = b_1] \wedge [B_2 = b_2] \wedge [B_3 = b_3])$

Figure (3.4) Boat location problem formal specification [15].

The schema 3.4 demonstrates formally how the problem's model is specified, namely the variables used, to which is shown that the sensors values is dependent on the boats coordinates, to which are added a random value form a Gaussian distribution (to simulate the sensor's error margins), and the generated likelihood of each sensor is formed with a certain normal distribution (also visible in the figure 3.2), and finally that the boats position probability distribution is obtained by the product of all of the sensors' likelihoods maps.

3.3 Information Flow

3.3.1 Host Side Control

After these files are generated, we use two C++ functions, one that converts the user inputs that is controlling the boat position in the grid into sensor values in model, and sends this values via socket to a program running in parallel.

The second function named is the OpenCL host of the program, that effectively manages the bayesian inference computation. This program will detect and initialize the available devices (FPGAs or GPUs), create the context and queues for each device, create the buffers and load them with the Lookup Tables and sensor data to be passed to the computing kernels. In addition it partitions the inference problem and distributes it through the queues of each device. It is also required to pass the necessary control variables to each kernel, such as number of iterations, and dimensions of the problem. When the number of iterations is completed the device or devices returns the computed posterior probability distribution and stores it both as a heatmap and as a file, to be used as comparation to the exact inference distribution file previous generated.

All this is done in real time, for each time new values are sent from the interface controled by the user. The host program only loads the likelihoods table in the device on the first time, since it is already loaded on to the device, and on the following turns only the new sensor values are sent, and thus saving in transfer times from host to devices, since the likelihoods table is the biggest file by several orders of magnitude.

In all this process we are at the same time collecting all the metrics relevant to the performance of such methods, such as time, total energetic consumption, average energetic consumption per unit of time, peak energetic consumption, area of resources used (FPGA). And all these methods will be tested in FPGAs and GPUs, for different dimensions of the model, different number of iterations, and comparing also the performance of single vs double floating point number programs.

3.3.2 Bayesian Inference Kernel

Given the fact that in the inference algorithm for this specific case, the end product of the calculus is a likelihood attributed to each cell on the problem's grid, then the process of parallelizing the tasks was performed by assigning a cell to each kernel of the problem since the data of each is independent and can be isolated without synchronization. So when possible the number of kernels instantiated should be the same, or at least, the closest as the number of cells in the grid.

The kernel will run a simple function for every sensor value, and this function is based on the inference model previously created, that receives as inputs the pre-computed likelihoods, the dimensions of the grid, and sensor values, and outputs the index position of the likelihood table loaded on device. The product of these values returned by the likelihood table is then multiplied by the posterior probability on the kernel's slot, resulting in the updated slot's probability.

Algorithm 1 Bayesian Inference Kernel

Input: N, likelihoodsVector[], distanceLikelihoodsTable[], bearingLikelihoodsTable[], distanceSensorValues[], bearingSensorValues[], maxDist, maxBear, nClocks;

Output: likelihoodsVector[];

1: $s \leftarrow 0$
2: $y \leftarrow 0$
3: $i \leftarrow globalID()$
4: $count \leftarrow 0$
5: **while** $count < nClocks$ **do**
6: **for** $s < 3$ By 1 **do**
7: $index \leftarrow (i \times 3 \times (maxDist + 1)) + (s \times (maxDist + 1)) + (sensorValues[s])$
8: $currentLikelihood \leftarrow currentLikelihood \times distanceLikelihoodsTable[index]$
9: **for** $s < 6$ By 1 **do**
10: $index \leftarrow (i \times 3 \times (maxBear + 1)) + ((s - 3) \times (maxBear + 1)) + (sensorValues[s])$
11: $currentLikelihood \leftarrow currentLikelihood \times distanceLikelihoodsTable[index]$
12: **if** $count = 0$ **then**
13: $y \leftarrow likelihoodsVector[i] \times currentLikelihood$
14: **else**
15: $y \leftarrow y + (y \times currentLikelihood)$
16: $count++$
17: $likelihoodsVector[i] \leftarrow y$
18: return $likelihoodsVector[]$

During the development of the kernels, was noticed that the seamless migration between OpenCL kernels of GPU's to FPGA's and vice-versa was not ideal due to the hardware differences and the significant inefficiencies it would create. So the inference kernels used in both situations although performing the same tasks were created with very slight modifications.

Another point to highlight is that the kernels were created with no vendor-specific optimizations put in place to enable migration of the kernels to other machines, yet, in the FPGA's kernels, pragmas were used. These pragmas in OpenCL are pre-processed

directives that the compiler only follows if the targeted machine can execute them, and it means that this code specifications causes no hurdles when running in other machines to which the code was not designed with that particular case in mind.

3.4 Double vs Single Precision Floating-Point Numbers

In this work, we also studied and implemented the use of Double vs Single (Float in C/C++) variables in the Bayesian inference programs, to discover the most appropriate option in each scenario.

As figure 3.5 shows the single floating point number program uses 32 bits (4 bytes) in which 23 of those are used for the mantissa, and the remaining 9 are used for the number's sign (1 bit), and for the exponent (8 bits).

In the double floating point number program, as the name suggests, the double of the bytes are used (64 bits), where 52 are used for the mantissa, 11 for the exponents and 1 for the number's sign.

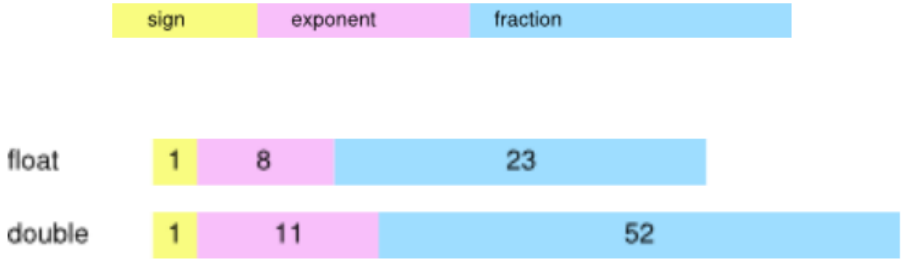


Figure (3.5) Single and double bit arrangement.

Obviously, between the two approaches, there's a precision trade-off. So, to find the number of precision digits we use the function $log_{10}(2^N)$, being N the number the bits that codify the mantissa of the number. Given this, we can determine that a single has approximately 7.22 digits of precision and 15.95 for double variables reserved for the mantissa. This without even counting the exponent, which also a great edge to the

double precision variable. This means that for the double variable the exponent can go from 10^{-1024} to 10^{1024} and the single 10^{-128} to 10^{128} (one bit is dedicated to the exponent sign) which in both cases the precision is far more than needed for this specific application.

Given the problem being tested, from one approach to the other, we find we have to duplicate data transfer (from host to device and vice-versa) and memory allocation on device.

Since that in our specific model the likelihoods file size is only dependent on the size of the grid itself, and that the size of this is solely dependent on the number of bits used to codify it, given the rest of the model the file's size follows this equation:

$$FileSize = N \times (MaxBearing \times Bs + MaxDistance \times Ds) \times S \tag{3.3}$$

Where the N represents the number of cells in the grid of the model, Bs and Ds respectively represent the number of bearing and distance sensors, S the size in Bytes of the variable being used and finally Max Distance and Max Bearing are the maximum possible values obtainable by those sensors in the context of that size of the model. So finally the equation translates to the following form:

$$FileSize = (2^{bits})^2 \times (2^{bits+2} \times 3 + 2^{bits+1} \times 3) \times S \tag{3.4}$$

In the function above, the word "bits" means the number of bits required to represent one of the coordinates on the grid (3 bits gives us 8x8, 4 bits 16x16, etc.). By this function we see that the size of the likelihoods file is only dependant on two variables, the grid size and the variable type, like the following table shows:

	single	Double
8x8	36 KBytes	72 KBytes
16x16	288 KBytes	576 KBytes
32x32	2 304 KBytes	4 608 KBytes
64x64	18 432 KBytes	36 864 KBytes
128x128	147 456 KBytes	294 912 KBytes
256x256	1 179 648 KBytes	2 359 296 KBytes

We can observe how important is to save space on the device due to the exponential growth of the likelihoods table. On the prior files is less of a problem due to linear growth, and much smaller size in relation to the likelihoods file.

3.5 Evaluation Metrics

3.5.1 Precision

To compare the precision of the final results achieved in this work we will use the Kullback–Leibler divergence method between the arrays of the exact inference distributions and the approximate inference distribution, in order to understand the loss in precision when using the second technique.

The Kullback–Leibler divergence is a measure of mathematical statistics that tells us how two probabilistic distribution are different from a each other [20].

Since we will be comparing values between two vectors the definition that we'll be using is the discrete one, which goes as:

$$D_{KL}(P \parallel Q) = \sum_{x \in X} P(x) \log \left(\frac{P(x)}{Q(x)} \right) \tag{3.5}$$

This means that the divergence is the sum of the expectation times the logarithm of the exact value dividing by the aproximation. Thus the closer the approximation, the closer to zero the sum will be [21].

3.5.2 CPU Power Estimation

In the CPU subcategory, we will run the exact inference algorithm in ProBT toolkit (python) as well a more lightweight C++ approach for a more fair comparison to the GPU and FPGA algorithms.

Estimation of the power consumption of a CPU can be worked out through simple circuit theory. The CPU can be approximately modeled as a variable resistor which changes its resistance as the workload increases. Considering that the power supply to the CPU provides a constant voltage, this will draw proportionally more current as its workload increases.

Given this fact, can find power consumption with only 3 values, workload, and the CPU voltages, namely resting voltage and max voltage. Thus total energy consumed can be found multiplying the execution time of the algorithm by the cpu's voltage (estimated by the workload value):

$$E = V_{CPU} \times t \tag{3.6}$$

To find the execution time of the algorithm in both the python code and C++ we use functions that keep track of processor usage time, respectively time.clock() [22] and std::clock_t clock() [23].

For measuring workload we have access to the command "top" on terminal (in Linux) that give us a lot of information on OS and hardware usage including the workload in percentage for each core. The CPU voltage info is retrieved from the vendors specifications where the algorithm is run.

After collecting this measures we use a linear regression between the average resting CPU power consumption and maximum power consumption (without overclock), where the determinant factor is the CPU's frequency, this provides us a decent estimation of power used *per* unit of time [24].

3.5.3 GPU Power Estimation

For the GPU power estimations, a similar procedure will be followed, yet we have available a much more accurate tool to perform these said estimations. NVIDIA Management Library (NVML) is a C-base API for monitoring and managing various states of NVIDIA GPU devices. It provides a direct access to the queries and commands exposed via nvidia-smi.

With this probe it is periodically captured several measurements relevant to power consumption and resourse utilization:

power.draw	The last measured power draw for the entire board, in watts. Only available if power management is supported. This reading is accurate to within +/- 5 watts. Requires Inforom PWR object version 3.0 or higher or Kepler device.
clocks.gr	Current frequency of graphics (shader) clock.
clocks.sm	Current frequency of SM (Streaming Multiprocessor) clock.
utilization.gpu	Percent of time over the past sample period during which one or more kernels was executing on the GPU. The sample period may be between 1 second and 1/6 seconds depending on the product.
utilization.memory	Percent of time over the past sample period during which global (device) memory was being read or written. The sample period may be between 1 second and 1/6 second depending on the product.
clocks.mem	Current frequency of memory clock.

Table (3.1) Information captured on the GPU hardware during the Bayesian inference algorithm execution

3.5.4 FPGA Power Estimation

The FPGA's power consumption is estimated using the Power Analysis Tools provided by the Intel Quartus software. More specifically, the tool being used is the Prime Power Analyzer, which estimates power consumption for a post-fit design, allowing to establish guidelines for the power budget.

It is up to the user to provide timing assignments for all clocks in the design, or specify signal activity data for power analysis, specify the I/O standard on each device input and output, and the board trace model on each output in the design.

3.6 Framework and Hardware

On the dissertation we use the same material from the previous work in order to maintain fair comparisons [14]. The requirements in terms of power, airflow and size for the FPGA accelerator boards installation are identical to those of server grade GPUs, for which the case was designed. The system has 64GB of RAM, two Intel Xeon E5-2620 CPUs running at 2.6GHz, each with 12 cores, and CPUs running tightly interconnected through two Intel QuickPath Interconnect (QPI) interfaces, supporting 9.6GT/s transfer speeds each. The FPGA used is a ProceV [25] board from Altera's Stratix V family. The ProceV provides massive capacity (up to 952K LEs), and high memory and I/O performance, 8-Lane PCIe gen 3, twenty six 12.5/14.1 Gb/s transceivers provide external IOs of up to 366 Gb/s (full duplex). The memory is embedded with 8 TB/s throughput, 16 GB ECC DDR III and optional 288 Mb DDR II SRAM.

To test the algorithms in the GPUs and CPUs it is used a Intel Core i5-4570 [26] quad-core processor, 3.20 GHz and 6MB cache, requiring 84W TDP. The GPU is a Nvidia GTX Titan [27] with 2688 cores, 837 MHz of graphics clock, 6.0 Gbps of memory clock, and 250 Watts of maximum power.

To run the algorithms on GPU it was used Intel FPGA SDK for OpenCL 19.1, and 14.1 for the FPGA due to the ProceV board incompatibly with more recent versions.

Chapter 4

Experimental Results

4.1 Finding the Position of the Boat

Figure 4.1 shows the Bayesian inference engine running on the devices in real time, with the user controlling the boat position with a simple interface and getting its likelihood position as return with very low latency.

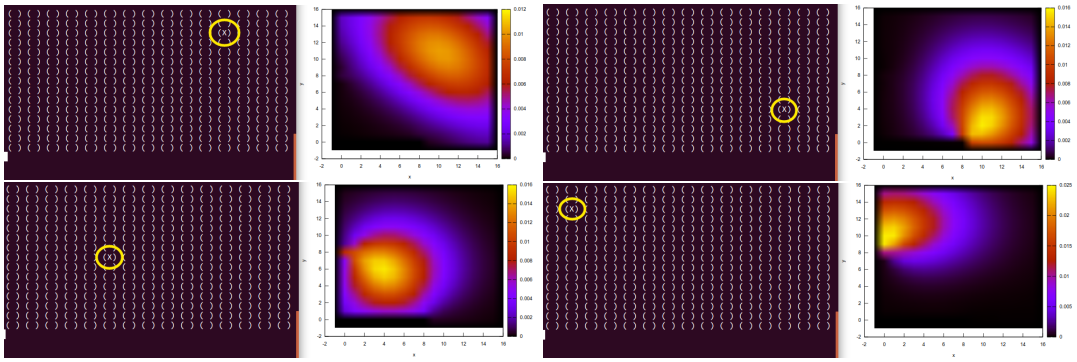


Figure (4.1) Finding the Positions of the Boat through Bayesian Inference.

In the following tests that will measure various metrics, such as transfer times, power consumption and executing times of the algorithm we will add a new dimension to the tests by performing the same algorithms different number of times each. This approach was used due to the relatively short running times the algorithm had on each device, even when comparing with just the transfer times of the tables from host to device and vice-versa. By extending running times, we can then have more precise power consumption

measures as well as more accurate executing times per iteration.

4.2 Scalability and Resource Usage

During the development of the project it was observed that (opposite to what was expected) the hardware "footprint" stopped growing after 32x32 grid size, with the following 64x64 and 128x128 hardware designs presenting the exactly same resource allocations as can be observed by the figures 4.3 and 4.4. Several attempts were made to discover the limiting factor to the increasing scale of the problem, but none was found since the compilation reports did not present physical max out.

This just means that a limited number of kernels are performing the work of several cells sequentially due to these constraints. Which leads us to assume that the results presented might be sub-optimal, and that probably there is space for improvement, either in execution time and power consumption.

Kernel Name	ALUTs	FFs	RAMs	DSPs
ComputePosteriorUnormalized4	32377	34913	311	24
Global Interconnect	1075	7592	36	0
Board Interface	62076	55847	259	0
Total	95528 (18%)	98352 (9%)	606 (24%)	24 (1%)
Available	524800	1049600	2567	1963

Figure (4.2) FPGA Resource utilization for 16x16 grid dimensions.

Kernel Name	ALUTs	FFs	RAMs	DSPs
ComputePosteriorUnormalized4	42390	46228	320	24
Global Interconnect	1392	5347	61	0
Board Interface	62076	55847	259	0
Total	105858 (20%)	107422 (10%)	640 (25%)	24 (1%)
Available	524800	1049600	2567	1963

Figure (4.3) FPGA Resource utilization for 32x32, 64x64, and 128x128 grid dimensions.

For the GPU, the utilization was always near the 50% and power consumption around 32% for all grid sizes when subjected to constant utilization. Due to the command "nvidia-smi" the limiting factor was very clear, since the access to global memory speed was constantly at 100%. This was not possible to replace or mitigate since the constant access to the likelihood lookup tables are the integral part of this algorithm, then the space for improvement might consist on a better choice of specific GPU to run these kind of algorithms.

4.3 Results Precision

As previously mentioned in the section 3.7, to check the results fidelity that came out of the devices, these were compared directly with the output of the model.

To achieve this, firstly we supply to the proBT model some coordinates (x,y) and perform one inference iteration, to a certain size of the model. After this, we proceed to do the same process on the programs developed in this dissertation, and compared this final matrices value by value using the Kullback-Leibler divergence.

$$D_{KL}(P \parallel Q) = \sum_{x \in X} P(x) \log \left(\frac{P(x)}{Q(x)} \right) \tag{4.1}$$

Since we're trying to obtain the results closest to the generated in the model, then in this equation $P(x)$ is the model and $Q(x)$ represents our results.

Table (4.1) Kullback-Leibler divergence for each grid size

Grid Size	Kullback-Liebler divergence
16x16	3.89257×10^{-16}
32x32	1.41173×10^{-15}
64x64	7.36625×10^{-16}
128x128	4.68538×10^{-15}

The table 4.1 represents Kullback-Leibler divergence using double floating point number program for the various grid sizes. The doubles were used in this case to compare with the standard result due to the output of the ProBT library was also stored in doubles. After comparing both matrices, we obtained the result of 3.82875×10^{-16}, which is virtually zero, and thus verifying the expected outcome. The reason to the divergence not being exactly zero, might be due to approximations that happen somewhere on any given operations. As expected it is indeed exact Bayesian inference being performed.

4.4 Execution Times

During the development and testing of the inference Kernels in the several devices, it was noticed that the vast majority of time and energy of the devices was being spent on the normalization of the posterior probability matrix. This was due to the significant overhead necessary to synchronize all the kernel threads to perform the several steps of this procedure.

This disproportional consumption of time and energy by the normalization operation in relation to the computation of the posterior probability was also noted on a previous work where both tasks were divided in two different kernels, in which 93.75% of time on the device was spent in normalizing the matrix [15]. Test runs performed on the GPU also show the similar gains when the normalization is passed to CPU.

After some testing it appeared that performing this final portion of the algorithm back on the host had a great reduction in time spent performing the total operation of Bayesian Inference, and also some reduction in total energy consumed (accounting for the energy consumed on CPU to perform that operation), these times are shown in table 4.2. This statement is true for small matrix sizes as the ones being discussed in this dissertation. For higher dimension sizes of matrices it is more time and energy economic to perform the normalization on devices since the overhead necessary to the kernels synchronization becomes more and more negligible in relation to said operation.

Table (4.2) Posterior Probability Matrix Normalization Time on CPU

16×16	32×32	64×64	128×128
1325ns	5326ns	21525ns	85884ns

Note that on the following tables of the Bayesian Inference times in GPU and FPGA, the values refer to the time spent only on the devices. To have a fair comparison with other approaches the time of the posterior probability matrix normalization time in CPU has to be added.

Table (4.3) Bayesian Inference in GPU

nº of Iterations	16×16	32×32	64×64	128×128
1	60.1μs	80.2μs	93.3μs	172.3μs
10	61.3μs	87.7μs	105μs	177.6μs
100	83.8μs	117.3μs	115.6μs	192.9μs
1000	108.8μs	129μs	188.2μs	366.5μs

Table (4.4) Bayesian Inference in a single FPGA

nº of Iterations	16×16	32×32	64×64	128×128
1	118μs	231μs	415μs	1361μs
10	145μs	265μs	436μs	1374μs
100	227μs	505μs	1346μs	4900μs
1000	853μs	3330μs	11990μs	47571μs

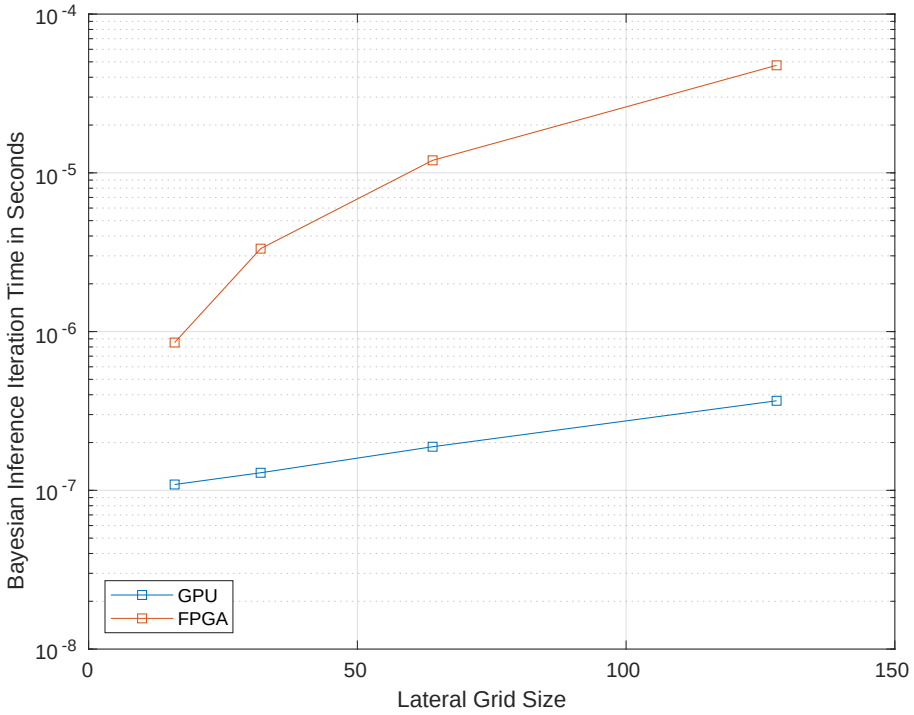


Figure (4.4) FPGA and GPU Execution Times for one Bayesian Inference Iteration on Device.

In the following tables we present the times obtained running exact inference on the ProBT library in python and optimized the C++ version.

Table (4.5) Bayesian Inference in CPU using ProBT version for one iteration

16×16	32×32	64×64	128×128
58.373ms	184.744ms	895.136s	5.998676s

Table (4.6) Bayesian Inference in CPU using Optimized C++ version

nº of Iterations	16×16	32×32	64×64	128×128
1	332.6μs	3061.3μs	22373.66μs	172216.2μs
10	1349μs	16116μs	241084.7μs	1.840625s
100	17628.4μs	203309.5μs	2.68634s	20.77593s
1000	1.622749s	18.622719s	102.94s	523.70552s

4.5 Power Consumption

Table (4.7) Avarege Energy input on Devices

	16×16	32×32	64×64	128×128
GPU	78.55 W	80.02 W	81.08 W	81.17 W
FPGA	12.584 W	12.564 W	12.65 W	12.618 W

The constant average energy consumption for problems of different dimensions as shown above indicates again the trouble found in scaling properly the algorithm on devices.

4.5.1 GPU Energy Consumption

Table (4.8) Power Consuption of One Iteration Bayesian Inference on GPU

	16×16	32×32	64×64	128×128
Bayesisan Inference (on Device)	0.008546 mJ	0.01032 mJ	0.015259 mJ	0.02975 mJ
Kernel Normalization (host)	0.1113 mJ	0.447384 mJ	1.8081 mJ	7.214256 mJ
Total	0.119846 mJ	0.457704 mJ	1.816259 mJ	7.244006 mJ

4.5.2 FPGA Energy Consumption

Table (4.9) Power Consuption of One Iteration Bayesian Inference on FPGA

	16×16	32×32	64×64	128×128
Bayesisan Inference (on Device)	0.010734 mJ	0.041839 mJ	0.151731	0.600271 mJ
Kernel Normalization (host)	0.1113 mJ	0.447384 mJ	1.8081 mJ	7.214256 mJ
Total	0.122034 mJ	0.489223 mJ	1.959831 mJ	7.814527 mJ

4.5.3 CPU Energy Consumption

Table (4.10) One Iteration Bayesian Inference in CPU using Optimized C++ version

16×16	32×32	64×64	128×128
11.3316 mJ	135.374 mJ	1.879 J	14.462 J

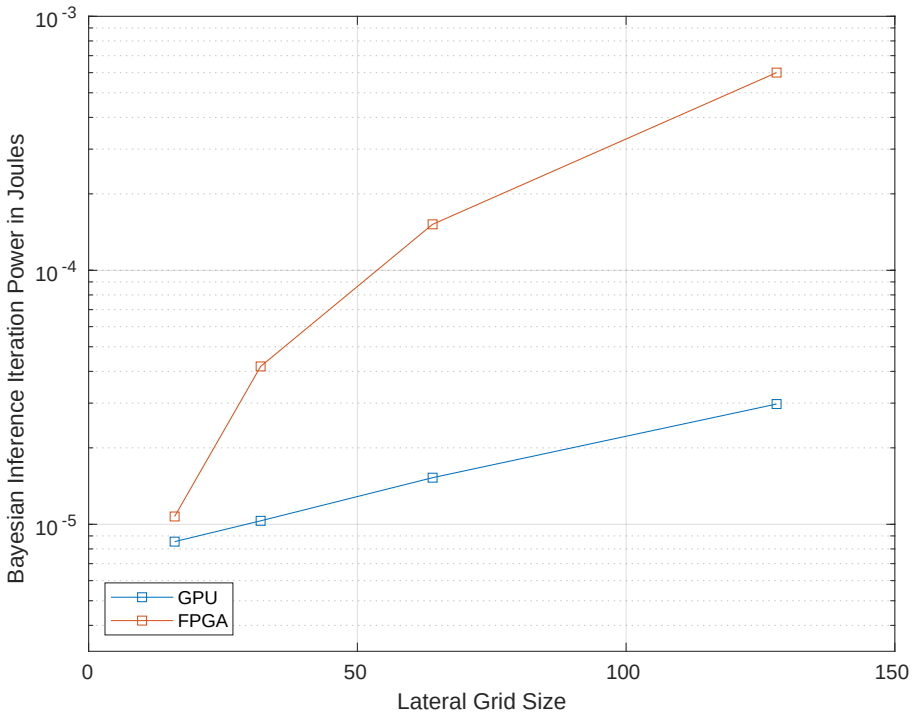


Figure (4.5) FPGA and GPU Power Consumption for one Bayesian Inference Iteration on Device.

In the tables of GPU and FPGA power consumption we can observe that the total energetic cost is very similar due to the normalization operation on host being by far the most substantial power drainer. On the other hand when comparing the operations on the device the OpenCL implementation applied on GPU has a clear energetic advantage due to the much faster execution times, even though it has almost 6.5 times larger average power input. In summary the FPGA implementation requires a smaller power supply, but with an overall higher energy consumption (and gets greater with the increase of problem dimensions), and higher latency in relation to the GPU implementation.

4.6 Computing with Double Precision vs Single Precision

Double vs Single Precision

The default programs developed were made using double precision variables, but another version was made for the GPU with single precision variables in order to compare performances.

On the single version of the program the Kullback-Leibler divergence result in relation to the output of the ProBT program was 2.84407×10^{-10} on a 16x16 grid size. Given this, we can observe this difference on the possible quotient size on the higher divergence shown.

This specific application does not require such precision provided on the double programs in order to maintain the results relevance or usability. So when developing a new program is important to reflect if it's worth the trade-off.

Transfer Times

The data presented in the following tables was obtained measuring the interval of time that took the host program to write in the GPU's memory all the information necessary to the execution of the program. This includes the likelihoods lookup table (that as we've seen previously, grows exponentially with the size of the problem), sensor values, priors vector, and finally control variables.

The times were recorded in for possible dimensions of the problem, 4, 5, 6, and 7 Bits, that represent grids of 16x16, 32x32, 64x64, and 128x128, respectively. To account for possible anomalies, each case was measured 5 times and the final line of the table represents the averaged value.

Table (4.11) Average Transfer Times to GPU

Table (4.12) Using Single

	16×16	32×32	64×64	128×128
Single	388.2μs	1616.4μs	7182.8μs	48155.6μs
Double	537μs	2225μs	13196μs	95522μ

As we can see from table 4.8, with the exponential increase of data being transferred to the device, the time consumed by the overhead processes becomes more negligible and the ratio between both transfer times tends to the ratio of the memory size being passed,

as expected.

Table (4.13) Ratio Between single and Double Transfer Times

16×16	32×32	64×64	128×128
0.7229	0.7264	0.54431	0.5041

Execution Times

To test the double vs single execution times, we counted the time since the kernels were launched until the host received the final data from the device. Important to note that these OpenCL kernels had no difference between them besides the variables it operated with.

The results were obtained testing the problem with 32x32 (5 bits) in dimension for several different iterations, respectively 1, 10, 100, and 1000. Due to relative stability of results obtained, only 5 samples were taken of each case.

Table (4.14) Avarege Execution Times on GPU

nº of Iterations	1	10	100	1000
Single	128.6μs	490.6μs	3688.8μs	37599.4μs
Double	141.4μs	528.6μs	3684.2μs	37424.8μs

From the fact that the results coming from the device with the kernels of the double variables having twice as much information being sent back, the prediction was that there would be some difference between the values in the times of both versions, yet this difference would have to stay constant throughout all the sets of iterations if they were to have the same execution times.

We can see those slight interval in the times of set of 1 and 10 iterations, of 12.8 μs and 38 μs respectively, yet in the last two sets, the double version of the kernels actually presents inferior times. This is due to the times measured had high intervals of variation.

From these values we can confirm that for inference algorithms no significant executing time difference exists for this commercial GPU, maybe the case that in other architectures results would vary between using single or double. As explained in the Background chapter, some vendors create specialized GPUs configurations for one type of variable. All this information should be taken into account when choosing the design approach.

Even though results were recorded while no other program (necessary for the execution of the experiment) was active, the values still show some fluctuation between the single and double approach. This may be explained by other extrinsic factors hard to isolate or control, such as GPU temperature, or other programs running concurrently (such as the OS) that the user has no control on.

4.7 Comparison With Previous Works

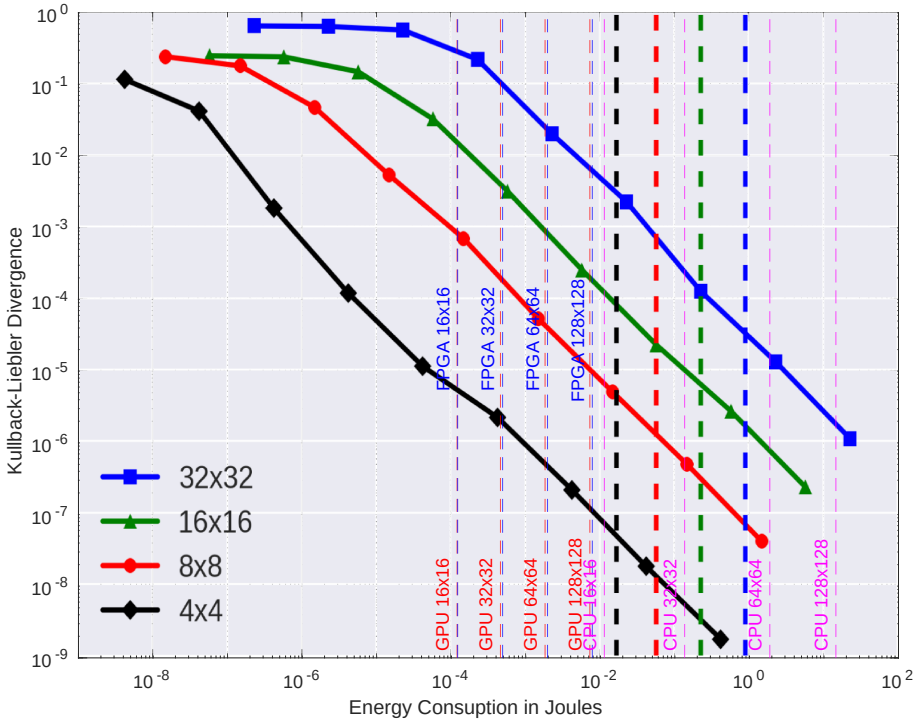


Figure (4.6) KL divergence results of boat example for different grid sizes and increasing energy consumption of the BM1 (background), OpenCL on FPGA (blue dotted) and GPU (red dotted), and finally our improved version of the CPU algorithm (pink dotted).

As it can be observed in figure 4.6 the background graph displayed corresponds to the diagram from the previous work on this field [10], that was fitted inside the dimensions of our work due to the individual point information of the latter could not be obtained at the time . The bold vertical lines indicate the energy used by a typical laptop computer to perform the corresponding exact inference on the old algorithm, the finer dotted lines corresponds to the data obtained during this dissertation. In this graph the OpenCL implementations on GPU and FPGA are occupying almost overlap due to the low granularity of this graph.

Despite the great improvement in precision with fairly similar power consumption, it is important to note that BM1 inference engine and the OpenCL kernels used in this work perform different operations. This work can achieve the exact inference by performing only one iteration, due to every single possible sensor's Bayesian Inference outcome being calculated *a priori*. Our method is itself already a trade off between memory and circuit space on the device. Both approaches present difficulties with increased scaling due to the exponential nature of the problem, BM1 with circuit space, and *a priori* calculations with memory on device, yet both represent a huge improvement in relation to desktop computations either in energy consumption and latency.

The preference for each system depends on the intended outcome, where for high precision results our method takes a decisive advantage in energy consumption either in FPGAs and GPUs, with the latter also presenting few scaling problems and great gains in executions times in all tested dimensions.

If precision is a less relevant factor, the BM1 engine might be a better suited choice due to the fact that enables a very selective precision by just regulating the number of inference iterations, and thus can lead to great energetic savings by relinquishing precision.

Although these inference problems if presented with large dimensions become difficult to scale on a single device, it is also important to note that the algorithm presented in this work can be easily divided by several devices, since each kernel is assigned one cell of the grid and each cell, only acesses one specific portion of the likelihoods table, this means the algorithm can be divided by cells and memory throughout several devices.

Another important factor to highlight is that the OpenCL implementation on GPU is faster to implement, compile and test in relation to the FPGA approach, as well it is possible allocate dynamically GPU resources during the execution of the algorithm. Due to this last reason, for the FPGA was needed to create, compile (several hours), and deploy

different codes to the FPGA device on each grid size tested. The GPU implementation also consistently performed with lower latency, with difference between the two approaches growing with increasing dimension sizes. As stated in the section 4.6 he FPGA program had the advantage of requiring less power to operate (approximately 6.5 times), but having a higher total energy consumption for the Bayesian inference operation (from 1.25 ate 16x16 size to 20 times on 128x128 grid size).

Also as expected even before the start of this dissertation the advantages of the OpenCL implementations in relation using BAMBI toolchain and Hardware Description Languages (HDL) have a much faster development cycle, less specialized development work, hardware flexibility allowing the development of an algorithm not specific to any particular hardware, and easy migration between devices.

Chapter 5

Conclusion

5.1 Result's Discussion

As stated in the previous chapter it is clear that no Bayesian inference implementation outmatches the other in all categories. The solutions pursued in this work lose to the BM1 engine where the main priority of an application is the low energetic consumption and have low precision requirements, although for higher grid dimensions, it seams that there would be less of a difference.

Exact Bayesian Inference on OpenCL using FPGA

Performing the exact Bayesian Inference on OpenCL using FPGA presents a advantage on the BM1 machine in terms of energetic consumption if the high precision is a requirement on the intended application, as well as presenting significantly less design effort (the same can be said for the OpenCL GPU implementation). The increased design, and energetic costs in relation to using GPU as hardware for exact Bayesian inference might be worth it in cases where possible power supplied might be reduced, such as is the circumstances of some robots and IoT devices.

Exact Bayesian Inference on OpenCL using GPU

The OpenCL implementation on the GPU presents significantly less latency the more the dimensions of the problem increase, as well a low energy consumption overall

making it a great candidate for solving Bayesian inference problems when considering the much reduced design and integration effort, especially in the developing and testing new Bayesian solutions. It is also important to highlight that it has the significant set back of being the approach that has the highest power requirement by far, making it an automatic disqualifier for several applications.

5.2 Hardware and Software Limitations

Throughout the development of the projects, several hardware and software issues and conflicts were found that had a significant impact on the development of these. The time taken to resolve, mitigate, or go around these problems severely delayed the finishing of the dissertation. The most prominent setbacks are the following:

The server's FPGA where the work was developed integrated Gidel's Boards to which their Support Package only supports version 14.1 of the ALTERA (now Intel) FPGA OpenCL SDK. This fact brought some legacy issues when operating in newer OSs, as well the impossibility of using more recent tools available that could have shorten the development times (e.g. Being able to use compilers that synthesize the FPGA's designs faster). Another problem found was also the incredible low support and documentation available for this outdated support package.

The full compilation times in FPGA's would take approximately 2 to 3 hours, making the process of debugging practical problems extremely slow. The emulation of these same programs only took a few minutes, enabling us to debug virtually every logic problem before deployment.

The commercial package Gidel's BSP has poor documentation support and includes various bugs on its use. Another problem related to this is that it being a commercial product, its source code is not available for correction. The most significant setback is that its use on the PCIe drivers is extremely unstable. Some common occurrences were

that any attempt to reconfigure one or several FPGAs that were configured not so long ago (a few hours) the host would freeze and force a reboot.

5.3 Future Work

In future works done in this field, probably the most significant goal would be to develop other benchmarks to test the performance of Bayesian inference algorithms, due to the fact that a single test can not be representative of all kinds of Bayesian inference applications.
Generalize and automate the toolchain in order to accept a broader range of Bayesian Inference problems, as well as integrate the toolchain with ROS (Robotic Operating System) to allow testing in real world conditions.

Another interesting goal would be to create a Graphical User Interface (GUI). Should be relatively easy to adapt inputs of the toolchain and the Python Master script execution to such an interface.

Finally to study further the limitations on the scaling of the OpenCL implementations.

Bibliography

[1] Schacter, Daniel (2011). *Psychology*. Worth Publishers.

[2] Bernstein, Douglas A. (5 March 2010). *Essentials of Psychology*. Cengage Learning. pp. 123–124. ISBN 978-0-495-90693-3. Archived from the original on 2 January 2017. Retrieved 25 March 2011.

[3] S. Legg; M. Hutter (2007). *A Collection of Definitions of Intelligence*. 157. pp. 17–24. ISBN 9781586037581.

[4] Bengio, Yoshua; LeCun, Yann; Hinton, Geoffrey (2015). *Deep Learning*. Nature. 521 (7553): 436–444. Bibcode:2015Natur.521..436L. doi:10.1038/nature14539. PMID 26017442

[5] Malcolm Tatum, October 3, 2012. *What is Machine Perception*.

[6] Brassard, G. and P. Bratley: Algorithmics - Theory and Practice, Prentice-Hall, 1988. (Chapter 8)

[7] Harel, D.: Algorithmics - The Spirit of Computing, 2nd Edition, Addison-Wesley, 1992. (Chapter 11)

[8] *Bayes' Theorem*. Stanford Encyclopedia of Philosophy. Plato.stanford.edu. Retrieved 2014-01-05

[9] Dr Jacques Droulez. *Bottom-up Approaches to Machines dedicated to Bayesian Inference*. Period covered: from 01/01/2014 to 31/12/2016, http://www.bambi-fet.eu.

[10] Alexandre Coninx, Pierre Bessière, Emmanuel Mazer, Jacques Droulez, Raphaël Laurent, et al.. *Bayesian Sensor Fusion with Fast and Low Power Stochastic Circuits*. IEEE International Conference on rebooting Computing, Oct 2016, San Diego, United States. hal-01374910

[11] J. F. Ferreira, P. Lanillos, and J. Dias, *Fast Exact Bayesian Inference for HighDimensional Models*. 2015.

[12] P. BAMBI, *BAMBI D3.10 : Emulation of a probabilistic computer on current recon-*

figurable logic. tech. rep., 2017

[13] A. Coninx, P. Bessi'ere, E. Mazer, J. Droulez, R. Laurent, M. A. Aslam, and J. Lobo, *Bayesian sensor fusion with fast and low power stochastic circuits* 2016 IEEE International Conference on Rebooting Computing, ICRC 2016 - Conference Proceedings, 2016.

[14] M. G. D. Mira, *Using an FPGA Mini-Cluster to Implement Bayesian Application-Specific Integrated Circuits for Robotic Applications*. 2017.

[15] José Carlos Direito May 2018, *Probabilistic Computing Using OpenCL on an FPGA Mini-cluster*, Dissertation submitted to the Electrical and Computer Engineering Department of the Faculty of Science and Technology of the University of Coimbra in partial fulfillment of the requirements for the Degree of Master of Computer Science.

[16] *FPGA Architecture for the Challenge*. toronto.edu. University of Toronto.

[17] K. Mekhnacha, J.-M. Ahuactzin, P. Bessière, E. Mazer and L. Smail. *A unifying framework for exact and approximate Bayesian inference*. January 2006. Institut National De Recherche en Informatique Et Automatique. And Retrived from the URL: https://hal.inria.fr/inria-00070226/document

[18] Janet Morss *FPGAs vs. GPUs: A Tale of Two Accelerators* January 16th, 2019. Retrived from URL: https://blog.dellemc.com/en-us/fpgas-vs-gpus-tale-two-accelerators/. Retrived Outober 17th, 2019.

[19] *The open standard for parallel programming of heterogeneous systems*, Retrived from URL: https://www.khronos.org/opencl/ at 26/02/2020

[20] Kullback, S.; Leibler, R.A. (1951). *On information and sufficiency*. Annals of Mathematical Statistics. 22 (1): 79–86. MR 39968. doi:10.1214/aoms/1177729694

[21] Kullback, S. (1959), *Information Theory and Statistics*, John Wiley Sons. Republished by Dover Publications in 1968; reprinted in 1978: ISBN 0-8446-5625-9.

[22] Janet Morss *Measure Time in Python – time.time() vs time.clock()*. Last Updated: Friday 3rd May 2013. Retrived from URL: https://www.pythoncentral.io/measure-time-in-python-time-time-vs-time-clock/. Retrived December 8th, 2019.

[23] Janet Morss *std::clock* Retrived from URL: https://en.cppreference.com/w/cpp/chrono/c/clock. Retrived December 8th, 2019.

[24] Matthew Travers 2015, *CPU Power Consumption Experiments and Results Analysis of Intel i7-4820K* , Systems Research Group School of Electrical and Electronic Engineering. Technical Report Series NCL-EEE-MICRO-TR-2015-197. Chapter 2.3.

[25] *ProceV-GX- PCIe based 100G+ ultra-low latency platform.* https://www.intel.com/content/www/us/en/programmable/solutions/partners/partner-profile/gidel–inc-/board/procev-gx–pcie-based-100g–ultra-low-latency-platform.html. Retrieved 2020-02-28.

[26] *Intel® Core™ i5-4570 Processor*. https://ark.intel.com/content/www/us/en/ark/products/75043/intel-core-i5-4570-processor-6m-cache-up-to-3-60-ghz.html. Retrieved 2020-02-28.

[27] *GeForce GTX TITAN — Specifications.*https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan/specifications. Retrieved 2020-02-28.