Facultad
Informática
Universidad
Murcia

University of Murcia
Computer Science Faculty

MSc on New Technologies in Computer Science

Towards a unified programming model for diverse computing architectures: Experiences using PHAST

Author:
Eduardo José Gómez-Hernández
{eduardojose.gomez@um.es}
Advisor:
José Manuel García
{jmgarcia@um.es}
June, 2019

This page has been intentionally left blank

Declaration of Authorship

I, Eduardo José Gómez Hernández, declare that this master's thesis, titled "Towards a unified programming model for diverse computing architectures: Experiences using PHAST"**, and the work presented in it are mine.**

I confirm that:

- **This work was done wholly or mainly while in candidature for a master's degree at this University.**
- **Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.**
- **Where I have consulted the published work of others, this is always clearly attributed.**
- **Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this dissertation is entirely my own work.**

_____

Eduardo José Gómez Hernández
Murcia, June 2019

Contents

Abstract

We are in an era that is reaching the limit of the multi- and many-core performance. Since the start of computing, the performance was always a limit to break, in the first place by scaling the frequency of each CPU generation, then Moore's Law forcing the increase of the transistors in an integrated circuit.

Nowadays, many tasks are delegated to specific devices capable to execute those tasks faster and more efficient than a traditional CPU, being the most known one the GPU. We have reached an accelerator boom, with the usage of FPGAs as accelerators instead of only prototyping and the creation of many ASICs for specific tasks, there are a vast of accelerators that can be used.

The problem of this accelerator boom is that to exploit the performance of each one, it is necessary to use its own programming language, a DSL. Therefore, to use multiple devices, programmers are forced to create multiple versions of the same source code for each device, increasing a lot the code complexity.

Machine learning is one of the fields where new accelerators are developed each day because being able to run more complex models in less time allows them to solve bigger problems.

To accomplish this big problem of multiple device programming, several programming models are developed. Therefore, we want to know which programming models are available, which is the main focus of each one, and their characteristics. The is not any complete classification of these high performance programming models, only an interesting survey from 2013, but many of the models classified are now deprecated.

We have developed a classification of 23 high performance models using 4 parameters, supported platforms, type of programming model, the programming paradigm and the memory paradigm. These models were selected from 50 reviewed models, but we discard discontinued models and the ones that have been merged into other models.

From this classification, we have extracted 8 models that can be fit into a single source programming model classification, it is only a sample of all models in the table that have this characteristic.

Then, we think that use one of these new models into a real application is interesting, therefore we selected PHAST, and as a joint collaboration with its creators, we have been porting the Caffe deep learning framework to PHAST.

In this work, we show the status of this ongoing project that we plan to finish and release freely. Using the Caffe CPU only code, we aim to port it using the performance portability approach, it means that one source code can be targeted to multiple devices. We have obtained to run two LeNet based networks for MNIST and CIFAR fully in PHAST, and passing almost all Caffe tests despite the unimplemented functionality.

We are not ready to give performance results, but our firsts tests give about 10x of performance loss, but with a few changes we got 2x improvement, and this can be greatly improved.

We continue working in this project, explore the remainder programming models is essential to complete our classification and finish the Caffe port extending it to other devices and getting better performance.

Resumen

En el inicio de la computación, aumentar el rendimiento ha sido la meta a conseguir. En el inicio, incrementar la frecuencia de la CPUs en cada generación era la clave. Adicionalmente incluyendo más transistores, creando la conocida Ley de Moore, que indicaba que cada 2 años se doblaba la cantidad de transistores en un circuito integrado. Así ha sido hasta principios del año 2000, donde los procesadores estaban alcanzando el límite de densidad energética. Con un cambio de paradigma, la creación de los procesadores multinúcleo, conseguimos seguir incrementando el rendimiento.

Durante este periodo de crecimiento de la capacidad de cómputo, distintos modelos de inteligencia artificial se iban creando, siendo las ANNs el algoritmo más conocido. Sin embargo, el uso de estos algoritmos conlleva una gran cantidad de tiempo, por lo que a mayor potencia de cómputo disponible, redes neuronales más complejas pueden ser desarrolladas, y de esta manera resolver problemas mayores.

A pesar de que aún seguimos incrementando el rendimiento de los procesadores multinúcleo, llevamos usando aceleradores hardware para ciertas tareas desde hace varios años. En un principio, los aceleradores más comunes, las GPUs, empezaron a ser usadas para cómputo de proposito general en lugar de ser exclusivamente para gráficos, pudiendo explotar así su paralelismo con muchos núcleos pero muy simples.

En esta revolución de aceleradores, empezamos a usar las FPGAs como hardware reconfigurable en tiempo de ejecución, en lugar de solo ser usadas como prototipado. A su vez, existen los ASICs, hardware específico aún más potente y eficiente que las FPGAs, siendo el ejemplo más conocido los minadores de bitcoins, ASICs desarrollados para hacer la función SHA256.

Con la gran cantidad de aceleradores disponibles, resulta extremadamente tedioso desarrollar aplicaciones capaz de usar varios aceleradores, ya que cada uno dispone de un lenguaje de programación propio denominado DSL. Para poder extraer el potencial disponible de un dispositivo hardware, resulta necesario el uso de su DSL. Por lo tanto, una aplicación que quiera extraer toda la potencia de cómputo que existe actualmente en los clústers heterogéneos, tiene que realizar distintas versiones del mismo código fuente, una para cada dispositivo que desee utilizar, incrementando radicalmente la complejidad del código.

Para facilitar el desarrollo de estas aplicaciones, existen los modelos de programación de alto rendimiento, los cuales facilitan la programación en entornos de alto rendimiento. Sin embargo, existen una gran cantidad de estos, y no hay ninguna clasificación actualizada de los modelos existentes. A pesar de esto, hay un pequeño estudio de 2013, sin embargo la mayoría de los modelos clasificados están en desuso.

A partir de esto, hemos desarrollado una clasificación de 23 modelos de programación de alto rendimiento de 50 que hemos revisado, siendo descartados la mayoría por estar descontinuados o haber sido mezclados con otros modelos ya existentes. Pero para saber las características principales que debe tener la clasificación, hemos empezado por coger varios modelos conocidos, y hemos examinado sus características para ver cuales son los puntos importantes a revisar. Entre estos modelos hemos elegido a OpenMP, OpenACC, OmpSs, OpenCL, SYCL, Kokkos, y PHAST. De esta selección, hemos concluido que una posible clasificación estaría basada en 4 características: las plataformas soportadas, el tipo de modelo, el paradigma de programación, y el paradigma de memoria.

De esa clasificación de 23 modelos, hemos extraído 8 que cumplen la característica de ser portables entre dispositivos, hay varios más, pero esto solo es una pequeña muestra. En estos 8 hemos observado los dispositivos que soportan, y los cambios que se deben realizar para usar el mismo código en otra plataforma.

Con toda esta información, hemos elegido PHAST para realizar este proyecto, en el cual hemos escogido el framework de deep learning Caffe, y hemos portado la funcionalidad del framework para ser portable entre plataformas. Este trabajo lo hemos realizado en colaboración los creadores de PHAST, pudiendo obtener así acceso temprano a la librería y teniendo su soporte durante el desarrollo.

En primer lugar Caffe tiene dos versiones del código fuente, una para CPU y otra para GPU, hemos eliminado el código de GPU para usar el CPU como base durante el desarrollo. Posteriormente hemos

observado el funcionamiento de Caffe como bloques, y hemos decidido portar la funcionalidad mínima necesaria para ejecutar dos redes basadas en LeNet disponibles como ejemplos de Caffe, una para la base de datos de MNIST y otra para CIFAR. Los bloques necesarios a portar son: Blob, Convolution, Pooling, InnerProduct, ReLU, SoftMax, SoftMax with Loss, y Accuracy.

Durante el trabajo, hemos usado la versión 1.0.1 de PHAST con GCC 6.30 y Cuda 9.0, usando como base Caffe obtenido del repositorio oficial en github. Con esta configuración, y un a máquina con dos Intel Xeon CPU E5-2603 v3 @ 1.60 GHz y una Geforce GTX 1080 8GB, hemos conseguido ejecutar satisfactoriamente ambas radas para CPU y GPU. Siendo el único cambio necesario el uso de un makefile u otro dependiendo de la plataforma deseada.

Adicionalmente, hemos decidido ejecutar los tests de Caffe para comprobar el estado actual de nuestro proyecto respecto a la implementación original, y a excepción de la funcionalidad no implementada, pasamos todos los tests. Y al ser un proyecto todavía en desarrollo, no tenemos resultados de rendimiento válidos, pero como primer vistazo, obtuvimos un 10x de perdida de rendimiento, que mejoró un 2x tras una pequeña revisión del código. Sin embargo, los casos de prueba son demasiado pequeños como para tenerlos en cuenta, y estamos convencidos de que pueden ser mejorados modernizando nuestro código modernizado.

Tras este trabajo, pensamos que nuestra clasificación puede ser usada como base para seguir clasificando el resto de modelos existentes, cosa que se queda como trabajo futuro. Y entre estos trabajos futuros también se encuentran pendientes terminar la adaptación de Caffe a PHAST, extender el trabajo a múltiples dispositivos, y mejorar el rendimiento obtenido.

List of Figures

List of Tables

Listings

This page has been intentionally left blank

1. Introduction

Since the start of computing, the performance was always a limit to break. In the beginning, the way to break this limit started by increasing the frequency of each CPU generation. Additionally, CPUs started to include more and more transistors. This worked for a lot of time, creating the Moore's Law (every two years, the transistors in dense integrated circuit was doubled). But, near to the early 2000s, processors were very near to reach its limit in power density [1]. Changing to another paradigm was necessary to continue improving the performance, the creation of multi-cores.

After all the new unlocked performance for CPUs, there are several things that other hardware devices are able to do faster and more efficient. In instance, real-time graphics processing continue to be done by specific accelerators (GPUs). These devices are developed with different paradigms in mind, and normally to be very efficient in only one task. GPUs have many more cores than a CPU, but they are simpler.

Another well-known accelerator is the Field Programmable Gate Array (FPGA), created in the beginning as a hardware prototyping device. Nowadays, they are also used in many other tasks like a reprogrammable accelerator.

Low power high-performance devices are highly demanded, and specific architectures builtin inside an Application-Specific Integrated Circuit (ASIC) are getting a lot of attention. Many custom architectures are being developed for Neural Networks, Physics, Simulations, etc.

These accelerator devices have shown that CPUs are not always the best option to solve a problem. As the multi-core era is ending, Domain Specific Architectures (DSAs) are getting the testimony [2].

On the other hand, meanwhile CPUs started to get relevant performance, machine learning fired. One of the most powerful machine learning techniques is Deep Neural Networks (DNNs), but their algorithms required a lot of computing time. Until now, a vast machine learning algorithms have been developed and improved.

Once the technology was able to process DNN algorithms in reasonable times, they have become more and more ubiquitous in everybody's daily life. Image classification, language processing, economics, medicine, video games, among others, are some of the fields where DNNs are being used. They are trying to improve, but requiring more time, and accelerators are used to reduce the amount of time needed.

In this new era of accelerators, the creation of a programming model able to use all this kind of new accelerators is extremely important [3]. This new approach is called performance portability [4], and it is not yet completely solved.

In this work, we have focused on the following objectives.

- **Search for relevant programming models for multiple device programming**: There are a vast of programming models for multiple devices, therefore we want to find and classify them.
- **Learn to use a high performance programming model**: Like any other programming paradigm, these programming models have their own way to be used. Being able to use one of them could not be an easy task.
- **Apply the previously selected model to a real application**: A new programming model can be awesome, but if it is not used, it has meaningless.

The ultimate goal of this work is to have a programming language that allows that one source code can be run in CPU or GPU with only recompiling it with the right options, and in addition, that could be targeted to other devices with no changes (in the source code). When it is finished, we will release it to allow everyone to use it freely.

2. Background

2.1 CPU Revolution

The multi-core revolution, after reaching the power density limit in traditional processors, allowed the creation of a new paradigm where the increase of throughput is based in the parallel execution of multiple independent workloads. Most of the computer power in data centers was brought by mono- and multi-core CPUs.

From some time ago, some researchers, most of them in computer vision, noticed that the GPUs, that they were already using, were able to solve their problems faster. At the begining, they started using programming languages for graphics, like OpenGL to implement some algoriths [5].

2.2 Accelerator Revolution

2.2.1 GPU & GPGPU

The most known accelerator is the GPU, able to generate graphics and compute data at high speed exploiting its Single Instruction Multiple Threads (SIMT) execution model, based in Single Instruction Multiple Data (SIMD) but with multithreading. Nowadays, GPUs are used like CPUs (GPGPU), we have specific languages to use them such as Cuda, or OpenCL. It is probably the most used accelerator.

2.2.2 FPGA

A FPGA is a configurable integrated circuit, built from an array of logic blocks and routing channels [6]. Each block is able to represent a little mathematical function, and connecting many of them, it is possible to build very complex devices.

In the beginning, it was mainly used for prototyping, but now it is also used as accelerators. There will no be as fast as a ASIC, but they can be reconfigured and can be used to implement any part of a program.

2.2.3 ASIC

To be faster than a FPGA and more power efficient, ASIC is developed and use in specific tasks. We have several ASICs currently in our devices. From the lowest computer expensive ones like the usb or keyboard controllers, to the bitcoin miners, which are ASICs developed to make the SHA256 function.

2.2.4 The problem

There are a vast of accelerators ready to be used and most of them are problem specific. Each one of these accelerators use its own programming language, a Domain Specific Language (DSL). Using the specific DSL for each device allows to benefit from the specific features of that device [2].

The problem with this accelerator boom is that each DSL is quite different. Therefore, making a portable software between accelerators is impossible without having multiple source files (at least one file per target device). Maintaining software with multiple source code files for the same functionality in different platforms is a very tedious task, greatly increasing the code complexity [7].

2.3 ML & DNNs

2.3.1 Development and usage

Theoretical and mathematical models of the artificial intelligence techniques were developed in the twentieth century, but the lack of computing power prevented it from progressing. Artificial Neural Networks (ANNs) are a type of brain-inspired learning algorithm, built from small units called neurons, being the perceptron the most common one. With the recent advances, deep neural networks have

been incorporated into numerous fields, such as image classification, language processing, economics, video games, and medicine. In some tasks, this new technology is being able to outperform human performance.

Medical image analysis has started to implement deep learning for screening and localization of malignant zones. Additionally, other medical areas are working with these kind of techniques as well, like the analysis of the genetic information inside DNA and RNA series [8]. The common objective is not replace physicians with deep learning techniques but supporting them to make better diagnoses.

2.3.2 Evolution with HPC

Machine learning techniques could be difficult to code and debug, therefore many frameworks have been developed to ease its use. Most of them are open source with software for most of the types of neural networks. The most known ones are Caffe, Caffe2, Tensorflow, Theano, PyTorch, Mxnet, and CNTK among others [9]. And there are frameworks like Keras, providing a more high-level experience, running on top of some of the aforementioned frameworks.

Specifically, the training phase is very time-consuming, since it is evaluating an optimization problem with hundreds, thousands, or even millions of parameters. Therefore, the reduction of the training phase execution time is a desirable feature for all frameworks. Thanks to this shorter training time, scientists using these frameworks can explore a wide solution space, and even develop more complex networks.

Therefore, most frameworks are able to use Nvidia's cuDNN, BLAS-like libraries, MKL-libraries, OpenCL, even specific libraries for custom integrated circuits, to speed up this computation. Also, many of them also allow other kind of parallelism for multiple nodes (using the MPI library).

3. An overview of high performance parallel programming models

3.1 Performance Scaling Problems

A High Performance Computing (HPC) cluster can run a vast of different jobs, each one specialized in different kinds of applications. Tradicional clusters had many difficulties in scaling the applications they had to run. One of the mitigations taken was the use of heterogeneous clusters instead of the traditional ones. They are more flexible, allowing the use of the specific resources to fit the application better.

Now, we have several different architectures together to be used by the applications, but this is not as easy as it may appear. An application needs to have the code to be run in each one of this architectures to be able to use the cluster efficiently.

3.2 High Performance Programming Models

There are many things to keep in mind to be able to program an HPC cluster efficiently, such as dependencies, communications, locks, possible failures, without forgetting multiple versions of the same code for each available architecture.

To address this problem, there are many programming models to help developers writing efficient code. In instance, OpenMP tackles multi-core but locally in the same machine, and MPI tackles the communication between nodes. But, these are low-level approaches, there are others with a higher abstraction level.

We want to know which programming models are available, which is the main focus of each one, and their characteristics, such as the programming paradigm or the memory abstraction model. But only partial classification have appeared. There is a interesting survey from 2013 [10], but many of those models are now deprecated. Therefore, we start to classify some of them, but to know which features are important, we start by selecting a few and looking for their main details.

3.3 Some High Performance Programming Models

This first selection of programming models is done with the idea of having at least one well-known programming model with different approaches. But, also with two very similar approaches to be able to find their differences. We think that we are able to find the main features of the programming models to classify them.

3.3.1 OpenMP

OpenMP is a specification for a set of compiler directives, library routines, and environment variable that can be used to specify high-level parallelism in Fortran and C/C++ programs [11]. These directives are also known as pragmas, they give information to the compiler about what to do with a statement. And, as it is only an Application Programming Interface (API), compilers have the responsibility of implementing it.

Nowadays, OpenMP has improved a lot; from being able to parallelize loops to currently use devices as accelerators. The API has an offload option, allowing the usage of some accelerators compatible with the implementation. For instance, the Intel Compiler supported the Xeon Phi KNC Coprocessor as an offloading target. But now, a lot of compilers are making a lot of effort trying to accomplish the GPU support using offload.

Depending on the point of view, it is possible to refer to OpenMP as a kernel or a task programming model. However, the main feature of OpenMP is the use of directives to annotate the source code in an elegant and efficient way. It is focused on shared memory environments using multi- and many-core CPUs.

3.3.2 OpenACC

OpenACC is a specification for a set of compiler directives performance-portable parallel programming model designed for scientists interested in porting their codes to a wide variety of heterogeneous HPC hardware platforms and architectures [12]. Similar to OpenMP, it is only an API which needs to be supported by compilers.

Like OpenMP, it is a directive-based programming model but focused in heterogeneous hardware, which means that OpenACC is expecting to be run from a CPU to an accelerator device, like another kind of CPU or a GPU.

3.3.3 OmpSs

OmpSs is a programming model developed at Barcelona Supercomputing Center (BSC) with the objective of merging StarSs with OpenMP as a way of extending its directives to support asynchronous parallelism and heterogeneity [13]. Nowadays, it supports Intel CPUs, Nvidia GPUs, Intel Xeon Phi KNC (native and offload), IBM Power8, ARM CPUs, and OpenCL compatible MALI GPUs. Also, it is evolving into OmpSs-2, a new programming model with a different approach.

OmpSs is builtin in the Mercurium Compiler and executed by the Nanos++ runtime system. Also, similar to OpenMP and OpenACC, it is a directive-based programming model, but in this case, tasks are the main focus.

3.3.4 OpenCL & SYCL

OpenCL has a very different approach to OpenMP, it is a kernel based programming model, and it is nearer to a language instead of an API. It is an open standard originally developed by Apple, but currently maintained by Kronos [14]. OpenCL depends on a Installable Client Driver (ICD), which manages all of the OpenCL calls. First, a loader retrieves all the available ICDs. After preparing the context of the execution, kernels get compiled to that specific platform at runtime and executed through the ICD.

SYCL is a specification over OpenCL to enable single source C++ programming to OpenCL, and Khronos expects to be a step forward in standards convergence. SYCL is very similar to OpenCL, but with many features from new C++ standards (11, 14, and 17) and enabling the C++17 parallel Standar Template Library (STL).

3.3.5 Kokkos

With a totally different approach, Kokkos is one of the modernist approaches available. It is a new C++ library with a lot of ideas from STL containers and algorithms. The main objective of Kokkos is to maximize the amount of user code that can be compiled for diverse devices and obtain the same performance as a variant code specifically written for that device [15].

It defines a new template type called View, that is used to allocate the data that will be used later. In this View type, it is possible to build an n-dimensional structure to be used as a container to be modified by the functions.

Kokkos allows the creation of new functions to modify the data stored in Views, but it needs to be encapsulated into a class with some restrictions and with the main entry point made with the operator '()'. This is a functor, and it is a concept borrowed from the C++ STL. These functors are run from special functions with a similar nomenclature as the algorithms library in C++ STL.

3.3.6 PHAST

Continuing with the previous idea, Parallel Heterogeneous-Architecture STL-like Template (PHAST) library [16–19] is a modern C++ programming template library based on the classic STL containers with the performance portability philosophy. It currently supports multi-core CPUs and Nvidia GPUs,

allowing the users to write expressive and concise sequential-like code that can be automatically parallelized. Its main goal is to let the programmers code using high-level programming approaches without preventing them from applying low-level optimizations, keeping the main code at a higher level of abstraction.

Similar to STL containers, PHAST provides a vector container but extending it with matrix, cube, and grid, all of them working with a very similar interface. Also having vectors like primitives similar to an SSE or AVX vectors.

These containers can be modified using functors. Similar to Kokkos, the idea of a functor is borrowed from STL as a struct that inherits from a base functor struct and at least has the operator '()' defined. There are a lot of algorithms and factors predefined, but this allows the creation of functionality that is not defined by default in the library, without losing performance or portability.

In its roadmap, there is planned support for multiple devices in the same executable, lambda syntax, FPGA support, OpenCL backend, multi-GPU, and many other interesting features.

After writing the sequencial code in the PHAST way, it is possible to change the device target changing a macro and the compiler. Therefore, having two different makefiles makes the trick. The most important thing is that the code has not changed, only the compiling process.

3.4 Classification

With this little overview of some programming models, we can proceed to classify them. But, first, it is necessary to specify which features we will be looking for, and a definition of it. There are some terms that could be misunderstandings or misinterpreted, and in this way, there will not be any kind of confusion.

3.4.1 Platforms

With the accelerators revolutions, there are a lot of devices that could be interesting, but in this case, we focus on CPU, GPU, and FPGA, without paying attention on any specific manufacturer.

3.4.2 Type

Depending on the requirements of modifying existing compiling architectures, we have defined 3 types of programming models:

- **Language**: A new set of primitives using a new compiler or a runtime system to run, such as Cuda or OpenCL.
- **Extension**: A modification to the compiler to add a set of directives or even primitives to be used in an existing language. These modifications may require a runtime system to run, but the original language is barely modified. An example of this could be OpenMP or AllScale.
- **Library**: Without any kind of modification to the existing compilers, it adds a set of primitives to be used in a program using the language specification. It could be a shared library or even a set of files to add to the compiling state. MPI, Phast, and TBB are good examples of a library.

3.4.3 Programming Paradigm

Each high-performance programming model has its own type of programming paradigm, i.e., how it is intended to be used. Most of the programming models can be classified into multiple programming paradigms. These are the programming paradigm we have taken into account:

- **Kernel**: Time-expensive parts are encapsulated into a special section that can be compiled apart, and it can be executed by other parts of the code. For instance, OpenCL creates kernels that are compiled in runtime and executed in the target platform.

- **Task**: The program is split into several tasks and they are run following the dependency graph. This can be very confusing because it is possible to use a kernel programming model as a task one, but it is not the same paradigm.
- **Directive**: Using a reduced set of primitives, it sets to the compiler what to do with a statement. It could be achieved with a set of functions or a set of pragmas.
- **C++ Template**: With the creation of C++11 and C++14 standard, the interest in the STL grew, new containers and the algorithm library extended its usage. In this paradigm, the data is stored in a set of containers, and only are modified using the functions available by an algorithm-like library, that usually allows to extend this set of functions. The functions executed by the algorithm-like library could seem like kernels, but the concept and data management is very different.
- **Skeleton**: The program has to be mapped to use a set of generic components with a specific pattern of computation and data dependence. For example, a dot product can be implemented as a MapReduce of multiplication and addition.
- **Wrapper**: In a wrapper programming model, it inherits the underlying programming paradigm, but encapsulating the functionality.
- **Threads**: This is not a programming paradigm by itself, as depending on the usage of the created threads it could be used as any paradigm.

3.4.4 Memory Paradigm

Memory paradigms are a bit tricky. Some of them are very similar, but the high-level concept is different, therefore we have split them into 6 categories.

- **Hierarchical**: Implicitly or explicitly copied, a hierarchical memory paradigm is shown when there are a host and at least one slave that is going to exchange information. It is normally associated with a CPU host and a device (like a GPU).
- **Shared**: In a shared memory environment, at any moment is possible to read or write at any part of the memory. Excluding conflicts between threads, devices, or compute units, all the memory is visible.
- **Implicit**: All memory transactions are handled by the programming model, without the involvement of the program. In some cases, the programming model allows to explicitly manage this memory, but it is not necessary.
- **Explicit**: This is exactly the opposite of implicit memory. The programmer must handle all memory transactions. In some cases, some of these transactions can be implicit if they are simple enough.
- **PGAS**: Partitioned Global Address Space (PGAS) assumes that there is a global memory address space, but it is logically partitioned to each thread, device, or computer unit.
- **Distributed**: Each thread, device, or compute unit has its own complete memory. Distributed memory is able to exchange data, but it is not necessary. In some cases, it is very similar to the explicit memory paradigm and hierarchical.

3.4.5 Table

With all this information, we have selected a total of 23 high-performance programming models from about 50 reviewed programming models to be classified in Table 1. Many of the discarded models are discontinued or have been merged into another one.

			Programing Paradigm							Memory Paradigm					
			Kernel	Task	Directive	C++ Template	Skeleton	Wrapper	Threads	Hierarchical	Shared	Implicit	Explicit	PGAS	Distributed
CPU	Type	Language	OpenCL	Cilk			Lift			OpenCL	Cilk	Lift			
		Extension		Cilk AllScale	OpenMP OpenAcc OmpSs					OpenMP	Cilk OpenMP	AllScale	OmpSs OpenAcc		
		Library	SyCL Kokkos PacxxV2 MultiController	TBB StarPU	MPI	Phast Dash Thrust Boost.Compute	SkePU EasyCL SkeCL		Pthreads	SkePU		Phast	PacxxV2 EasyCL Thrust Boost.Compute SkeCL Kokkos MultiController StarPU Pthreads	Dash	SyCL TBB MPI

			Programing Model							Memory Model					
			Kernel	Task	Directive	C++ Template	Skeleton	Wrapper	Threads	Hierarchical	Shared	Implicit	Explicit	PGAS	Distributed
GPU	Type	Language	OpenCL Cuda				Lift			OpenCL Cuda		Lift			
		Extension			OpenMP OpenAcc OmpSs					OpenMP	OpenMP		OmpSs OpenAcc		
		Library	SyCL Kokkos PacxxV2 MultiController	StarPU		Phast Dash Thrust Boost.Compute	SkePU EasyCL SkeCL			SkePU		Phast	PacxxV2 EasyCL Thrust Boost.Compute SkeCL Kokkos MultiController StarPU	Dash	SyCL

			Programing Model							Memory Model					
			Kernel	Task	Directive	C++ Template	Skeleton	Wrapper	Threads	Hierarchical	Shared	Implicit	Explicit	PGAS	Distributed
FPGA	Type	Language	OpenCL							OpenCL					
		Extension			OmpSs								OmpSs		
		Library	SyCL				SkeCL	EasyCL					EasyCL SkeCL		SyCL

Table 1: **High Performance Programming Models Classification Table**: HLS has been discarded because it depends heavily on the manufacturer. Libraries like: C++ AMP, ArrayFire, POCL, Halide, Raja, Tiramisu, NOVA, Bolt, FastFlow, Muesli, Chapel X10, UPC, among many others, have been omitted because some of them have been discontinued and others have evolved into new models. **Some highlights:** - OpenMP is in two memory paradigms at the same time because the device target mode is hierarchical. - Cilk is in two model types because is an extension but has features of a new language. - PHAST is the only C++ template paradigm with implicit memory transference. - All FPGAs programming models are based in OpenCL.

3.5 Portable Programming Models

Single source programming targeting multiple devices is a development goal to be accomplished. Performance portability is the programming paradigm with the ability to resolve the problem. It recalls well-establish highly-expressive techniques. For instance, programming is done using a set of primitives to develop complex programs, and most of the performance portability techniques use the same approach but with higher level primitives.

3.5.1 Classification

When trying to use multiple devices, some of these models could not be enough, but there are others with the characteristic of being single source programming models. We have extracted some of these models from our research (shown in Table 2) and find their principal device targeting and the changes needed to use the same source code in a different device.

Programming Model	Programming Paradigm	Memory Paradigm	Devices	Changes to other device
OpenMP	Directive	Hierarchical Shared	CPU, GPU Others	Few changes (only the directives)
OpenAcc	Directive	Hierarchical	CPU, GPU FPGAs, Others	Few changes (only the directives)
OmpSs	Directive	Hierarchical	CPU, GPU FPGAs, Others	Few changes (only the directives)
OpenCL	Kernel	Hierarchical	CPU, GPU FPGAs, Others	ICD, loader, few changes
SyCL	Kernel	Hierarchical	CPU, GPU FPGAs, Others	ICD and Loader
SkePU	Skeleton	Explicit	CPU, GPU	Recompile
Kokkos	Kernel	Explicit	CPU, GPU, Others	Recompile
PHAST	C++ Template	Implicit	CPU, GPU	Recompile

Table 2: **Single Source Programming Models:** Some of the models from Table 1 that are single source programming models, and the changes needed to use the same code to another device.

4. PHAST applied to the Caffe Framework

4.1 Why PHAST

PHAST code can be written once and targeted to different devices via a single macro. Its inner layers are implemented in Cuda C++ or `std::threads` so to allow targeting on Nvidia GPUs and multi-core CPUs. These layers are not part of the interface, so users can express their code in a platform-independent way. In fact, PHAST programmers can code their applications in terms of containers, iterators, and algorithms in an STL-like, thus using common sequential techniques [18].

As we have said in the previous section, PHAST use functors to modify the data stored in containers. This idea of a functor is borrowed from STL as a struct that inherits from a base functor struct and at least has the operator '()' defined. These functors are executed like kernels, therefore it is possible to see it like a loop calling the function in each iteration using each time the next element in the containers.

Despite PHAST promoting an architecture-agnostic programming style for productivity reasons, some small portions of code could benefit in performance from leveraging low-level architecture-specific features and optimizations. Well-known techniques like code replication on GPUs or the use of intrinsics on multi-cores are examples of this kind of low-level optimizations. Due to a possible performance gain, PHAST users could be willing to specialize some portions of their code according to the underlying device. PHAST does not prevent them from doing so, and low-level architecture-specific optimizations are still possible under the scope of ifdefs or similar constructs.

To test the PHAST library, it has been used to implement some applications such as AES [17], TRIAD [18] and DCT8x8 [18]. Also, an implementation of a histogram-stretching and an unsharp-mask filter is available as a set of tasks [19].

The PHAST library is available through request at its web page[1], but as part of a joint collaboration, we have early access to it.

4.2 The Caffe Framework

Caffe is the first DNN framework, developed by Berkeley AI Research. Nowadays, in the production environment, Caffe is replaced by Caffe2 and pyTorch, the Caffe successors, but in other cases, TensorFlow and MXNet are the selected ones. Caffe continues to be used in researching, due to the fact that it is very easy to modify, extend, or use, all of this without losing flexibility to run most of the state-of-the-art DNN models.

Caffe is a very good framework that runs on CPU or GPU only changing a flag, but this is done by having two implementations in two sorce codes files, one for CPU (.cpp) and other for GPU (.cu). Therefore, developers are forced to maintain two different versions of the same functionality. In this case, this is very well done and there are not a large number of differences between the files being the perfect target for a single source approach.

Before starting, an overview of the Caffe's internal structure, is given next. Caffe is built from multiple modules that work by themselves. It is possible to classify the blocks in two parts, containers and executors. Containers store data to be used by executors. Executors use the containers to exchange data and process it. For example, a layer gets a set of blobs, and with its own blobs, it computes the output ones.

A neural network has two different phases, inference and training. In the inference phase, data is passed through all the layers of the neural network in feed-forward mode to reach the last layer and get a result. But, in training mode, data is passed through all layers in feed-forward, like inference, but when it reaches the last layer of the network, that data is brought to a solver, it recalculates some values and starts the back-propagation through each layer but in reversed direction.

In this work, we are going to port all the blocks needed to be able to run two LeNet variations from example networks in Caffe for MNIST and CIFAR-10 databases. The blocks we need to port are

[1] https://www.phast-library.com/

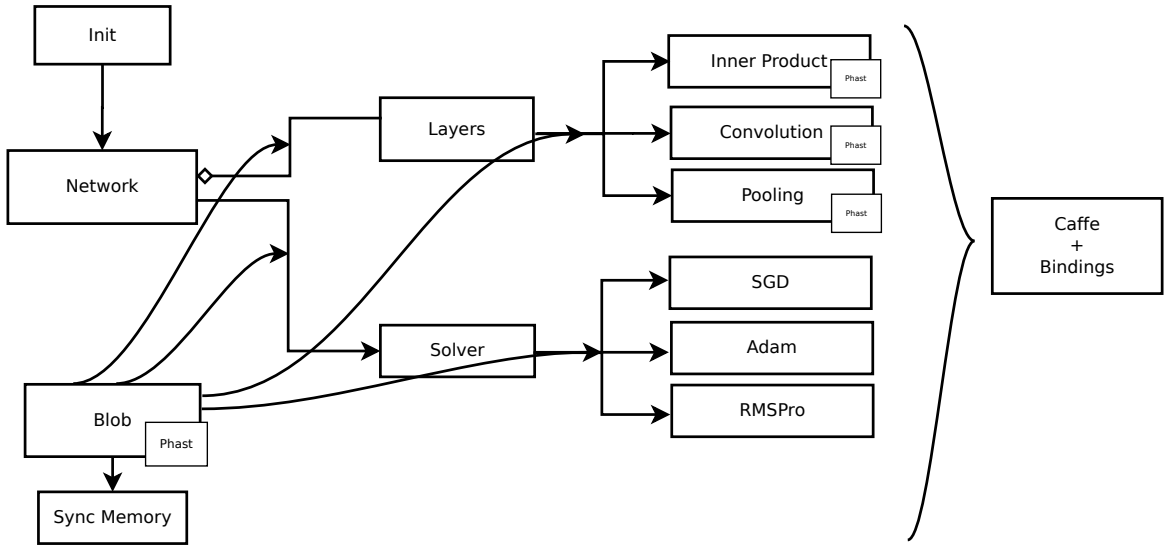

Figure 1: Part of the Caffe framework as a block diagram showing communications between blocks.

Blob, InnerProduct, Convolution, Pooling, ReLU, Accuracy, SoftMax, and SoftMax with Loss. We expect to run these neural networks through the Caffe binary in train and test mode using PHAST, therefore, we choose to modify only the blob and the lowest part of the main executors (Figure 1).

4.3 Development

We have removed the GPU code from Caffe to use the CPU version as our base for this project. In this way, we can focus on the algorithm itself and not in its GPU implementation.

From each block (or layer), we explain its most relevant part, showing its code compared to the original implementation. Sometimes, this new code is larger, but it is portable across the platforms that PHAST supports. We will show some functors related to its own block, but all of them are collected in one file, enabling the shared use of them.

4.3.1 Blob

The Blob block is a container for all the information in the neural network. It has to store two big arrays of data to be used at any required executor. The first array is named `data` and the second is `diff`. `data` is used as part of the feed-forward step. On the other hand `diff` is used to modify `data` at the back-propagation step.

Despite the fact that Caffe supports two data types, floating point in single and double precision, the Caffe binary only supports single precision, but using the python binding is possible to access to the double precision. Also, other data types are unsigned and signed integer that are used as extra information for the rest of the framework. One problem we found is that the current version of PHAST allows only single precision floating point numbers, therefore we have to override the Blob data type to use PHAST containers only when is possible.

The Blob block depends on the SyncMemory block to manage the memory, and depending on if the GPU is being used or not, it allocates the memory and manages the syncing process (Figure 2a). In our case, since we are only modifying a part of the framework, all the remainder part has to interact with our container, therefore we used the same trick as the GPU version of SyncMemory and have two containers, a host raw pointer and a PHAST vector (Figure 2b).

Although the Blob block is a container, it has some functionality inside. All of this functionality has been ported to PHAST, and we have added some extra functionality respecting to the PHAST vector container to convert it to another container (like a matrix or a cube) using a shallow copy.

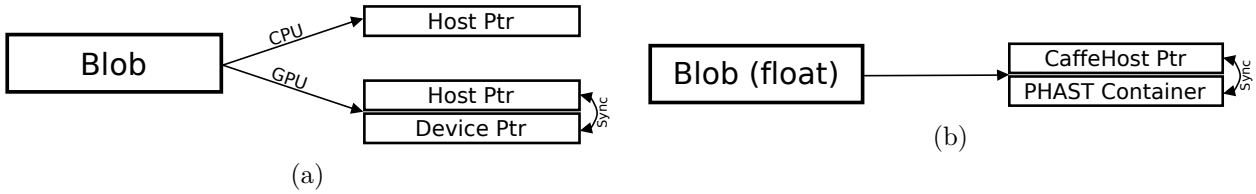

Figure 2: The Blob block from original Caffe (a) and the instanced version for single precision floating point numbers using PHAST (b).

4.3.2 Convolution

4.3.2.1 Feed-Forward

The Convolution block, or Convolution Layer, applies to the input a set of filters using a sliding window over the input. There are many ways to implement this sliding window, but applying the filter is doing a vector inner product for each sliding window. The most common variant of Convolution is Convolution 2-D (Figure 3), which is a simplification of a Convolution N-D.

As our sample network only use Convolution 2-D, we only port that specific variation. There is not much difference between a Convolution 2-D and a Convolution N-D, because we use the `im2col + gemm` implementation.

The `im2col + gemm` implementation is a way to map a convolution as a matrix multiplication (General Matrix Multiplication (GeMM)), but a data manipulation is needed to accomplish it. The `im2col` function maps the input matrix into columns to make the Convolution using a GeMM (Figure 4).

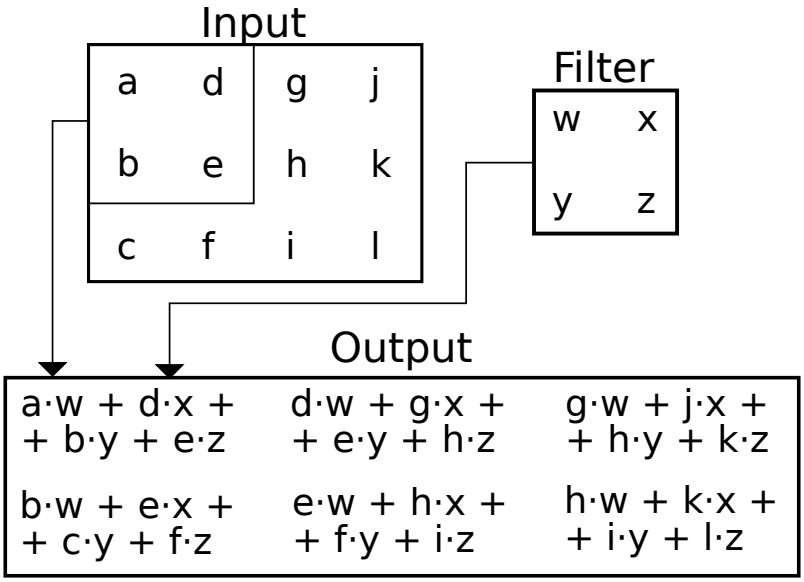

Figure 3: A convolution example using a 2x2 filter with stride 1 and padding 0 over a 4x3 input matrix.

The original Caffe's `im2col` function is a Penta-loop with dependencies in each iteration (Listing 1), therefore we decide to adapt it to be able to exploit a bit of parallelism. To create the PHAST version, we have merged all the loops and parametrized it with only one index (Listing 2). This change allows PHAST to use all the threads it sees appropriate.

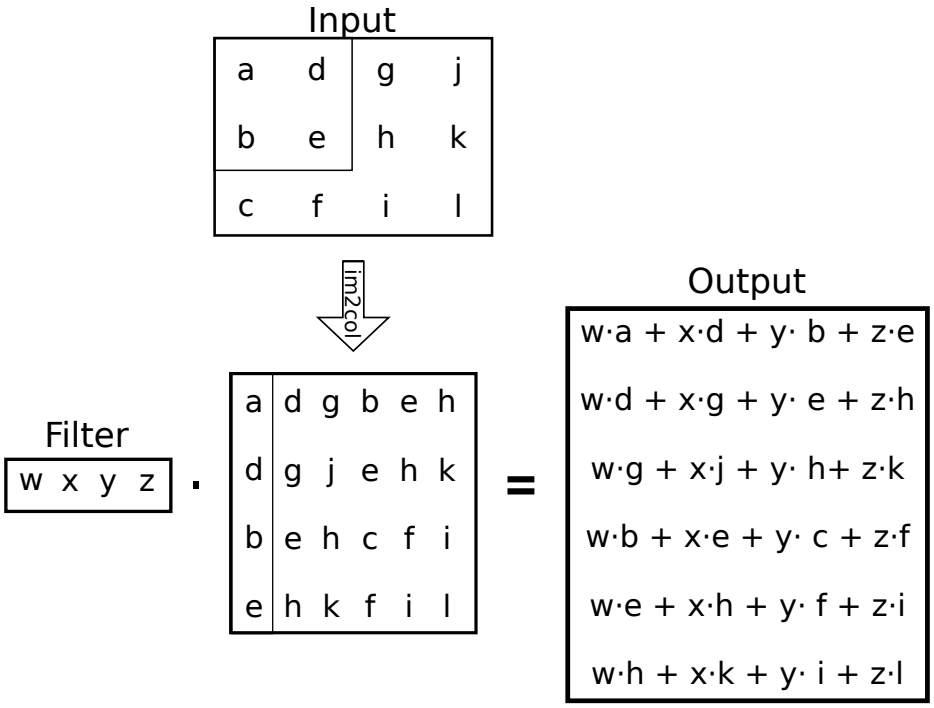

Figure 4: Convolution as a GeMM using the `im2col` function with 2x2 filter, stride 1, and padding 0.

```
template <typename Dtype>
void im2col_cpu(const Dtype* data_im, const int channels,
    const int height, const int width, const int kernel_h, const int
        ↪ kernel_w,
    const int pad_h, const int pad_w,
    const int stride_h, const int stride_w,
    const int dilation_h, const int dilation_w,
    Dtype* data_col) {
    const int output_h = (height + 2 * pad_h -
    (dilation_h * (kernel_h - 1) + 1)) / stride_h + 1;
    const int output_w = (width + 2 * pad_w -
    (dilation_w * (kernel_w - 1) + 1)) / stride_w + 1;
    const int channel_size = height * width;
    for (int channel = channels; channel--; data_im +=
        ↪ channel_size) {
    for (int kernel_row = 0; kernel_row < kernel_h; kernel_row++)
        for (int kernel_col = 0; kernel_col < kernel_w; kernel_col
            ↪ ++) {
        int input_row = -pad_h + kernel_row * dilation_h;
        for (int output_rows = output_h; output_rows; output_rows
            ↪ --) {
            if (!is_a_ge_zero_and_a_lt_b(input_row, height)) {
            for (int output_cols = output_w; output_cols;
                ↪ output_cols--) {
                *(data_col++) = 0;
            }
            } else {
            int input_col = -pad_w + kernel_col * dilation_w;
            for (int output_col = output_w; output_col; output_col
                ↪ --) {
                if (is_a_ge_zero_and_a_lt_b(input_col, width)) {
                *(data_col++) = data_im[input_row * width +
                    ↪ input_col];
                } else {
                *(data_col++) = 0;
                }
                input_col += stride_w;
            }
            }
            input_row += stride_h;
        }
        }
    }
    }
}
```

Listing 1: Caffe version: im2col

```
template <typename T, unsigned int policy = phast::
    ↪ get_default_policy()>
struct im2Col : phast::functor::func_scal<T, policy> {

  _PHAST_METHOD im2Col(int dilation_h, int dilation_w,
          int stride_h, int stride_w,
          int pad_h, int pad_w,
          int kernel_h, int kernel_w,
          int height, int width) {
    dh = dilation_h; dw = dilation_w;
    sh = stride_h; sw = stride_w;
    ph = pad_h; pw = pad_w;
    kh = kernel_h; kw = kernel_w;
     h = height; w = width;
  }

  _PHAST_METHOD void operator()(phast::functor::scalar<T>&
      ↪ col) {
    int index = this->get_index();

    const int oih = (h + 2 * ph - (dh * (kh - 1) + 1)) / sh + 1;
    const int oiw = (w + 2 * pw - (dw * (kw - 1) + 1)) / sw +
        ↪ 1;

    const int irow = -ph + (((index / (oih*oiw)) % (kh * kw)) /
        ↪ kw) * dh + ((index % (oih*oiw)) / oiw) * sh;
    const int icol = -pw + (((index / (oih*oiw)) % (kh * kw)) %
        ↪ kw) * dw + ((index % (oih*oiw)) % oiw) * sw;
    if (irow >= 0 && irow < h && icol >= 0 && icol < w)
      col = in.at(((index / (oih*oiw)) / (kh * kw)), irow * w +
          ↪ icol);
    else col = 0;
  }

  int dh, dw;
  int sh, sw;
  int ph, pw;
  int kh, kw;
  int h, w;

  phast::functor::matrix<T> in;
};
```

Listing 2: PHAST version: im2col

4.3.2.2 Back-Propagation

In the feed-forward stage, the im2col function duplicates some values to make a Convolution with a GeMM. In the back-propagation, we need to apply the reverse step to propagate the gradients to the previous layers. The most important part is the usage of `col2im` to map the gradients to the size of the input data.

Like in the feed-forward stage, the original implementation is also a Penta-loop (Listing 3), therefore we have followed the same approach as before and we have merged the loops and parametrized with only one index (Listing 4).

```
template <typename Dtype>
void col2im_cpu(const Dtype* data_col, const int channels,
    const int height, const int width, const int kernel_h, const int
        ↪ kernel_w,
    const int pad_h, const int pad_w,
    const int stride_h, const int stride_w,
    const int dilation_h, const int dilation_w,
    Dtype* data_im) {
  caffe_set(height * width * channels, Dtype(0), data_im);
  const int output_h = (height + 2 * pad_h -
    (dilation_h * (kernel_h - 1) + 1)) / stride_h + 1;
  const int output_w = (width + 2 * pad_w -
    (dilation_w * (kernel_w - 1) + 1)) / stride_w + 1;
  const int channel_size = height * width;
  for (int channel = channels; channel--; data_im +=
      ↪ channel_size) {
    for (int kernel_row = 0; kernel_row < kernel_h; kernel_row++)
        ↪ {
      for (int kernel_col = 0; kernel_col < kernel_w; kernel_col++)
          ↪ {
        int input_row = -pad_h + kernel_row * dilation_h;
        for (int output_rows = output_h; output_rows; output_rows
            ↪ --) {
          if (!is_a_ge_zero_and_a_lt_b(input_row, height)) {
            data_col += output_w;
          } else {
            int input_col = -pad_w + kernel_col * dilation_w;
            for (int output_col = output_w; output_col; output_col
                ↪ --) {
              if (is_a_ge_zero_and_a_lt_b(input_col, width)) {
                data_im[input_row * width + input_col] += *
                    ↪ data_col;
              }
              data_col++;
              input_col += stride_w;
            }
          }
          input_row += stride_h;
        }
      }
    }
  }
}
```

Listing 3: Caffe version: col2im

```
template <typename T, unsigned int policy = phast::
    ↪ get_default_policy()>
struct col2Im : phast::functor::func_scal<T, policy> {

  _PHAST_METHOD col2Im(int dilation_h, int dilation_w,
                       int stride_h, int stride_w,
                       int pad_h, int pad_w,
                       int kernel_h, int kernel_w,
                       int height, int width) {
    dh = dilation_h; dw = dilation_w;
    sh = stride_h; sw = stride_w;
    ph = pad_h; pw = pad_w;
    kh = kernel_h; kw = kernel_w;
    h = height; w = width;

    oh = (h + 2 * ph - (dh * (kh - 1) + 1)) / sh + 1;
    ow = (w + 2 * pw - (dw * (kw - 1) + 1)) / sw + 1;
  }

  _PHAST_METHOD void operator()(phast::functor::scalar<T>&
      ↪ col) {
    int index = this->get_index();

    int oc = index % ow;
    int orp = (index / ow) % oh;
    int c = index / (ow*oh*kh*kw);
    int kr = ((index / (ow * oh)) % (kh*kw)) / kw;
    int kc = ((index / (ow * oh)) % (kh*kw)) % kw;
    int ir = -ph + kr * dh;
    int ic = -pw + kc * dw;

    if (ic >= 0 && ic < w && ir >= 0 && ir < h)
      in.at(c, (ir + orp * sh) * w + (ic + oc * sw)) += col;
  }

  int dh, dw;
  int sh, sw;
  int ph, pw;
  int kh, kw;
  int h, w;
  int oh,ow;

  phast::functor::matrix<T> in;
};
```

Listing 4: PHAST version: col2im

4.3.3 Pooling

4.3.3.1 Feed-Forward

The Pooling layer applies a mathematical function to a set of numbers to get only one value, such as maximum or minimum. Like the Convolution layer, it works using a sliding window over the input data and applying the function to each set.

As it can be seen in Listing 5, the structure is very similar to the Convolution block, but this time, we do not apply the same technique, because we have not enough time to verify that the merged version works properly. Therefore, we have only parallelized the outer loop (Listing 6).

```
// The main loop
for (int n = 0; n < bottom[0]->num(); ++n) {
    for (int c = 0; c < channels_; ++c) {
    for (int ph = 0; ph < pooled_height_; ++ph) {
        for (int pw = 0; pw < pooled_width_; ++pw) {
        int hstart = ph * stride_h_ - pad_h_;
        int wstart = pw * stride_w_ - pad_w_;
        int hend = min(hstart + kernel_h_, height_);
        int wend = min(wstart + kernel_w_, width_);
        hstart = max(hstart, 0);
        wstart = max(wstart, 0);
        const int pool_index = ph * pooled_width_ + pw;
        for (int h = hstart; h < hend; ++h) {
            for (int w = wstart; w < wend; ++w) {
            const int index = h * width_ + w;
            if (bottom_data[index] > top_data[pool_index]) {
                top_data[pool_index] = bottom_data[index];
                if (use_top_mask) {
                top_mask[pool_index] = static_cast<Dtype>(index
                    ↪ );
                } else {
                mask[pool_index] = static_cast<Dtype>(index);
                }
            }
            }
        }
        }
    }
    // compute offset
    bottom_data += bottom[0]->offset(0, 1);
    top_data += top[0]->offset(0, 1);
    if (use_top_mask) {
        top_mask += top[0]->offset(0, 1);
    } else {
        mask += top[0]->offset(0, 1);
    }
    }
}
```

Listing 5: Caffe version: Max Pooling

```
template <typename T, unsigned int policy = phast::
    ↪ get_default_policy()>
struct poolingMax : phast::functor::func_mat_mat<T, policy> {

  _PHAST_METHOD poolingMax(int stride_h, int stride_w,
                           int pad_h, int pad_w,
                           int kernel_h, int kernel_w,
                           int height, int width) {
    sh = stride_h; sw = stride_w;
    ph = pad_h; pw = pad_w;
    kh = kernel_h; kw = kernel_w;
    h = height; w = width;
  }

  _PHAST_METHOD void operator()(phast::functor::matrix<T>&
      ↪ in,phast::functor::matrix<T>& out) {
    auto outIT = out.begin_ij();
    int index = this->get_index();

    for (int i = 0; i < out.size_i(); ++i) {
      for (int j = 0; j < out.size_j(); ++j, ++outIT) {

        int hstart = i * sh - ph;
        int hend = smin(hstart + kh, h);
        hstart = smax(hstart, 0);

        int wstart = j * sw - pw;
        int wend = smin(wstart + kw, w);
        wstart = smax(wstart, 0);

        T num = *outIT;
        for (int y = hstart; y < hend; ++y) {
          for (int x = wstart; x < wend; ++x) {
            if (num < in[y][x]) {
              num = in[y][x];
              mask.at(index , i, j) = y * w + x;
            }
          }
        }
        *outIT = num;
      }
    }
  }

  phast::functor::cube<T> mask;

  int sh, sw;
  int ph, pw;
  int kh, kw;
  int h, w;
};
```

Listing 6: PHAST version: Max Pooling

4.3.3.2 Back-Propagation

During the feed-forward stage, we have stored from where we have been taking each output value, therefore we have to map the values from the output to the input using that list of mapped values. Its objective is to apply all the gradients to its corresponding positions.

Like in the feed-forward stage, we have not merged all the loops (in Listing 7), because we did not verified that it will continue working properly. Therefore, we have only parallelized the first loop (Listing 8).

```
for (int n = 0; n < top[0]->num(); ++n) {
    for (int c = 0; c < channels_; ++c) {
    for (int ph = 0; ph < pooled_height_; ++ph) {
        for (int pw = 0; pw < pooled_width_; ++pw) {
        const int index = ph * pooled_width_ + pw;
        const int bottom_index =
        use_top_mask ? (int)top_mask[index] : (int)mask[index];
        bottom_diff[bottom_index] += top_diff[index];
        }
    }
    bottom_diff += bottom[0]->offset(0, 1);
    top_diff += top[0]->offset(0, 1);
    if (use_top_mask) top_mask += top[0]->offset(0, 1);
    else mask += top[0]->offset(0, 1);
    }
}
```

Listing 7: Caffe version: Back Max Pooling

```
template <typename T, unsigned int policy = phast::
    ↪ get_default_policy()>
struct poolingMaxBack : phast::functor::func_mat_mat<T, policy>
    ↪ {
  _PHAST_METHOD poolingMaxBack(int pooled_h, int pooled_w,
      ↪ int width) {
    ph = pooled_h; pw = pooled_w;
    w = width;
  }
  _PHAST_METHOD void operator()(phast::functor::matrix<T>&
      ↪ bottom,phast::functor::matrix<T>& top) {
    int channel = this->get_index();
    for (int i = 0; i < ph; ++i) {
      for (int j = 0; j < pw; ++j) {
        int index = mask[channel][i][j];
        int y = index / w; int x = index % w;
        bottom[y][x] += top[i][j];
      }
    }
  }
  phast::functor::cube<T> mask;
  int ph, pw;
  int w;
};
```

Listing 8: PHAST version: Back Max Pooling

4.3.4 InnerProduct

4.3.4.1 Feed-Forward

In neural networks, there is a layer usually known as Perceptron layer (or Dense layer), sometimes is also known as InnerProduct, because the output of the layer is the inner product of the input with the weights.

```
template <typename Dtype>
void InnerProductLayer<Dtype>::Forward_cpu(const vector<Blob
    ↪ <Dtype>*>& bottom,
    const vector<Blob<Dtype>*>& top) {

  const Dtype* bottom_data = bottom[0]->cpu_data();
  Dtype* top_data = top[0]->mutable_cpu_data();
  const Dtype* weight = this->blobs_[0]->cpu_data();

  caffe_cpu_gemm<Dtype>(CblasNoTrans, transpose_ ?
      ↪ CblasNoTrans : CblasTrans,
      M_, N_, K_, (Dtype)1.,
      bottom_data, weight, (Dtype)0., top_data);

  if (bias_term_) {
    caffe_cpu_gemm<Dtype>(CblasNoTrans, CblasNoTrans, M_,
        ↪ N_, 1, (Dtype)1.,
        bias_multiplier_.cpu_data(),
        this->blobs_[1]->cpu_data(), (Dtype)1., top_data);
  }
}
```

Listing 9: Caffe version: InnerProduct

```
template <typename T, unsigned int policy = phast::
    ↪ get_default_policy()>
struct matrixPlusVectorRows : phast::functor::func_vec<T, policy
    ↪ > {
  _PHAST_METHOD matrixPlusVectorRows() {}

  _PHAST_METHOD void operator()(phast::functor::vector<T>&
      ↪ row) {
    for(auto r = row.begin(), i = vec.begin(); r != row.end(); ++r,
        ↪ ++i)
      *r += *i;
  }
  phast::functor::vector<T> vec;
};

template <>
void InnerProductLayer<float>::Forward_cpu(const vector<Blob<
    ↪ float>*>& bottom,
  const vector<Blob<float>*>& top) {
  phast::matrix<float> matA = bottom[0]->getDataAsMatrix(M_
      ↪ , K_, false);
  phast::matrix<float> matB = this->blobs_[0]->
      ↪ getDataAsMatrix(K_, N_, !transpose_);
  phast::matrix<float> matC = top[0]->getDataAsMatrix(M_, N_
      ↪ , false);

  phast::dot_product(matA, matB, matC);

  if (bias_term_) {
    matrixPlusVectorRows<float> matrixPlusVectorRows;
    matrixPlusVectorRows.vec.link(this->blobs_[1]->
        ↪ getDataAsVector(N_));
    phast::for_each(matC.begin_i(), matC.end_i(),
        ↪ matrixPlusVectorRows);
  }
  if(!transpose_) matB.transpose();
}
```

Listing 10: PHAST version: InnerProduct

It is interesting that, in Listing 9 in line 13, under the condition of `bias_term_`, there is a matrix multiplication, but in Listing 10 under the same condition in line 21, there is a call to `matrixPlusVectorRows`. This is a trick made very often by Caffe, its creators have mapped all possible operations to matrix multiplications to be easier to port them to GPU, but now, we can make specific functors for these operations.

4.3.4.2 Back-Propagation

The InnerProduct back-propagation is a bit different from the other block we have seen. Instead of reversing the operation made in feed-forward to map to each value its own gradient and modify its weight, we add to the weights a scaled gradient based on the original data, then the same with the bias, and lastly, we propagate the changes to the previous layer. Despite of the trick to map all operations to matrix multiplications, this layer is very straight forward (Listings 11 and 12).

```
1  template <typename Dtype>
2  void InnerProductLayer<Dtype>::Backward_cpu(const vector<
       ↪ Blob<Dtype>*>& top,
3      const vector<bool>& propagate_down,
4      const vector<Blob<Dtype>*>& bottom) {
5    if (this->param_propagate_down_[0]) {
6      const Dtype* top_diff = top[0]->cpu_diff();
7      const Dtype* bottom_data = bottom[0]->cpu_data();
8      // Gradient with respect to weight
9      if (transpose_) {
10       caffe_cpu_gemm<Dtype>(CblasTrans, CblasNoTrans,
11           K_, N_, M_,
12           (Dtype)1., bottom_data, top_diff,
13           (Dtype)1., this->blobs_[0]->mutable_cpu_diff());
14     } else {
15       caffe_cpu_gemm<Dtype>(CblasTrans, CblasNoTrans,
16           N_, K_, M_,
17           (Dtype)1., top_diff, bottom_data,
18           (Dtype)1., this->blobs_[0]->mutable_cpu_diff());
19     }
20   }
21   if (bias_term_ && this->param_propagate_down_[1]) {
22     const Dtype* top_diff = top[0]->cpu_diff();
23     // Gradient with respect to bias
24     caffe_cpu_gemv<Dtype>(CblasTrans, M_, N_, (Dtype)1.,
         ↪ top_diff,
25         bias_multiplier_.cpu_data(), (Dtype)1.,
26         this->blobs_[1]->mutable_cpu_diff());
27   }
28   if (propagate_down[0]) {
29     const Dtype* top_diff = top[0]->cpu_diff();
30     // Gradient with respect to bottom data
31     caffe_cpu_gemm<Dtype>(CblasNoTrans, transpose_ ?
         ↪ CblasTrans : CblasNoTrans,
32                           M_, K_, N_,
33                           (Dtype)1., top_diff, this->blobs_[0]->
                               ↪ cpu_data(),
34                           (Dtype)0., bottom[0]->mutable_cpu_diff
                               ↪ ());
35   }
36 }
```

Listing 11: Caffe version: Back InnerProduct

```
1  template <>
2  void InnerProductLayer<float>::Backward_cpu(const vector<Blob
       ↪ <float>*>& top,
3      const vector<bool>& propagate_down,
4      const vector<Blob<float>*>& bottom) {
5
6    if (this->param_propagate_down_[0]) {
7      if (transpose_) {
8        phast::matrix<float> diff = top[0]->getDiffAsMatrix(M_,
             ↪ N_, false);
9        phast::matrix<float> data = bottom[0]->getDataAsMatrix
             ↪ (K_, M_, true);
10       phast::matrix<float> tmp(K_, N_, 0);
11       phast::matrix<float> wdiff = this->blobs_[0]->
             ↪ getDiffAsMatrix(K_, N_, false);
12       phast::dot_product(data, diff, tmp);
13       phast::transform(tmp.begin_ij(), tmp.end_ij(), wdiff.begin_ij()
             ↪ , wdiff.begin_ij(),
14                        phast::plus<float>());
15       data.transpose();
16     }
17     else {
18       phast::matrix<float> diff = top[0]->getDiffAsMatrix(N_,
             ↪ M_, true);
19       phast::matrix<float> data = bottom[0]->getDataAsMatrix
             ↪ (M_, K_, false);
20       phast::matrix<float> tmp(N_, K_, 0);
21       phast::matrix<float> wdiff = this->blobs_[0]->
             ↪ getDiffAsMatrix(N_, K_, false);
22       phast::dot_product(diff, data, tmp);
23       phast::transform(tmp.begin_ij(), tmp.end_ij(), wdiff.begin_ij()
             ↪ , wdiff.begin_ij(),
24                        phast::plus<float>());
25       diff.transpose();
26     }
27   }
28   if (bias_term_ && this->param_propagate_down_[1]) {
29     phast::matrix<float> diff = top[0]->getDiffAsMatrix(N_, M_,
           ↪ true);
30     phast::vector<float> weig = this->blobs_[1]->
           ↪ getDiffAsVector(N_);
31
32     phast::transform(diff.begin_i(), diff.end_i(), weig.begin(),
           ↪ reduceMatrixVectors<float>());
33
34     diff.transpose();
35   }
36   if (propagate_down[0]) {
37     phast::matrix<float> diff = top[0]->getDiffAsMatrix(M_, N_,
           ↪ false);
38     phast::matrix<float> weig = this->blobs_[0]->
           ↪ getDataAsMatrix(N_, K_, transpose_);
39
40     phast::matrix<float> bdiff = bottom[0]->getDiffAsMatrix(M_
           ↪ , K_, false);
41
42     phast::dot_product(diff, weig, bdiff);
43     if(transpose_) weig.transpose();
44   }
45 }
```

Listing 12: PHAST version: Back InnerProduct

4.3.5 ReLU

4.3.5.1 Feed-Forward

ReLU or Rectified-Linear layer applies the ReLU function to each of its inputs. It is mainly used to remove negative numbers from a neural network, but Caffe has merged it with the Leaky-ReLU. The difference between a Leaky-ReLU and a ReLU is in negative numbers, ReLU outputs 0s, and Leaky-ReLU outputs the input value multiplied by a constant (Equation 1 and Listings 13 and 14).

$$Leaky - ReLU(x) = \begin{cases} x & \text{if } x > 0 \\ ax & \text{otherwise} \end{cases} \tag{1}$$

```
for (int i = 0; i < count; ++i) {
  top_data[i] = std::max(bottom_data[i], Dtype(0))
      + negative_slope * std::min(bottom_data[i], Dtype(0));
}
```

Listing 13: Caffe version: ReLU

```
template <typename T, unsigned int policy = phast::
    ↪ get_default_policy()>
struct reluFunc : phast::functor::func_scal_scal<T, policy> {

  _PHAST_METHOD reluFunc(T negative) {
    slope = negative;
  }

  _PHAST_METHOD void operator()(phast::functor::scalar<T>&
      ↪ in,phast::functor::scalar<T>& out) {
    out = smax(in, T(0)) + slope * smin(in, T(0));
  }

  T slope;
};
```

Listing 14: PHAST version: ReLU

4.3.5.2 Back-Propagation

Using the gradient as x, and the original data as y, we have a very similar equation to make the back-propagation algorithm (Equation 2). Like before, the code is very straight forward (Listings 15 and 16).

$$Back - Leaky - ReLU(x, y) = \begin{cases} x & \text{if } y > 0 \\ a & \text{otherwise} \end{cases} \tag{2}$$

```
for (int i = 0; i < count; ++i) {
  bottom_diff[i] = top_diff[i] * ((bottom_data[i] > 0)
      + negative_slope * (bottom_data[i] <= 0));
}
```

Listing 15: Caffe version: Back ReLU

```
template <typename T, unsigned int policy = phast::
    ↪ get_default_policy()>
struct reluBackFunc : phast::functor::func_scal_scal_scal<T, policy
    ↪ > {

  _PHAST_METHOD reluBackFunc(T negative) {
    slope = negative;
  }

  _PHAST_METHOD void operator()(phast::functor::scalar<T>&
      ↪ in, phast::functor::scalar<T>& diff, phast::functor::
      ↪ scalar<T>& out) {
    int m0 = in > 0 ? 1 : 0;
    int m1 = in <= 0 ? 1 : 0;
    out = diff * (m0 + slope * m1);
  }

  T slope;
};
```

Listing 16: PHAST version: Back ReLU

4.3.6 Softmax and Softmax Loss

4.3.6.1 Feed-Forward

In classifiers, a probability distribution over a discrete variable with n possible values, is represented using the softmax function [20]. Indeed, we require not only that each element to be between 0 and 1, but also that the entire vector sums to 1 so that it represents a valid probability distribution [20].

We are going to apply the softmax function (Equation 3) to each value of the output. As we work with many outputs at the same time, we apply it over each vector of the data matrix (Listing 17). It is worth to notice that, most of the code in the PHAST version (Listing 18) is data preparation, which will be improved in future versions of PHAST.

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \tag{3}$$

```
template <typename Dtype>
void SoftmaxLayer<Dtype>::Forward_cpu(const vector<Blob<
    ↪ Dtype>*>& bottom,
    const vector<Blob<Dtype>*>& top) {
  const Dtype* bottom_data = bottom[0]->cpu_data();
  Dtype* top_data = top[0]->mutable_cpu_data();
  Dtype* scale_data = scale_.mutable_cpu_data();
  int channels = bottom[0]->shape(softmax_axis_);
  int dim = bottom[0]->count() / outer_num_;
  caffe_copy(bottom[0]->count(), bottom_data, top_data);
  // We need to subtract the max to avoid numerical issues,
      ↪ compute the exp,
  // and then normalize.
  for (int i = 0; i < outer_num_; ++i) {
    // initialize scale_data to the first plane
    caffe_copy(inner_num_, bottom_data + i * dim, scale_data);
    for (int j = 0; j < channels; j++) {
      for (int k = 0; k < inner_num_; k++) {
        scale_data[k] = std::max(scale_data[k],
            bottom_data[i * dim + j * inner_num_ + k]);
      }
    }
    // subtraction
    caffe_cpu_gemm<Dtype>(CblasNoTrans, CblasNoTrans,
        ↪ channels, inner_num_,
        1, -1., sum_multiplier_.cpu_data(), scale_data, 1., top_data
            ↪ );
    // exponentiation
    caffe_exp<Dtype>(dim, top_data, top_data);
    // sum after exp
    caffe_cpu_gemv<Dtype>(CblasTrans, channels, inner_num_, 1.,
        top_data, sum_multiplier_.cpu_data(), 0., scale_data);
    // division
    for (int j = 0; j < channels; j++) {
      caffe_div(inner_num_, top_data, scale_data, top_data);
      top_data += inner_num_;
    }
  }
}
```

Listing 17: Caffe version: SoftMax

```
template <>
void SoftmaxLayer<float>::Forward_cpu(const vector<Blob<float
    ↪ >*>& bottom,
    const vector<Blob<float>*>& top) {
  int channels = bottom[0]->shape(softmax_axis_);
  phast::cube<float> data = bottom[0]->getDataAsCube(
      ↪ outer_num_, channels, inner_num_);
  phast::cube<float> out = top[0]->getDataAsCube(outer_num_,
      ↪ channels, inner_num_);
  phast::vector<float> scale = scale_.getDataAsVector(inner_num_
      ↪ );

  phast::copy(data.begin_ijk(), data.end_ijk(), out.begin_ijk());

  auto dataIt = data.begin_i();
  auto outIt = out.begin_i();

  for (; dataIt != data.end_i() && outIt != out.end_i(); dataIt++,
      ↪ outIt++) {
    phast::matrix<float> dataM;
    phast::matrix<float> outM;
    dataM.set_dev(data.size_j(), data.size_k(), dataIt.get_dev() +
        ↪ dataIt.get_abs_pos());
    outM.set_dev(out.size_j(), out.size_k(), outIt.get_dev() + outIt.
        ↪ get_abs_pos());

    auto dataMIt = dataM.begin_i();

    phast::vector<float> tmpRow;
    tmpRow.set_dev(dataM.size_j(), dataMIt.get_dev() + dataMIt.
        ↪ get_abs_pos());
    phast::copy(tmpRow.begin(), tmpRow.end(), scale.begin());

    for (; dataMIt != dataM.end_i(); dataMIt++) {
      phast::vector<float> dataV;
      dataV.set_dev(dataM.size_j(), dataMIt.get_dev() + dataMIt.
          ↪ get_abs_pos());
      phast::transform(dataV.begin(), dataV.end(), scale.begin(),
          ↪ maxFunc<float>());
    }

    matrixMinusVectorRows<float> matrixMinusVectorRows;
    matrixMinusVectorRows.vec.link(scale);
    phast::for_each(outM.begin_i(), outM.end_i(),
        ↪ matrixMinusVectorRows);
    phast::for_each(outM.begin_ij(), outM.end_ij(), expFunc<float
        ↪ >());
    outM.transpose();
    phast::transform(scale.begin(), scale.end(), outM.begin_i(),
        ↪ matrixReduceByRows<float>());
    outM.transpose();
    matrixDivVectorRows<float> matrixDivVectorRows;
    matrixDivVectorRows.vec.link(scale);
    phast::for_each(outM.begin_i(), outM.end_i(),
        ↪ matrixDivVectorRows);
  }
}
```

Listing 18: PHAST version: SoftMax

4.3.6.2 Back-Propagation

Like in the InnerProduct layer, the objective of the back-propagation stage is not to reverse the feed-forward stage, but to scale the data to the previous layers in the network (Listings 19 and 20). Same as before, most of the code is the PHAST version is data preparation.

```
template <typename Dtype>
void SoftmaxLayer<Dtype>::Backward_cpu(const vector<Blob<
    ↪ Dtype>*>& top,
  const vector<bool>& propagate_down,
  const vector<Blob<Dtype>*>& bottom) {
 const Dtype* top_diff = top[0]->cpu_diff();
 const Dtype* top_data = top[0]->cpu_data();
 Dtype* bottom_diff = bottom[0]->mutable_cpu_diff();
 Dtype* scale_data = scale_.mutable_cpu_data();
 int channels = top[0]->shape(softmax_axis_);
 int dim = top[0]->count() / outer_num_;
 caffe_copy(top[0]->count(), top_diff, bottom_diff);
 for (int i = 0; i < outer_num_; ++i) {
   // compute dot(top_diff, top_data) and subtract them from the
       ↪ bottom diff
   for (int k = 0; k < inner_num_; ++k) {
     scale_data[k] = caffe_cpu_strided_dot<Dtype>(channels,
         bottom_diff + i * dim + k, inner_num_,
         top_data + i * dim + k, inner_num_);
   }
   // subtraction
   caffe_cpu_gemm<Dtype>(CblasNoTrans, CblasNoTrans,
       ↪ channels, inner_num_, 1,
       -1., sum_multiplier_.cpu_data(), scale_data, 1., bottom_diff
           ↪ + i * dim);
 }
 // elementwise multiplication
 caffe_mul(top[0]->count(), bottom_diff, top_data, bottom_diff);
}
```

Listing 19: Caffe version: Back SoftMax

```
template <>
void SoftmaxLayer<float>::Backward_cpu(const vector<Blob<
    ↪ float>*>& top,
  const vector<bool>& propagate_down,
  const vector<Blob<float>*>& bottom) {

 int channels = top[0]->shape(softmax_axis_);

 phast::cube<float> topDiff = top[0]->getDiffAsCube(
     ↪ outer_num_, channels, inner_num_);
 phast::cube<float> topData = top[0]->getDataAsCube(
     ↪ outer_num_, channels, inner_num_);

 phast::cube<float> botDiff = bottom[0]->getDiffAsCube(
     ↪ outer_num_, channels, inner_num_);
 phast::vector<float> scaler = scale_.getDataAsVector(
     ↪ inner_num_);

 phast::copy(topDiff.begin_ijk(), topDiff.end_ijk(), botDiff.
     ↪ begin_ijk());

 auto botIt = botDiff.begin_i();
 auto topDIt = topData.begin_i();

 for (; botIt != botDiff.end_i() && topDIt != topData.end_i();
     ↪ botIt++, topDIt++) {
   phast::matrix<float> botM;
   phast::matrix<float> topDM;
   botM.set_dev(botDiff.size_j(), botDiff.size_k(), botIt.get_dev()
       ↪ + botIt.get_abs_pos());
   topDM.set_dev(topData.size_j(), topData.size_k(), topDIt.
       ↪ get_dev() + topDIt.get_abs_pos());

   botM.transpose();
   topDM.transpose();

   reduceMatrixVectorByVectorDot<float>
       ↪ reduceMatrixVectorByVectorDot;
   reduceMatrixVectorByVectorDot.scal.link(scaler);
   phast::transform(botM.begin_i(), botM.end_i(), topDM.begin_i
       ↪ (), reduceMatrixVectorByVectorDot);

   botM.transpose();
   topDM.transpose();

   matrixMinusVectorRows<float> matrixMinusVectorRows;
   matrixMinusVectorRows.vec.link(scaler);
   phast::for_each(botM.begin_i(), botM.end_i(),
       ↪ matrixMinusVectorRows);
 }

 phast::transform(botDiff.begin_ijk(), botDiff.end_ijk(), topData.
     ↪ begin_ijk(), botDiff.begin_ijk(), phast::multiplies<float
     ↪ >());
}
```

Listing 20: PHAST version: Back SoftMax

4.3.6.3 With Loss

Regarding the SoftMax with Loss block, it is the same as SoftMax but adding similar code to the Accuracy Layer into the SoftMax with Loss to calculate the logarithmic error instead of accuracy. And, like the Accuracy layer, it has not back-propagation except the SoftMax itself.

4.3.7 Accuracy

4.3.7.1 Feed-Forward

The Accuracy layer is implicit in Caffe, it checks the numbers of good predictions and generates the corresponding percentage (Listings 21 and 22).

```
for (int j = 0; j < inner_num_; ++j) {
    const int label_value =
        static_cast<int>(bottom_label[i * inner_num_ + j]);
    if (has_ignore_label_ && label_value == ignore_label_) {
    continue;
    }
    DCHECK_GE(label_value, 0);
    DCHECK_LT(label_value, num_labels);
    if (top.size() > 1) ++nums_buffer_.mutable_cpu_data()[
        ↪ label_value];
    const Dtype prob_of_true_class = bottom_data[i * dim
                                    + label_value * inner_num_ + j];
    int num_better_predictions = -1; // true_class also counts as "
        ↪ better"
    // Top-k accuracy
    for (int k = 0; k < num_labels && num_better_predictions <
        ↪ top_k_; ++k) {
    num_better_predictions +=
        (bottom_data[i * dim + k * inner_num_ + j] >=
            ↪ prob_of_true_class);
    }
    // check if there are less than top_k_ predictions
    if (num_better_predictions < top_k_) {
    ++accuracy;
    if (top.size() > 1) ++top[1]->mutable_cpu_data()[label_value
        ↪ ];
    }
    ++count;
}
```

Listing 21: Caffe version: Accuracy

```
template <typename T, unsigned int policy = phast::
    ↪ get_default_policy()>
struct doAccuracy : phast::functor::func_vec<T, policy> {

  _PHAST_METHOD doAccuracy(int label_, int maxLabels, int
      ↪ ignore, int ignoreValue, int isTop, int top_k) {
    label = label_;
    labels = maxLabels;
    hasIgnore = ignore;
    value = ignoreValue;
    top = isTop;
    topk = top_k;
  }

  _PHAST_METHOD T operator()(phast::functor::vector<T>&
      ↪ row) {
    T acc = 0;
    if (hasIgnore && label == value) return 0;
    if (label < 0 || label > labels) return 0;

    if (top > 1) num[label]++;
    T prob = row[label];

    int predicts = -1;

    for (auto it = row.begin(); it != row.end() && predicts < topk
        ↪ ; ++it) {
      if ((*it) >= prob) predicts++;
    }

    if (predicts < topk) {
      ++acc;
      if (top > 1) out1[label]++;
    }

    return acc;
  }

  phast::functor::vector<T> out1;
  phast::functor::vector<T> num;

  int label;
  int labels;
  int hasIgnore;
  int value;
  int top;
  int topk;
};
```

Listing 22: PHAST version: Accuracy

4.3.7.2 Back-Propagation

The Accuracy layer does not have any kind of back-propagation because it does not modify the data.

4.3.8 Pending work

There is some functionality that we have not implemented due to lack of time. In the Accuracy block test, we have not implemented yet multiple axes checking, it means that our version will not check against multi-labeled output. Also, due to some PHAST limitations that are going to be solved soon, we cannot implement the ignore label, which allows ignoring some parts of the batch in the accuracy measurement. On the other hand, the Convolution block has been only implemented the simplest 2D convolution using a 2D `im2col`. Also, we discarded the usage of independent filters using groups until we get more familiarized with the PHAST library.

5. Results & Evaluation

5.1 Testbench

In this study, we have used the PHAST library 1.0.1 compiled with GCC 6.3.0 and Cuda 9.0. Caffe was obtained from the official git repository[2], the commit we have used is `99bd99795dcdf0b1d3086a8d67ab1782a8a08383`. Our compute machine is part of GACOP's computer cluster [21], it is running CentOS Linux 7.5 1804 with Linux 3.10.0-862.14.4, powered by two Intel(R) Xeon(R) CPU E5-2603 v3 @ 1.60GHz with 64 GiB RAM memory, and a Geforce GTX 1080 8GB GDDR5X with Nvidia driver 410.48.

With the original Caffe code, we have trained two neural networks from Caffe examples, both are LeNet-based networks. The first one is able to classify the MNIST[3] database, the network is built from 6 layers: 2 Convolutions, 2 Pooling, and 2 InnerProduct. The second one is designed for CIFAR10[4] database, it has 8 layers: 3 Convolutions, 3 Pooling, and 2 InnerProduct. Additionally, both networks have a SoftMax layer with loss, an Accuracy layer, and at least 1 ReLU function.

We have successfully run both neural networks in CPU and GPU using PHAST. We obtain the same results as the original CPU Caffe's implementation. To check that the results are the same, we have used the set of inputs that Caffe provides, then we have checked several parameters such as the output of the network, the accuracy, the loss, and some intermediate matrices. Despite we have noticed that the accuracy and the loss are enough to validate the results, we continue using the intermediate matrices and the outputs to be sure that all was working properly. Now, we are able to choose between the CPU or the GPU version depending on the makefile we use.

5.2 Experiences

During the development, we have confirmed that applying a high performance programming model is not much different from the programming methodologies we are used to. There are some exceptions, such as the Convolution block and the Accuracy block, where rethinking the algorithm is necessary, but despite that, most of the high performance models are based in well-known approaches, as we have mentioned before.

5.3 Caffe Tests

Once our testbench is running as intended, we decide to test how accurate is our implementation to the original implementation. Therefore, using Caffe tests files, we have checked all the blocks we have ported to PHAST that have a test (Table 3). We notice that all the functionality we have ported is working, and tests only fail on the unimplemented functionality.

Block	Passed	Not Passed	Total	%Passed
Convolution	3	12	15	20
Pooling	11	0	11	100
InnerProduct	9	0	9	100
SoftMax	4	0	4	100
SoftMax Loss	4	0	4	100
Accuracy	9	3	12	75

Table 3: Each available test for the block we have modify in single precision floating point numbers and the number of tests passed and not passed.

[2] https://github.com/BVLC/caffe
[3] More information: http://yann.lecun.com/exdb/mnist/
[4] More information: https://www.cs.toronto.edu/~kriz/cifar.html

5.4 Performance

Since this is a ongoing project, we are not ready to give performance results because our current tests are very small. But, compared to Caffe's original CPU code using openblas, our code was running 10 times slower. After some little changes, we improved it by a factor of 2x. But these are not final results and they can be improved by finding and fixing the current hotspots. Also, we are trying to reduce the number of copies made at each operation, by adopting part of the code to use read-only data, or maintain some structures as long as possible. It is like modernizing our modernized code.

6. Conclusions & Future Work

In this work, we aim to find the most relevant high-performance portable programming models available and classify them into a table to be able to easily use the one that fits our needs. But, to actually understand how these models work, we selected one of them, PHAST, and applied it to a real case. Since deep learning and neural networks are very important today, we selected one of the firsts deep learning framework (Caffe) and adapted it to use PHAST.

From our high performance portable programming model classification, we found that there are many available models, each one with its own characteristics. Nowadays, all is merged and making a strict classification is more subjective that it should be, but we think that we have stablish a baseline that could be used to classify the majority of the available models.

In this work, we have seen that PHAST is suitable to be used in real applications, it is easy to use and remembers well-established highly-expressive techniques.

There are several works left behind due to lack of time, among them we have:

- Explore the remainder programming models and devices: There are many models we have left behind, exploring them can open new characteristics in the classification table.
- Complete the port of Caffe to PHAST: We plan to release Caffe freely when the port is finished. We think that the information gathers during this port is very useful for improving PHAST.
- Extend our work to another device: Increasing the portability is one of our main goals in this work, another compatible device goes to give us valuable information when developing using these programming models.
- Improve the performance: Our firsts results show that checking the code it can be greatly improved with no so much effort.

Acknowledgments

We are glad to thank Sandro Bartolini and Biagio Peccerillo, from University of Siena, for granting us the demo version of the PHAST library and all the support given during this period, allowing us to develop this project as a joint collaboration.

References

[1] Antonio Gonzalez. Trends in processor architecture. *CoRR*, abs/1801.05215, 2018.

[2] John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Commun. ACM*, 62(2):48–60, January 2019.

[3] Intel. Intel's 'one api' project delivers unified programming model across diverse architectures. https://newsroom.intel.com/news/intels-one-api-project-delivers-unified-programming-model.

[4] JR Neely. Doe centers of excellence performance portability meeting. Technical Report LLNL-TR-700962, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2016.

[5] Christian-A. Bohn. Kohonen feature mapping through graphics hardware. In *In Proceedings of Int. Conf. on Compu. Intelligence and Neurosciences*, pages 64–67, 1998.

[6] Vaughn Betz. FPGA Architecture for the Challenge. http://www.eecg.toronto.edu/~vaughn/challenge/fpga_arch.html.

[7] V. Antinyan, A. B. Sandberg, and M. Staron. A pragmatic view on code complexity management. *Computer*, 52(2):14–22, Feb 2019.

[8] Christof Angermueller, Tanel Pärnamaa, Leopold Parts, and Oliver Stegle. Deep learning for computational biology. *Molecular Systems Biology*, 12(7):878, 2016.

[9] W. G. Hatcher and W. Yu. A survey of deep learning: Platforms, applications and emerging research trends. *IEEE Access*, 6:24411–24432, 2018.

[10] Henry Kasim, Verdi March, Rita Zhang, and Simon See. Survey on parallel programming model. In *Proceedings of the IFIP International Conference on Network and Parallel Computing*, NPC '08, pages 266–275, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] Tim Lewis. OpenMP FAQ. https://www.openmp.org/about/openmp-faq/.

[12] OpenACC.org organization. Homepage | OpenACC. https://www.openacc.org/.

[13] Barcelona Supercomputing Center. The OmpSs Programming Model | Programming Models @ BSC. https://pm.bsc.es/ompss.

[14] The Khronos Group Inc. Opencl overview. https://www.khronos.org/opencl/.

[15] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202 – 3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[16] Biagio Peccerillo and Sandro Bartolini. PHAST library - enabling single-source and high performance code for gpus and multi-cores. In *2017 International Conference on High Performance Computing & Simulation, HPCS 2017, Genoa, Italy, July 17-21, 2017*, pages 715–718, 2017.

[17] Biagio Peccerillo, Sandro Bartolini, and Çetin Kaya Koç. Parallel bitsliced AES through PHAST: a single-source high-performance library for multi-cores and gpus. *J. Cryptographic Engineering*, 9(2):159–171, 2019.

[18] Biagio Peccerillo and Sandro Bartolini. PHAST - A portable high-level modern C++ programming library for gpus and multi-cores. *IEEE Trans. Parallel Distrib. Syst.*, 30(1):174–189, 2019.

[19] Biagio Peccerillo and Sandro Bartolini. Task-dag support in single-source PHAST library: Enabling flexible assignment of tasks to cpus and gpus in heterogeneous architectures. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM@PPoPP 2019, Washington, DC, USA, February 17, 2019*, pages 91–100, 2019.

[20] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[21] Eduardo José Gómez-Hernandez and José Manuel García. Preparing and managing and hpc cluster: Lessons learned. In *Jornadas Sarteco 2019*, Cáceres (Spain), September 2019.

Acronyms

ANN Artificial Neural Network. 2

API Application Programming Interface. 4, 5

ASIC Application-Specific Integrated Circuit. 1, 2

BSC Barcelona Supercomputing Center. 5

DNN Deep Neural Network. 1, 10

DSA Domain Specific Architecture. 1

DSL Domain Specific Language. 2

FPGA Field Programmable Gate Array. 1, 2, 6

GeMM General Matrix Multiplication. 12, 14

HPC High Performance Computing. 4, 5

ICD Installable Client Driver. 5

PGAS Partitioned Global Address Space. 7

PHAST Parallel Heterogeneous-Architecture STL-like Template. 5, 6, 10–12, 19–22, 24

SIMD Single Instruction Multiple Data. 2

SIMT Single Instruction Multiple Threads. 2

STL Standar Template Library. 5–7, 10