POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione

Corso di laurea in Ingegneria Informatica

Title

Relatore: Prof.

Tesi di laurea di
Ghitti Marco
Matr. 893986

Anno Accademico 2019/2020

Abstract

Abstract content

Ringraziamenti

Ringraziamenti

Contents

Chapter 1

Introduction

Intro content

Chapter 2

Definitions

Intro

2.1 Conclusions

Chapter 3

State of the art

In this chapter we explore previously existing implementations of developement stacks used to produce stream computing applications and deep learning accelerators.

3.1 Chapter structure

Section 3.2 introduce StreamIt, a research compilation infrastructure to deploy optimized streaming applications from a high level definition. Section 3.3 describe VitisAI, a proprietary end-to-end developement stack to deploy DL models on Xilinx HW. Section 3.4 describe Halide, a open source language and compilation infrastructure to produce optimized code for image and tensor processing.

3.2 StreamIt

Writing a program in the context of a streaming application while satisfying a set of requirements in terms of throughput and latency require a careful management of memory transfer and HW resources. That problem can be solved by writing custom code that directly manage these aspects but the development process is error prone and slow.

StreamIt is a programming language and research compilation infrastructure for streaming systems. It is an MIT project started in the year 2000 with the goal of executing high-performance streaming applications while increasing the programmer productivity through stream-specific abstractions. The compilation infrastructure take as input a program written with the StreamIt programming language, perform stream specific optimizations and produce the optimized final program to be deployed on the target architecture.

3.2.1 Streaming Application Domain

The application domain of a streaming program is characterized by a set of assumptions that can be leveraged by a developement stack. By leveraging these characteristics a streaming programming language can increase performances while providing a set of abstractions that make easier for the programmer to write and maintain the source code of the application.

3.2.1.1 Large stream of data

The execution model of a general program is fundamentally different than the model of a streaming application: A common program is usually invoked with a finite input set that is kept until termination; instead, in the context of a streaming application the input is virtually infinite and is a sequence of data items where each element require a finite amount of operations before being discarded. A stream program can run indefinitely, waiting for items to be processed from an external input source.

3.2.1.2 Independent stream filters

To exploit the parallelism of a streaming application the programmer must specify a set of interconnected filters. The whole set of interconnected filters is referred as Stream Graph and contain information about communication among filters and how to exploit the intrinsic parallelism of the specific application.

Each filter define a self contained transformation on data items that can be executed independently from all others, while the connections among filters define the computation pattern, and thus the kind of parallelism that can be exploited.

3.2.1.3 Stable computation pattern

To be able to exploit the parallelism among filters and all other advantages of the streaming domain, the stream graph is assumed to be constant during steady state execution. The domain allow occasional modifications to the computation pattern but can lead to a significant performance degradation if used improperly by changing it's structure frequently.

3.2.1.4 Occasional out-of-stream communication

Even if in a streaming application most of the communication volume is dedicated to data items, there is still need to send control messages on irregular and infrequent basis. The domain allow for infrequent control messages from host to filter and between filter to react to specific situations. As example the host might want to change a filter's parameter during runtime execution.

3.2.1.5 High performance expectations

The streaming application is going to be deployed in a real system that is going to have its own application specific real time constraints (Throughput, Latency, Power consumption, Memory, ...).

3.2.2 Streamit Program

The StreamIt programming language is a high level specification of the application; it defines what the final application should do in terms of item transformations and communication without defining how the final code will perform these operations.

A StreamIt program is composed by 4 main blocks defining the model of computation:

Filter Basic unit of computation of a StreamIt program.

Contain two main functions, work and init; the work function define the transformation performed on input data items whenever the actor is fired, the init function define the initialization procedure of the filter needed to initialize the first invocation. A filter can be stateful or stateless and must always define at compile time the amount of push, pop and peak operations that are performed at each invocation of the work function.

In a streaming application each filter must represent a self contained transformation on data items and the only way for a filter to communicate with other filters is through FIFO channels. StreamIt apply an additional constrain on the communication among filters, each filter have only one input and one output channels; this allow the compiler to further optimize the final code and extract more parallelism from the application.

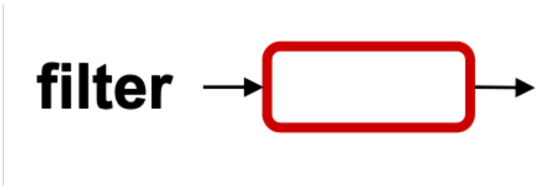

Figure 3.1: StreamIt Filter

Pipeline High level abstraction of a software pipeline among filters.

Allow the programmer to specify a set of filters that are connected through a pipeline connection; this allow the compiler to exploit pipeline parallelism.

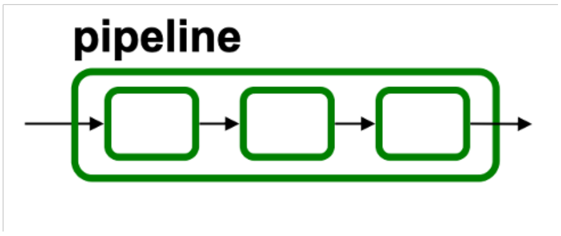

Figure 3.2: StreamIt Pipeline

SplitJoin Allow the specification of multiple parallel and independent computation paths that diverge from a common splitter and merge into a common joiner.

Can be used to extract both task level and data level parallelism. If all items are sent to all filters and and each filter perform a different transformation on the data item it extract task level parallelism; if the incoming data stream is load balanced among the different filters and all filters perform the same operation it extract data level parallelism. The behavior of the SplitJoin construct is thus defined by the behavior of the splitter and the joiner blocks.

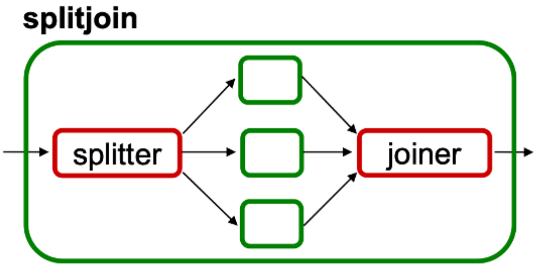

Figure 3.3: StreamIt SplitJoin

Feedback loop Allow the creation of loops in the computation.

Wrap a main body filter in a feedback loop; the data items that enter the section are merged with the feedback connection through a joiner block while the items that exit the body filter are splitted between the feedback connection and the next filter according to a splitter block.

The programmer can also define a computation path along the feedback connection, this allow transformations on items that are sent along the feedback connection.

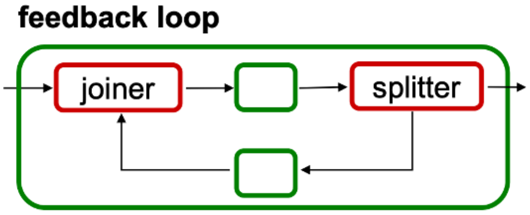

Figure 3.4: StreamIt Feedback Loop

3.2.3 Compilation Infrastructure

The StreamIt compilation infrastructure is designed to start from the high level abstraction of a StreamIt program and produce the final application to be deployed. The programmer describe the application in terms of which transformations need to be applied to each item and how different transformations are connected; then, the StreamIt compiler automatically optimize the StreamIt program for the streaming context and produce the final optimized code for the target back end.

3.2.3.1 Stream Graph Scheduling

One of the most important pieces of information that is compiled from the StreamIt code is the Stream Graph of the application. The Stream graph is an internal representation of the streaming application that represent how different actors interact; the StreamIt compiler use it to retain information about how different filters interact and the amount of parallelism that can be exploited among filters.

The goal of the Stream Graph Scheduling phase is to find a steady state schedule that allow the program to process the stream of input items while maintaining constant the number of live items on each communication channel.

3.2.3.2 Partitioning

Given the Stream Graph and the steady state schedule of the application the compiler need to adapt the computation to the granularity of the target architecture. The Stream Graph compiled from the StreamIt program do not take into consideration the number of processing elements. Mapping directly the original nodes to physical processors would lead to unoptimized results; in case of a too coarse grained graph the program would not use some processing elements, in case of a too fine grained graph the final program would require unnecessary memory communication. The Streamit compiler use a heuristic to decide the number of nodes to be mapped to physical elements, the graph is adapted by producing a new graph with a number of nodes that match the number of processing elements of the target architecture. In order to create the new stream graph, the compiler must implement Fusion and Fission operations. Fusion merge two filters into one and Fission does the opposite operation. Fusion and

Fission are not simple operations, the compiler must take into consideration the type of connections between filters in order to apply these operations efficiently.

To partition the stream graph in a set of balanced filters, the compiler must be able to estimate the amount of work performed by the work function of each filter. StreamIt estimate the work through static inspection. Since the number of iterations of each loop is known at compile time, the compiler can estimate the total amount of work by unrolling each loop and considering the number and the type of instructions executed at each fire.

The partitioning phase can then be performed automatically with a simple greedy algorithm that split the most demanding filters and merge the least demanding ones.

3.2.3.3 Layout

Given the work balanced nodes from the partitioning phase, the StreamIt compiler need to assign each node to a physical node on the target architecture while minimizing the communication overhead between nodes. To optimize the communication and synchronization overhead a back end dependent cost function must be defined and optimized. The Cost function should accurately measure the added communication and synchronization generated by mapping the work balanced Stream Graph to the communication model of the target.

3.2.3.4 Communication scheduler

When the layout phase has mapped filters to physical computation nodes the only remaining abstraction that need to be mapped to the target architecture is the communication queues between filters. The communication scheduler maps the infinite FIFO abstraction to the limited resources of the target architecture while avoiding deadlocks and starvation.

Once the Communication Scheduler phase is finished the StreamIt compiler is ready to generate the final code for the target back-end.

3.3 VitisAI

VitisAI is a development stack for AI inference on Xilinx's HW; it is a complete development stack to optimize and deploy a pretrained model on FPGA without the need of knowing any implementation detail about the underlying HW.

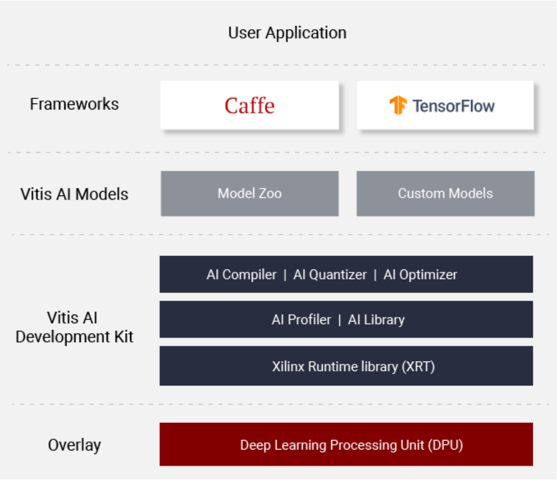

Figure 3.5: Vitis AI stack

The programmer can feed to the VitisAI stack an abstract model in ONNX format or defined with one of the most common frameworks. The stack automatically optimize the model for the target architecture with few optimization tools and produce the final program to be deployed on the target Xilinx architecture.

3.3.1 Optimization tools

The stack have multiple free tools that can be used to perform optimizations on the input model; these optimization tools can also be used as preprocessing steps on an abstract model before feeding it to a different stack.

AIOptimizer Perform pruning on the input model, it automatically prune the connections between artificial neurons that less affect the accuracy of the final model.

AIQuantizer Automatically convert the floating-point precision of the input model's weights to a fixed-point representation. This is especially useful on FPGA since it is possible to use a completely custom number of bits without being restricted by the HW design.

AICompiler The quantizer perform HW depentant optimizations on the quantized model, it maps the model to optimized DPU instructions while performing optimizations like node partitioning and instruction scheduling.

3.4 Halide

With the increasing size of deep learning models and the widening gap between processing power and memory bandwidth it is increasingly difficult to train and deploy new and more computationally demanding DL models. A possible solution might be to write custom code to exploit the right trade-off among data locality, recomputation and parallelism but the process is slow, prone to errors, do not allow for fast prototyping of different scheduling solutions and the programmer has a constant focus on implementation details. Halide is a language and compiler to write high performance tensor processing pipelines for multiple target platforms with the goal of solving the emerging problems of image processing pipelines and deep learning applications; the Halide domain-specific language allow to separately define the algorithm that need to be executed, as functions of the input tensor, and the scheduling strategy to be applied for the specific algorithm on the target architecture. Decoupling the definition of a Halide program in two different components allow the exploration of a large number of scheduling strategies with far fewer lines of code and without affecting the correctness of the program.

3.4.1 Algorithm definition

The Halide algorithm describe what the programmer want to compute without defining how the final result should be computed; this guarantee the correctness of the final program while experimenting with different scheduling strategies and leave to the compiler the responsibility of optimizing the final code. The Halide DSL represent the algorithm definition as an Abstract Syntax Tree of 3 main components: Variables, expressions and functions.

Each function is defined over a set of variables and it's value is computed as an expression of other functions. As example we can use the 3x3 blur filter definition:

$$\mathsf{blur\_x}(x, y) = (\mathsf{input}(x - 1, y) + \mathsf{input}(x, y) + \mathsf{input}(x + 1, y))/3;$$

$$\mathsf{blur\_y}(x, y) = (\mathsf{blur\_x}(x, y - 1) + \mathsf{blur\_x}(x, y) + \mathsf{blur\_x}(x, y + 1))/3;$$

The pipeline is composed by 2 functions, blur_x and blur_y; both functions are defined over two different dimensions that define the x and y coordinates of the image. The value of each function is an expression composed as the mean of the 3 nearest pixels in both directions.

3.4.2 Schedule definition

Now that the algorithm is defined Halide need to know how to compute the results in terms of storage and compute granularity. Storage granularity and compute granularity refer to how big the buffers between stages need to be and how frequently consumers and producers are interleaved.

One possible naive solution might be to compute and store each stage of the pipeline one at a time and separately from from others; this is actually the approach of the most common frameworks, after each stage the results are stored in main memory before proceeding with the next stage of the pipeline. This approach suffer of the drawback of having poor locality since every tensor must be computed and stored in its entirety and is unlikely to fit into a low level memory. A second possible approach is to compute only the last stage of the pipeline without storing intermediate results; this maximize locality by recomputing every value each time but require an amount of work that grow exponentially with the number of stages in the pipeline. A third approach is to allocate large buffers while performing fine-grained computation, this allow for solving both problems by reusing all previously computed results while exploiting locality. The problem of this last approach is the introduction of dependencies into computed results thus reducing the amount of parallelism that can be exploited from different chunks of work. The programmer should find the right trade-off by using store_at and compute_at directives and by performing tiling, unrolling and reordering operations on different pipeline stages.

An other important decision to be made by the schedule is how to use HW specific features to increase the pipeline performances (such as multiple cores, vectorized instructions, GPUs, and so on..). The schedule allow the programmer to combine in different ways different features for each variable and function; this make easier for the programmer to explore different acceleration strategies without touching the code that defines the algorithm.

Finding the right schedule to manage storage and compute granularity while exploiting the HW specific features is not a trivial task, especially when considering complex pipelines for complex image filters and DL applications. For that reason Halide have a built in autoscheduler to automatically find a schedule; the current version require extensive optimization to find a schedule that is as good as one manually created by a expert programmer and can be used to find a baseline in a small amount of time.

To make an example we can take again the 3x3 blur filter:

```
blur_y.tile(x, y, xi, yi, 256, 32).vectorize(xi, 8).parallel(y);

blur_x.compute_at(blur_y, x).vectorize(x, 8);
```

The schedule use both parallelization and vectorization operations, tile the loop nest and compute blur_x for each unique value of the variable x of blur_y. The programmer only need to specify how the computation is performed, Halide takes case of all implementation details and generate efficient code.

3.4.3 Halide Compiler

The Halide compiler take as input both the algorithm definition and the schedule and produce a cross platform internal representation. The internal representation is then optimized by performing target independent optimizations and compiled into the final code of the target back-end.

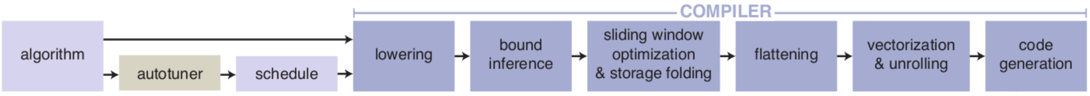

Figure 3.6: Halide stack

The first step ot the Halide compilation infrastructure is to lower the algorithm and schedule definition to a set of loop nests and buffer allocations. Each loop is labeled as serial, parallel, unrolled or vectorized and loop bounds are left as symbolic expressions of the required region of the output function. The lowering process start from the output function and recursively proceeds backward toward input functions. The process is complete once all functions has been lowered.

Once the Lowering process has been completed the Halide compiler stack have a complete representation of the Halide program as a loop nest operating on multidimensional tensor objects. The compiler know the set of loops and the number of dimensions that are required to compute the final result but do not know the extend of such dimensions. The Bound Inference procedure is a two step process. The first step calculate the extends of each tensor; the compiler propagate backward the information about the size of the output tensor and calculate recursively the extents of intermediate bounds. The second step use this information to calculate the extent of each loop.

By knowing the extend of each dimension, Halide can traverse the loop nest to seek for sliding window and storage folding optimizations. These two optimizations passes are used to leverage the storage granularity allowed in the current schedule and reuse data already computed in previous iterations.

The optimized sequence of loop nests with multi-dimensional store and load is then lowered to a single-dimensional strided representation. Each multi-dimensional tensor is converted into a single-dimensional representation and each load and store operation on tensor is represented as a stride access on the corresponding single-dimensional buffer.

The final step before invoking the target back-end to compile the Halide IR into the final code is to remove unrolled and vectorized loops. Unrolled loops can be simplified by performing the loop unrolling operation directly on the halide IR while vectorized loops can be removed by replacing them with dense memory accesses and operations using ramps to represent strided sets of indexes on memory buffers.

3.4.4 Deep learning applications

DL models are composed by a sequence of layers that can be expressed as an Halide algorithm by composing functions and expressions. As for other most common DL frameworks Halide automatically differentiate feed forward models; the programmer only need to define the model structure and the Automatic Differentiation return a new function that calculate the requested derivatives. The Halide Automatic Differentiation system must take into consideration some optimizations to exploit as much parallelism as possible while performing gradient propagation on scatter-gather operations; since gather operations become become scatter operations when differentiated and scatter operations are not easily parallelizable, the Automatic Differentiation system automatically convert scatter operations back to gather operations to allow a higher degree of exploitable parallelism in the final gradient function.

Since the increasing complexity of modern DL models are making impractical the definition by hand of a model as an Halide algorithm the programmer can provide an abstract definition with a cross-platform format like ONNX. Providing an abstract model avoid to the programmer the tedious and error prone work of defining a model as composition of functions. Halide automatically convert the ONNX model to an Halide algorithm and return a function that can be used to perform inference or differentiated through automatic differentiation to calculate the gradient for the backpropagation algorithm.

3.5 Conclusions

This chapter described the State of the art compilations stacks used for deploying deep learning models. Section 3.2 introduced the problem of producing efficient code in the context of streaming application; we explored the application domain of a streaming application and how to optimize the computation of multiple independent tasks. Section 3.3 presented a possible proprietary solution to deploy DL models. Section 3.4 explored the challenges introduced when writing high-performance code on multiple interconnected transformations; in particular Halide introduced the problems of how to optimize the computation of multiple interconnected filters on different HW architectures while producing code able to leverage the memory organization.

Chapter 4

From ONNX model to Bambu HLS

Intro

4.1 Motivations

Since the state of the art DL models are becoming incresingly more complex and demanding in terms of computational power, using CPUs and GPGPUs is becoming a less appealing approach. When a model is deployed the system is required to meet a set of latency, throughput and memory consumption constraints; CPUs are not suitable for high throughput applications and power hungry GPUs have high latency due to memory transfers from and to the host. The use specialized HW meet all 3 requirements with high throughput, low latency and memory consumption; but such a solution is not flexible and not able to express new layers that might have not been yet invented.

A different approach might be to use Field Programmable Gate Arrays (FPGA); the use of reprogrammable HW allow the possibility to meet all system requirements while being able to use the system to deply different models. The problem with FPGAs is the time required to deploy an FPGA's application; deploying a correct and optimized bitstream is not trivial and require a high design time. To reduce the deployment time of a new application High Level Syntesys tools are used; these tools take as input the abstract representation of a program, in this case a DL model, and produce as ouput the optimized bitstream to be deployed. The final result do not have the same performance of a manually optimized solution but the results are comparable and allow to perform fast prototyping and the possibilty of using low level HW for non experts.

The integration of a Bambu back-end in Halide allow the programmer to easily deploy a DL model on FPGA systems; the decoupoling of the shedule from the algorithm definition allow the programmer to leverage the possibility to deploy multiple models in a short ammount of time to find the schedule that better fit the available memory hirearchy.

4.2 Design flow

The goal of the proposed design flow is to leverage PandA - bambu and the Halide infrastructure to deploy a Deep Learning model on FPGA while being able to optimize the sequence of scheduled operations for the target architecture.

The starting point is a Deep Neural Network model provided from one of the most used frameworks. Since each framework represent the computational graph in different ways we need a common representation to provide a unique entry point to the design flow. The Open Neural Network eXchange specification (ONNX) has been chosen as common representation which also allows the portability of computational graphs between different frameworks.

Since the output models from most framework is unoptimized, the programmer can optionally use the VitisAI tools to perform optimizations on the model to be deployed to increase performances and eventually meet the system requirements.

Given a ONNX model as input the Halide framework have a built-in tool to convert it to a Halide generator, the programmer can then try different schedules and find the ont that best fit the final application's requirements.

Finally, the scheduled Halide generator is optimized and compiled to a Halide Intermediate Representation before being processed by the Bambu back-end that we added to provide support for FPGA platforms.

4.3 The Bambu back-end

Codegen description

4.4 Conclusions

Conclusions

List of Figures

List of Tables

Bibliography