Approximated Computing-based Methods for Hardware Resources Reduction Targeting Heterogeneous Systems

Hugo Miomandre

DOCTORAT BRETAGNE LOIRE / MATHSTIC

INSA RENNES

THÈSE DE DOCTORAT DE

L'INSTITUT NATIONAL DES SCIENCES APPLIQUÉES DE RENNES
COMUE UNIVERSITÉ BRETAGNE LOIRE

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : Signal, Image, Vision

Par
Hugo MIOMANDRE

Approximated Computing-based Methods for Memory Resources Reduction Targeting Heterogeneous Systems

Thèse présentée et soutenue à Rennes, le 05 Décembre 2022
Unité de recherche : IETR
Thèse N° : 22ISAR 35 / D22 - 35

Rapporteurs avant soutenance :

Alberto BOSIO	Professeur des Universités, INL, École Centrale Lyon
Christophe JÉGO	Professeur des Universités, IMS, Bordeaux INP

Composition du Jury :

Président :	Daniel CHILLET	Professeur des Universités, IRISA, Enssat
Examinateurs :	Francesca PALUMBO	Associate Professor, UNISS
	Nicolas GAC	Maître de Conférence (HdR), L2S, CentraleSupélec
	Alberto BOSIO	Professeur des Universités, INL, École Centrale Lyon
	Christophe JÉGO	Professeur des Universités, IMS, Bordeaux INP
Dir. de thèse :	Jean-François NEZAN	Professeur des Universités, IETR, INSA Rennes
Encadrant :	Daniel MÉNARD	Professeur des Universités, IETR, INSA Rennes

Table of Contents

Acknowledgements

I would like to thank my thesis directors, Jean-François Nezan et Daniel Ménard for their guidance and support during the 3 years of this PhD, my reviewers Alberto Bosio and Christophe Jégo for taking the time to read this manuscript, and the members of the jury Daniel Chillet, Francesca Palumbo and Nicolas Gac.

I would also like to thank Karol Desnos and Kevin Martin for the internship opportunity that ultimately led me down this path.

Moreover, thank to the SARS-CoV-2 which made me waste roughly a year of work and productivity.

Finally, I would like to thank everyone from the EII department and the VAADER research team, both for the three years of my engineering degree and for the three years of my PhD and INSA Rennes, and to the occupants of office 214 for the everyday stupidity.

CHAPTER 1

Introduction

Since the last few decades, the volume of processed data has been growing in conjunction with the increase in computational power of High-Performance Computing (HPC) systems, as well as the generalization of connected embedded systems in all forms, from Internet of Things (IoT) devices to smartphones.

Embedded systems are computer systems purposely designed for specific applications, as stand-alones such as autonomous lawnmower, digital camera or home appliances, or as part of bigger assemblies such as the multiple subsystems in vehicles.

The increase in computing resources also goes in pair with the augmentation and complexification of memory related systems. Both on personal computers and smartphones, the amount of available RAM doubles every 2 to 3 years[1]. In a computer system, memory is used to store data, to store program instruction, to save temporary values, and for synchronization.

An infinitely fast processor can only operate as fast as data are transmitted to and from it. The issue of avoiding this memory wall [WM95; JWN10; Ziv+17] has been known for at almost 3 decades, leading to the conception of dedicated memory-specific mechanisms. Consequently, it is evaluated that up to 80% of silicon area can be dedicated to caches, memories, memory controllers, interconnects for the sole purpose of data transmission

1. Example with smartphones: iPhone 2G (2007): 128MB, iPhone 14 Pro (2022): 6GB, Current highest-end: 18GB

inside a chip [DCD97]. Additionally, the transfer of data between memory and Processing Elements (PEs) can represent up to 62% of the energy consumed by the whole system.

As an example, in Advanced Micro Devices (AMD) Central Processing Units (CPUs) with the Zen 3 architecture, the 32MB L3 cache alone takes around 52% of a CCD[2] silicon area, but an additional 64MB can be stacked on top of it [Bur+22], for a total of 96MB, 12MB per core. Similarly, in AMD Graphics Processing Units (GPUs) with the RDNA 3 architecture, a fully enabled chip features a 300mm^2 main die accompanied by 6 individual 37mm^2 16MB Last Level Cache (LLC) dies, resulting in 43% of the silicon area dedicated solely to LLC.

1.1 General Context

The increased volume of processed data impacts computing systems on every scale, from edge computing [ZC20] to HPC. It also affects the consumer scale, with use-cases such as image processing on smartphones [Mor+21], where the amount of data processing have skyrocketed both in hardware and in software, with the improvement of camera sensors and screens along with the onboard processing power.

Large Scale Computations

The increase of data processing capabilities enables the computation of complex datasets for specific purposes [Bra19]. The most common everyday example concerns the collection and exploitation of user-generated data by companies of the information technology industry. Often designated with acronyms such as GAFAM[3] or FAANG[4], these companies rely on large scale data processing for user profiling and recommendations [AT05].

Other application examples of intense data processing are scientific research projects such as the Large Hadron Collider (LHC) or the Square Kilometre Array (SKA) radio telescope.

The LHC is largest and highest-energy particle collider, designed for research in particle physics, at the European Organization for Nuclear Research (CERN). It consists in a 27 kilometres ring, with detectors specialized for specific phenomena. The LHC contains about 150 million sensors delivering data 40 million times per second, filtered to a few

2. Core Complex Die, containing up to 8 cores.
3. GAFAM: Google, Apple, Facebook, Amazon, Microsoft.
4. FAANG: Facebook, Amazon, Apple, Netflix, Google.

thousands collision per seconds, for a yearly production of 88 Petabytes. The power consumption for the LHC alone is about 90 Megawatts, and 200 Megawatts for all of CERN. The LHCb detector alone, upgraded in 2022, can process around 4 Terabytes per seconds of raw data, selecting 10 Gigabytes per seconds of interesting collision data.

The SKA radio telescope is an international effort to build the world's largest radio-telescope, with a final collecting area of around one square kilometre. The SKA is expected to start its operations with hundreds of parabolic dishes and tens of thousands antennae in South Africa and Australia, producing more than 30 terabits per seconds of raw data.

Machine Learning

Machine learning have found applications in various domains, such as image/video processing (classification [KSH12], video compression [Tis+20], image upsampling [Bas+21], autonomous driving [Kir+21; Gri+20]), language processing [OMK18], network intrusion detection [Sou+22; De +21]...

The gain of popularity around machine learning during the last decade as given birth to extremely large deep learning models. These models can be composed of tens to hundreds of billions of parameters and trained on large dataset.

As an example, a few large language processing models can be cited: Generative Pre-trained Transformer 3 (GPT-3) [Bro+20] with 175 billion parameters trained on a 45 Terabytes dataset, DeepMind Gopher [Rae+21] with 280 billion parameters trained on MassiveText, a 10.5 Terabytes dataset, Microsoft/Nvidia Megatron-Turing [Smi+22] with 530 billion, or Google PaLM [Cho+22] with 540 billion parameters.

A currently trendy application of machine learning is Text-to-Image generation, with examples such as: DALL-E [Ram+21] (derived from GPT-3) with 12 billions parameters trained on a 250 million image-text pairs dataset, Midjourney, Google Imagen [Sah+22] with 2 billion parameters, Parti [Yu+22] with 20 billion, or Stable Diffusion [Rom+21].

These models require a lot of hardware resources for training and inference, with the power consumption that goes along with it. Specific optimizations such as memory optimizations become increasingly necessary to facilitate the deployment and embedding of these models by reducing the hardware resources requirements as well as the associated memory consumption.

1.2 Scope of this Thesis and Contributions

Memory-related limitations are a major consideration during design and deployment of data processing applications on computer systems, either in terms of capacity, energy, transmission or area.

The Approximate Computing (AxC) paradigm [HO13] has emerged as a way to improve the energy efficiency and/or the performance of computer systems, by trading-off result accuracy in application where data integrity is not critical, such as dropping a frame during the encoding or decoding of a video stream. AxC techniques can be segmented in 3 categories, whether they impact the data used during computations, the nature of the computation itself, or parameters of the hardware. These techniques tend to be difficult to implement but are susceptible to reduce the strain on the memory systems of a computing platforms.

Another method to impact the requirement related to memory systems is the use of specific Models of Computation (MoCs). Describing a data processing application with specific MoCs can highlight opportunities for memory optimizations. Dataflow-based MoCs have the particularity of bringing to the forefront the handling of memory, both in terms of allocation and transit, highlighting opportunities for optimization.

The objective of this thesis is to develop new techniques to enable reductions of memory footprint requirements of data processing applications using the AxC paradigm alongside dataflow-based representations.

The main contributions of this thesis are:

1. A study of the impact of bit-width and representation of the storage memory of applications on the output quality. Data storage is considered using an arbitrary number of bits, coupled with a customizable data representation to minimize the loss of quality. This contribution has been published partly in [Mio+20].

2. A method to convert and store in memory of data with an arbitrary bit-width for CPU-based platforms. This method handles the memory operations to store unaligned data of unconventional bit-width into concatenated segments. This contribution has been published partly in [Mio+20].

3. A method to efficiently insert and extract data from packet for Field-Programmable Gate Array (FPGA) storage resources optimization. This method is able, from a dataflow representation, right after the buffer-sizing process, to find optimal pack-

ing ratio to reduce the requirement on embedded hardware storage resources. This contribution has not been published yet.

4. A Design Space Exploration (DSE) method based on AxC to reduce the memory footprint of an application by reducing the bit-width used for internal data storage. This method finds an acceptable association between data representation and storage bit-width for a set of memory buffer of an application. The memory footprint is minimized while respecting a determined quality constraint. This contribution has been published in [MNM22].

Most of the contributions of this thesis as been integrated into the Parallel and Real-time Embedded Executives Scheduling Method (PREESM) application development framework.

1.3 Outline

This thesis is organized in to parts: Part I presents the general context and motivation of this thesis, and Part II introduces and evaluates the contributions of this thesis.

In Part I, Chapter 2 provides a general overview of AxC techniques and which aspect of an application they affect. Chapter 3 presents the main programming model paradigms for parallel computations, along with the concept of dataflow MoC used in this thesis. Finally, Chapter 4 details the applications used to evaluate the contributions of this thesis.

In Part II, Chapter 5 goes over the general concept and impact on quality of the AxC contributions of this thesis. Chapter 6 provides guidelines one the implementation of these contributions. Chapter 7 presents a DSE method for efficient parametrization of the AxC techniques presented in previous chapters. Finally, Chapter 8 concludes this work and proposes potential future research paths.

Part I

Background

CHAPTER 2

Approximate Computing

Introduction

An objective of this thesis is to design techniques based on the Approximate Computing (AxC) paradigm. For this purpose, this chapter introduces the concept of AxC and presents a non-exhaustive list of diverse existing AxC techniques. An AxC technique is a technique which, when applied on a system (hardware and/or software), will lead to the production of an inaccurate yet usable output. The compensation for this inaccuracy is a reduced set of resources (processing power, memory footprint, energy consumption, silicon area, ...) or an increase in performance (higher throughput, lower latency, ...) depending on the needs and the AxC techniques used [BMS22].

These AxC techniques can be segmented in 3 groups depending on the way they affect the system. First, *Computation Level* AxC techniques modifying the nature of the computations are presented in Section 2.1.1. Then, *Hardware Level* AxC techniques relying on specific hardware behaviour or modification are presented in Section 2.1.2. Finally, *Data Level* AxC techniques applied on the data themselves are presented in Section 2.1.3.

This thesis will specifically focus on *Data Level* AxC techniques aiming at reducing the memory footprint of an application. A wider summary of other AxC techniques is available in [Bon19; XMK15; Mit16; BMS22].

Section 2.1 presents a non-exhaustive overview of various AxC techniques while Section 2.2 specifically focuses on precision optimization and specific data-types.

2.1 Overview of Approximate Computing Techniques

2.1.1 Computation Level

Computation Level AxC techniques consists in modifying the algorithm itself to reduce its complexity. The two main methods are *Computation Skipping* and *Computation Approximation*, as shown in Figure 2.1.

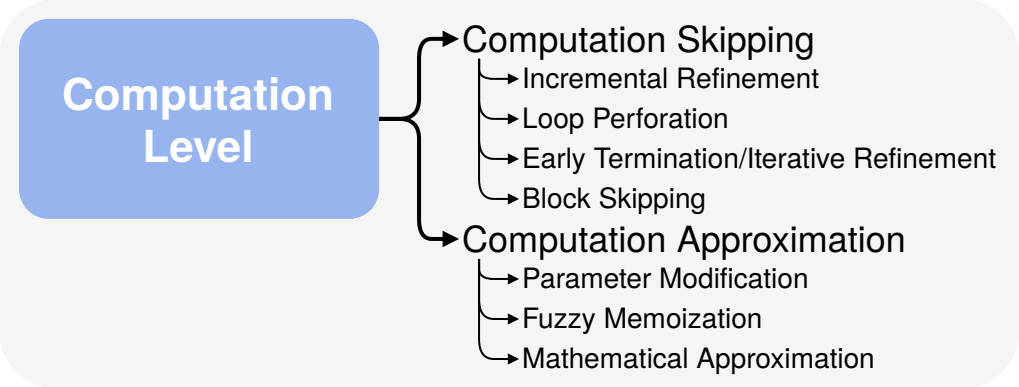


Figure 2.1: Computation-level approximate computing techniques.

Computation Skipping consists in not executing portions of the program to reduce the computation complexity. Fine-grained *Computation Skipping* can be done with *Loop Perforation*, by executing only a subset of the iterations of a loop [Sid+11; Vas+15], or by *Early Termination*, generally by prematurely ending a computation loop of a convergent iterative algorithm such as gradient descent [Meh+09; Zha+14]. A coarser-grained *Computation Skipping* is doable with *Block Skipping*, by not executing entire processing blocks of a complex application such as a video encoder/decoder [Bon+22].

Computation Approximation consists in replacing a complex computation by a simpler, faster one, producing close enough results. Two possible methods are *Fuzzy Memoization* and *Mathematical Approximation*. *Memoization* consists in storing the result of a computation in a Look-Up Table (LUT) to reuse it the next time the same computation has to be processed, simply skipping it, working like a cache memory, using input operands as ad-

dresses [Sur+15]. By itself, *Memoization* is not an AxC technique, and benefits from using it could be voided by the size of the LUT stored in memory. *Fuzzy Memoization* extends this principle by masking N Least Significant Bit (LSB) from the operands. This results in group of operands being affected the same LUT address, and thus the same result. The trade-off between output quality, processing time and the LUT memory footprint can be tweaked by adjusting the parameter N. *Mathematical Approximation* techniques are quite straightforward and are commonly used in embedded systems. It consists in implementing simpler mathematical functions in place of more sophisticated ones, as presented in [Mul20]. One way of implementing such method is to approximate a sophisticated function by a group of simpler polynomial functions, which coefficients are stored in a LUT, the N Most Significant Bit (MSB) of the operand serving as the LUT address.

Another method of mathematical approximation is to simply perform a set of computation producing an accurate enough result compared to its accurate counterpart. An example of this is the algorithm for the fast inverse square root (Listing 2.1), made popular for its use in the video game Quake III Arena for computing angles of incidence and reflection for lighting and shading. It computes an approximation of $\frac{1}{\sqrt{x}}$ with unconventional operations, such as shifting a floating-point operand and subtracting to a magic number (`0x5F3759DF`).

```
1  float Q_rsqrt(float number){
2      long i;
3      float x2, y;
4      const float threehalfs = 1.5F;
5
6      x2 = number * 0.5F;
7      y  = number;
8      i  = * ( long * ) &y;            // evil floating point
          bit level hacking
9      i  = 0x5f3759df - ( i >> 1 );    // what the fuck?
10     y  = * ( float * ) &i;
11     y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st
          iteration
12     //  y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd
          iteration, this can be removed
13
14     return y;
15 }
```

Listing 2.1: Fast inverse square root algorithm from Quake III Arena, with the original comments.

This first approximation is refined with a single iteration of the Newton's method, bringing down the maximal relative error to 0.175%. This algorithm is however completely outmatched on x86 architectures, both in terms of accuracy and performance, by the `rsqrtss` Streaming SIMD Extension (SIMD) instruction.

These methods can be applied on a wide range of applications, and can be combined with other types of AxC techniques to target specific use-cases, such as video compression encoders like the High Efficiency Video Coding (HEVC) standard [NMP16].

2.1.2 Hardware Level

Hardware Level AxC techniques consists in using circuit built with a known inaccuracy compensated by a lower energy consumption or by a faster output production. Figure 2.2 shows an overview of *Hardware Level* AxC techniques.

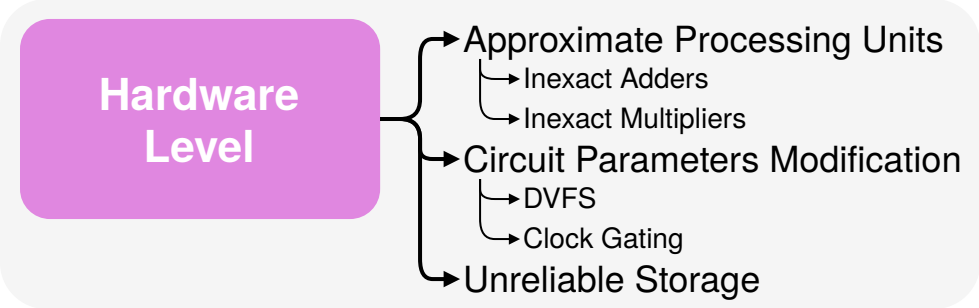


Figure 2.2: Hardware-level approximate computing techniques.

Approximate Processing Units are designed to be faster than their accurate counterparts at the price of producing a slightly inaccurate output [Gup+11; Mah+10]. *Inexact Adders* can be designed in multiple ways, the majority of which consists in reducing the length of the critical carry chain path [KGE11]. This can be done by segmenting the adder, processing the MSB and the LSB with different methods, or with speculative adders, which can speculate the carry from previous adder stages to produce faster results. *Inexact Adders* are design to reduce their hardware requirement and associated power consumption while keeping their error-rate within reasonable margins.

Circuit parameters modification consists in changing parameters such as operating voltage and frequency of a circuit. A circuit producing accurate results is only guaranteed to do so if its operating voltage and frequency requirement are satisfied. These requirements provided by the manufacturer have a margin to account for manufacturing process

variability. This means a lower-than-nominal voltage can be applied without necessarily compromising the output accuracy, resulting in a lower energy consumption, as long as it stays above the critical voltage. Around/below this critical voltage, the circuit is not guaranteed to produce inaccurate results, which may be corrected by an error-correcting circuit. Using a near/sub critical operating voltage can result in important energy saving as the dynamic power consumption of a circuit is proportional to the square of the operating voltage. However, reducing the operating voltage leads to an increase in the delay to produce an output. A similar observation can be made of operating frequency, as an increased frequency will lower the output delay but at the same time increase power consumption and critical voltage.

The use of *Unreliable Storage* consists in storing data in a memory system having a known tendency to decay. This decay may have different origins. Memory cell identified as faulty, either from factory or from wear, can be reused [Smo+13] depending on the kind of fault introduced. Operation parameters of the memory system can be adjusted to find a compromise between energy consumption, latency and accuracy. These operating parameters can be the supply voltage [Fru+15] or the refresh rate [Rah+14].

Other strategies aim at reducing the amount of memory required to store data, by using in-memory compression or by using the same memory space to store similar values. This latter method is experimented with in [Mig+15], by using the same tag for similar cache blocks for error-resilient application. This enables substantial reduction in required storage area, as well as energy consumption.

2.1.3 Data Level

Data-driven AxC techniques benefit from a reduction in either the amount of data to process, or the actual representation in computer memory. Less data to process means a faster execution time for an optimized application, and alternative data representation can lead to faster computations. Different data-driven AxC techniques are presented in Figure 2.3.

2.1.3.1 Reduction of the number of data

The basic idea associated with the reduction of the number of data is shown in Figure 2.4. This reduction can be done in the spatial or in the time domain. Both techniques are massively used by application designers in signal, image or video processing to meet real time

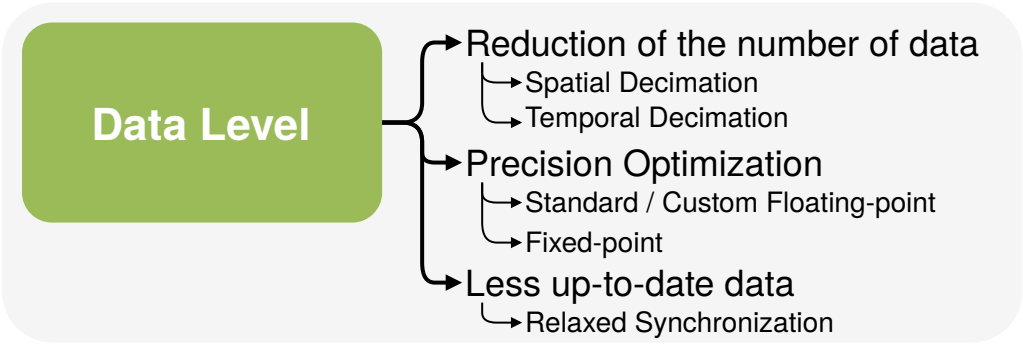
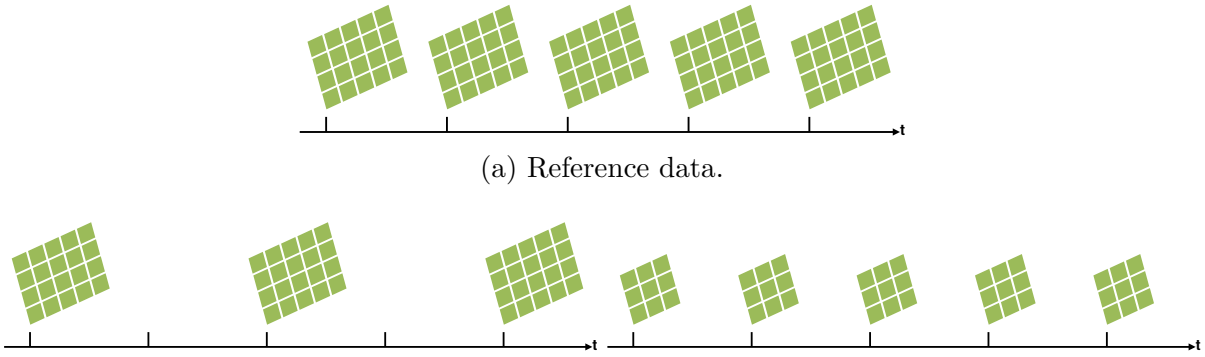


Figure 2.3: Data-level approximate computing techniques.

and complexity constraints. Concerning the time domain, reducing the sampling rate of a signal to process allows decimating it. The reduction of the sampling rate reduces the bit rate of the process which impacts the quality of the signal to process. In Figure 2.4b, data downsampling is done by doing the computations with a sampling frequency lower than the sampling frequency without data downsampling presented in Figure 2.4a, which implies that data are present less often. When it comes to the spatial domain, the signal to process can be downscaled (fitting a signal to a lower resolution) or upscaled (fitting a signal to a higher resolution). In Figure 2.4c, data downscaling is done by doing the computations on input data of smaller sizes compared with the original input data presented in Figure 2.4a.



(a) Reference data.

(b) Downsampled data

(c) Downscaled data

Figure 2.4: Reduction of the volume of data to process

In [ACN14], data downsampling is used to reduce of the amount of computations in the time domain. The proposed AxC method is demonstrated on a timing synchronization problem for Wideband-Code Division Multiple Access (W-CDMA) systems. The timing synchronization is originally done with the computation of a matched filter. The input

samples of the matched filter can be decimated by a factor D, consequently reducing by the same factor the amount of Multiply-accumulate (MAC) operations to perform. This AxC technique enable a tunable trade-off between the complexity of the algorithm and the performance, depending on the targeted Signal-to-Noise Ratio (SNR).

This method has the advantage of reducing the computational load as well as enabling the use of additional techniques, such as a reduction of the sampling rate of the system.

2.1.3.2 Relaxed Synchronization

Another method to reduce the amount of data used by a processing platform is to use *outdated* data. Indeed, depending on the application, synchronizations in parallel program can lead to a significant overhead.

It is shown in [Ren+12] that the K-Means algorithm on an IBM Power 7 can spend as much as 90% of computation time in synchronization. While synchronizations are mandatory in parallel programs for concurrent threads progression as well as shared memory manipulation, relaxing these constraints can lead to a reduction in computation time and energy consumption.

Data manipulations within critical sections of a program represent fundamental sources of serializations and bottlenecks in the way of computation performance and efficiency. Methods exist to reduce the performance degradation from synchronization. The Speculative Lock Elision (SLE) [RG01] technique consists in executing a critical section without acquiring the corresponding lock. Much like branch prediction mechanisms, miss-calculations from inter-thread data conflicts are checked and corrected if necessary. The SLE method potentially enables performance improvement by reducing the amount synchronization and communication. An AxC-based version of SLE, named Approximate Speculative Lock Elision (ASLE) [KAK18], consists in selectively not performing error correction despite data conflicts detection. This enables an increase in parallelization efficiency at the price of a small loss of accuracy.

2.2 Precision Optimisation

AxC techniques revolving around data representation belong into the *Data Level* category, and are the focus of this thesis.

Algorithms are usually designed with the intent of producing high-accuracy results, relying for instance on IEEE-754 Floating-Point (FP) arithmetic and written with a high-

level programming language (`C`/`C++`, Matlab, Python, ...). The constraints of implementing on an actual embedded architecture with limited resources are taken into account at a latter time. However, these complex and highly accurate reference implementations are obstacles to their embeddability. Indeed, in the case of a naïve but straight-forward implementation, the high-precision requested of the results would be achieved at the cost of high memory storage and processing power needs, as well as a high memory footprint, and even hardware resources such as a capable Floating-Point Unit (FPU). Not only these needs can be unsatisfiable on a given embedded platform, but even if they are, they may be a significant source of latency, violating a potential real-time constraint. Real-time requirements and power consumption being among the major constraints associated with the design of embedded systems, there is a need to adapt reference algorithms before embedding them. Variation in the data representation is one such method to efficiently trade-off results precision for processing power [BSM17].

2.2.1 Floating-Point Representation

Floating-point arithmetic is commonly used in application development for its ease of use. It offers both a high dynamic range and a high precision, and any necessary conversion step is directly handled by the hardware. An overview of software implementation of algorithms using floating-point arithmetic is provided in [Mul+18]. Moreover, floating-point representation is able to encode both very small and very large values. According to the IEEE-754 standard for floating-point arithmetic [19], floating-point numbers representation is composed of four parameters: the sign s, the exponent e, the mantissa m and the exponent bias b (usually equal to half the amplitude of the exponent minus 1). The corresponding value is encoded as follows:

$$x = (-1)^s \times 2^{e-b} \times (m_{M+1} + \sum_{i=1}^{M} m_{M-i} \times 2^{-i}) \tag{2.1}$$

with s the sign bit, m the *M+1*-bit wide significand e the E-bit wide exponent, stored as an integer representing the position of the radix-point, and b the exponent bias equal to $2^{E-1} - 1$. The MSB m_{M+1} of the significand is implicit, meaning it is not actually stored in memory. Instead, the value of this implicit lead-bit m_{M+1} is inferred from the exponent and is equal to 1 most of the time.

Table 2.1: IEEE-754 floating-point formats and their associated $s.e.m.b$ parameters.

	Data-type	Sign	Exponent	Mantissa	Exponent Bias	Format
IEEE-754	FP16	1	5	10	15	$\text{FlP}_{1.5.10.15}$
	FP32	1	8	23	127	$\text{FlP}_{1.8.23.127}$
	FP64	1	11	52	1023	$\text{FlP}_{1.11.52.1023}$
	FP128	1	15	112	16383	$\text{FlP}_{1.15.112.16383}$
	FP256	1	19	236	262143	$\text{FlP}_{1.19.236.262143}$

The FP format is capable of handling specific situations using the lowest and the highest binade: a) The lowest binade is used to represent subnormal numbers. When exponent $e = 0$, the implicit lead-bit m_{M+1} becomes 0 to allow the representation of number below the regular representation space. b) The highest binade is used to represent $\pm\infty$ and to represent result of incorrect operation with Not-a-Numbers (NaNs). These specific cases need to be supported by the FPU to comply with the IEEE-754 standard.

The most commonly used floating-point data-types are the IEEE-754 32-bit Single-Precision Floating-Point (FP32) format (Figure 2.6a) and the IEEE-754 64-bit Double-Precision Floating-Point (FP64) format. A FP32 number is represented with $M = 23$ bits for the mantissa and $E = 8$ bits for the exponent, and a FP64 number is represented with $M = 52$ bits for the mantissa and $E = 11$ bits for the exponent. The exponent is stored with a bias of $2^{E-1} - 1$, meaning that its value is interpreted as $2^{e-(2^{E-1}-1)}$ and is actually stored in memory as $e - (2^{E-1} - 1)$.

Using the notation $\text{FlP}_{s.e.m.b}$, with s the sign-bit, e the number of exponent bits, m the number of mantissa bits, and b the exponent bias, the FP32 format can be represented as $\text{FlP}_{1.8.23.127}$ and the FP64 format as $\text{FlP}_{1.11.52.1023}$.

The IEEE-754 standard also defines the IEEE-754 16-bit Half-Precision Floating-Point (FP16) format (Figure 2.6b) represented as $\text{FlP}_{1.5.10.15}$ and the IEEE-754 128-bit Quadruple-Precision Floating-Point (FP128) format represented as $\text{FlP}_{1.15.112.16383}$. The FP16 format is well suited for applications where higher precision is not essential in the computation, such as image processing pipelines or machine learning, while the latter is effectively only used to minimize the overflow and rounding error of FP64 operations. It also mentions an IEEE-754 256-bit Octuple-Precision Floating-Point (FP256) format represented as $\text{FlP}_{1.19.236.262143}$. These IEEE-754 formats are summed up in Table 2.1

Figure 2.5 shows the maximal relative error and the range for the three most used FP data-types for normal numbers in $\mathbb{R}^+$. Results for values from $\mathbb{R}^-$ are **symmetric**. Values

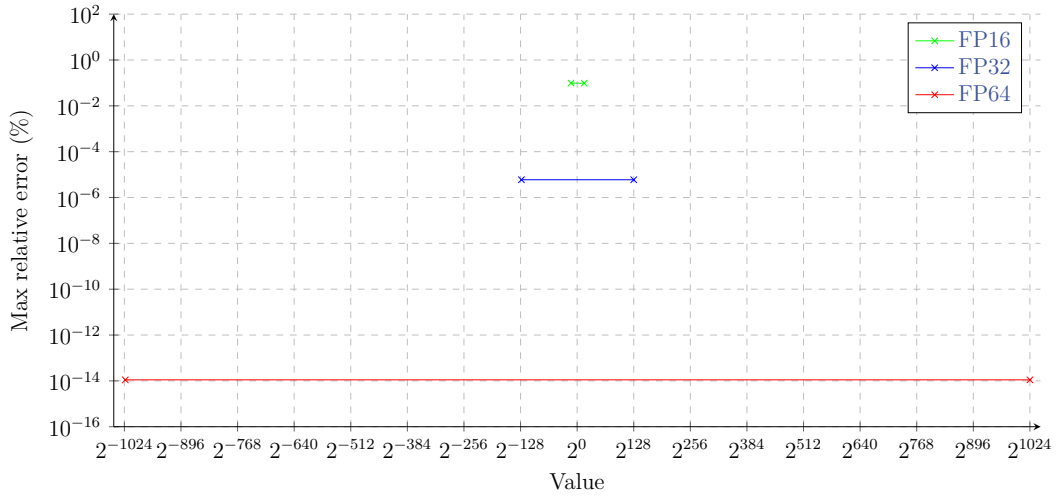


Figure 2.5: Relative error of FP data-types compared to real values.

below and above the representation space of a data-type are not shown as they are either represented as subnormal numbers or flushed to 0, or rounded to $\pm\infty$.

Increasing e enable a widening of the representation space toward 0 and $\pm\infty$, while increasing m improve the accuracy of the format. The exponent bias b is constrained to $2^{E-1} - 1$ to keep the representation space centred around 2^1.

Some FPUs may use extended-precision formats either as an independent format or as an intermediary to minimize overflow and rounding error on FP operations. The most common is the 80-bit $\text{FlP}_{1.15.63.1023}$, but a 40-bit $\text{FlP}_{1.8.31.1023}$ also exists. The compliance to the IEEE-754 standard is hardware-dependant.

While FP arithmetic is usually used both for its ease of use and its accuracy, it has drawbacks that may need to be accounted for. This may lead to accuracy issues, whose severity depends on the actual application.

FP computations are not necessarily associative, meaning that $a + (b + c)$ may not be equal to $(a + b) + c$. This inaccurate behaviour is especially susceptible to appear in heavily parallel application using atomic operation where a race condition is bound to happen, make the computation effectively unreproducible. Moreover, this leads to the necessity for leniency of an *epsilon* when checking for equality between two floating-point numbers [Knu97]. Another cause of non-reproducibility issues may be due to hardware dependent behaviour such as rounding modes, which may cause inconsistent computation between hardware architectures[xAn18]. The ability to represent special states such as `NaN` and *Infinity*s can lead to disastrous error propagation if not properly accounted for.

The use of an explicit exponent field allow FP representations to maintain a flat relative error across the representation space, which implies an increasing absolute error. This may lead to issues such as losses of significance through *catastrophic cancellation* [Gol91], when subtracting nearly equal values obtained with a small imprecision, increasing the relative error beyond acceptable bound [BOB91].

Using the already available and well-supported FP formats can be enough to substantially impact performance or resources requirements without the need to implement specific AxC techniques.

In [Tin+19], the authors show, on the NEMO and ROMS ocean models, that while a global reduction of variables precision leads to change in results beyond what was deemed acceptable, it is possible to selectively reduce variables precision without affecting the results. As scientific models tend to exclusively use FP64 variables, these reductions consist in using FP32 variables instead when possible, and even in some cases FP16 variables. This process can lead to a diminution of the memory footprint (44.9% decrease in memory usage with the NEMO model) as well as a potential performance increase, thanks to modern Central Processing Units (CPUs) vector operation capabilities.

In [Hat+17], the authors demonstrate that in some applications, such as weather forecasts, results are more impacted by observation noises and models imperfections than by computational accuracy limitations. Consequently, it is possible to reduce the size of variables from FP64 down to as low as FP16 variables, without any significant loss in results accuracy. The additional computing time recovered through the use of narrower data types can be reinvested into larger data ensembles, leading to an increase in quality of weather forecasts.

2.2.2 Fixed-Point Representation

The Fixed-Point (FxP) arithmetic enables non-integer computations to be performed on conventional integer Arithmetic Logic Units (ALUs). This combines the relative versatility of the floating-point representation with the lightweightness of integer computations. The FxP representation consists in representing an FP value as an integer.

Implementing an algorithm using floating-point arithmetic is simple and straightforward, but it has certain requirements. The implementation of an FPU has a significant cost in terms of area, energy consumption and latency. These costs are evaluated in [Bar17], for basic operations such as addition and multiplication using MentorGraphics, and shown in Table 2.2.

Table 2.2: Cost of floating-point addition and multiplication compared to integer [Bar17].

	Area (μm^2)	**Total Power** (μW)	**Critical path** (ns)
32-bit float ADD	653	0.439	2.42
64-bit float ADD	1453	1.120	4.02
32-bit integer ADD	189	0.037	1.06
64-bit integer ADD	373	0.071	2.10
32-bit float MPY	1543	0.894	2.09
64-bit float MPY	6464	6.560	4.70
32-bit integer MPY	2289	0.065	2.38
64-bit integer MPY	8841	0.184	4.52

Concerning the addition, a floating-point adder requires an area 3.5 to 3.9 times larger that an integer adder and has a power consumption 12 to 15.7 times higher and the latency is around 2 times longer. Concerning the multiplication, a floating-point multiplier requires approximately 30% less area than an integer multiplier, has a similar latency, but has a power consumption up to 35 times higher. The area used by floating-point multiplier is lower than its integer counterpart as multiplication on floating-point values is only done between mantissae, exponent being added together. This is explained by the process of adding and multiplying floating-point data within an FPU. The process of adding together two floating-point numbers requires the exponent value of the lower to be adjusted to the higher one, shifting the mantissa to the left along the way, then performing the M-bit addition, which is significantly more complex than performing an integer addition. A floating-point multiplication consists in adding together the exponent values and multiplying the mantissae, equivalent to an M-bit multiplication and an E-bit addition.

This gap in energy consumption between integer and FP computations is also verified in [Hor14] which shows that FPU-based operations can require up to 10 times more energy compared to integer operations. It also insists on the energy-consuming nature of the whole memory system, with cache-related energy consumption being 1 to 2 orders of magnitude higher than operations, and 3 orders of magnitude for DRAM accesses.

An FPU is not mandatory to perform FP computations, and resorting to software emulation is possible in cases where silicon area is critical, but comes at the cost of a significant energy and latency penalties, hence the appeal of FxP arithmetic.

A simplified way to apprehend FxP arithmetic is to consider that instead of storing the position of the radix-point with the exponent field as done in FP representation, it is implicitly kept track of.

A number x is encoded using FxP arithmetic with three parameters, the sign-bit s, the number of bits m used to encode the integer part and the number of bits n used to encode the fractional part. The parameter m usually includes the sign-bit. The sum $s+m+n$ gives the total width of the data. m represents the distance in number of bits between the radix-point and the Most Significant Bit (MSB) and n the distance between the radix-point and the Least Significant Bit (LSB). The parameter n can also be used as a scaling factor of 2^{-n}. The FxP representation x_{FxP} of the floating-point number x is obtained with:

$$x_{FxP} = < x_{FP} \times 2^n > \tag{2.2}$$

with $<\ldots>$ being the rounding mode. The data is encoded as follow:

$$x_{FxP} = (-2)^{m-1} \times s + \sum_{i=-n}^{m-2} b_i \times 2^i \tag{2.3}$$

b being the value of each bit. As with regular integers, negative values are represented with the two's complement instead of using a dedicated sign-bit.

A common notation for the FxP format associated with a variable is the Q notation $Q_{m.n}$, with n the number of bits for the fractional part and m the number of bits for the integer part. The actual use of this notation may vary from person to person on whether the sign-bit is included in the parameter m, but is always signed.

Consequently, converting an FxP value to an FP format is simply done with:

$$x_{FP} = x_{FxP} \times 2^{-n} \tag{2.4}$$

Performing computations on FxP formatted values requires a similar logic as software-emulated FPU operations: For addition and subtraction, the two operands need to have the same m parameter. The operand with the lowest m value needs to be shifted right accordingly. To account for potential overflows, the results may require an additional integer bit. In cases where the total bit-width of data is constrained, this increase of the parameter m goes along with a decrease of the parameter n.

For the multiplication of two operands $Q_{1m_1.n_1}$ and $Q_{2m_2.n_2}$, the values for Q_1 and Q_2 can be multiplied directly, but the result Q_r will be formatted as $Q_{r(m_1+m_2).(n_1+n_2)}$.

Similarly to additions, parameter m and n need to be adjusted, though multiplication between integers often imply a type promotion.

The modification of an application's computation from FP to FxP arithmetic can be a complicated and time-consuming process. As the exponent is no longer stored explicitly, every arithmetic operations with two FxP operands Q_1 and Q_2 producing a result Q_r such as $Q_{1m_1.n_1} \cdot Q_{2m_2.n_2} = Q_{rm_r.n_r}$ is unique for the set of parameter m_1, n_1, m_2, n_2, m_r and n_r.

FxP arithmetic may be used in specific use-cases where hardware resources and cost need to be minimized by omitting an FPU, such as decoding audio codec like MP3 or Ogg Vorbis. Older video game systems such as the Sony PlayStation and the Nintendo GameCube also relied on FxP arithmetic for their 3D graphics engines. The first generation of Google Tensor Processing Units (TPUs) was designed to perform 8-bit matrix multiplication through a 256x256 systolic array.

On the software side, the TeX typesetting software uses a $Q_{16.16}$ format for its position calculation and a $Q_{20.12}$ format for font metrics. The TrueType font format uses $Q_{26.6}$ for font hinting. The video game Doom and all its modern ports use a $Q_{16.16}$ format for all non-integer computations.

FxP arithmetic may also be used to guaranty bit-accurate results from an algorithm by avoiding hardware-dependent behaviour of FP calculations such as rounding rules.

It is also usable for data processing application on processors with large Single Instruction Multiple Data (SIMD) units capable of performing operations on packed integer data.

Modifying an application to support FxP can be beneficial in terms of performance, but may also enable energy efficiency improvements.

In [Has+15], the authors detail a memory efficient implementation of a Fast Fourier Transform (FFT) algorithm onto a manycore architecture, namely the Kalray MPPA-256 Andey, a 256-core processor. This implementation of the FFT algorithm makes use of FxP arithmetic to allow computation to take place on this memory-constrained architecture, having only around 1.7 MB of available memory per 16-core clusters. It results in an implementation 671 times more energy efficient than the x86 reference.

The FxP representation can also serve as a basis for alternative formats. The Logarithmic Number System (LNS) consists in encoding the logarithm of the data. The conversion process from the linear domain to the logarithmic domain is performed as $L_X = \log_b X$,

and the inverse as $b^{L_X} = X$, with L_X the logarithm of the data X and b the base. This exponent L_X can be stored and used in computation using FxP $Q_{m.n}$ representation, with two's complement. As $\log_b X$ is only defined for $X \in \mathbb{R}_+^*$, an additional sign-bit is required for case when $X < 0$. Moreover, a specific status bit is necessary to represent $X = 0$. An additional status bit can be added to enable the representation of specific cases such as $\pm\infty$ and `NaNs`. In this case, the data-width is $2 + m + n$.

The use of LNS, with the transition from the linear domain to the logarithmic domain, can be taken advantage of to modify subsequent computations [DD07]. Indeed, logarithmic identities can be applied to perform the required operation in the logarithmic domain. Some operations are simplified, such as multiplication, division, exponentiation and n^{th} root:

$$\begin{aligned} L_{X\times Y} &= L_X + L_Y \\ L_{X/Y} &= L_X - L_Y \\ L_{X^n} &= n \times L_X \\ L_{\sqrt[n]{X}} &= \frac{1}{n} L_X \end{aligned}$$

While some are made more complex:

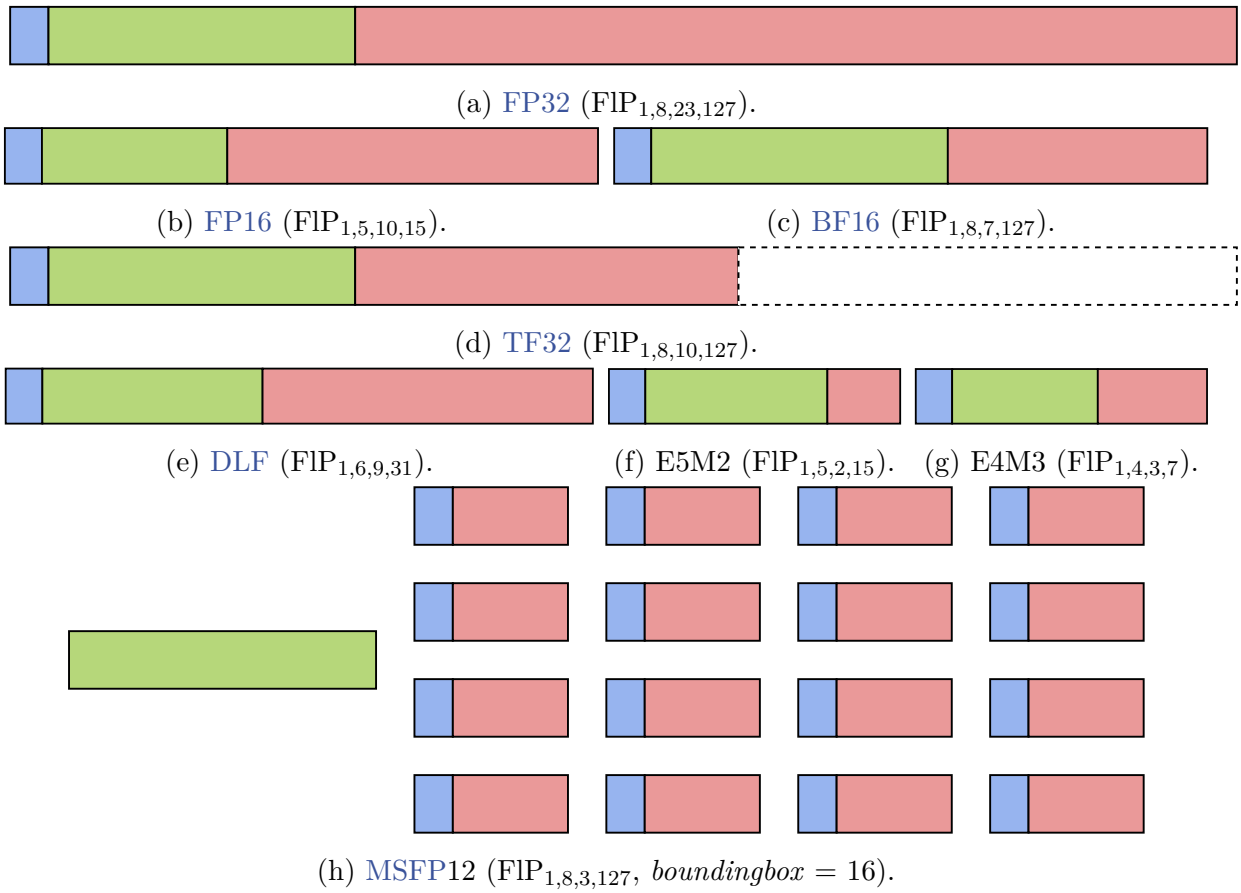
$$\begin{aligned} L_{X+Y} &= L_X + L_{(1+\frac{Y}{X})} \\ L_{X-Y} &= L_X + L_{(1-\frac{Y}{X})} \end{aligned}$$

The relevance of LNS depend on the application it is used with. It can for example be used for efficient Field-Programmable Gate Array (FPGA) implementations of neural network models, by coupling linear and logarithmic computation paths [CDP22], almost eliminating linear multiplications. By reducing the bit-width, conversion from one domain to the other can be implemented with LUTs.

2.2.3 Variation from IEEE-754 Floating-Point Standard

Custom floating-point data-types can be used on specific applications using platforms such as Application-Specific Integrated Circuits (ASICs) or FPGAs, as these can not abide by the standards, such as IEEE-754, as opposed to CPUs, Digital Signal Processors (DSPs) of Graphics Processing Units (GPUs) which are bounded by their internal architectures and

instruction sets. This allows ASICs and FPGAs to implement floating-point arithmetic using custom length of even custom behaviours.



(a) FP32 ($\mathrm{FlP}_{1,8,23,127}$).

(b) FP16 ($\mathrm{FlP}_{1,5,10,15}$). (c) BF16 ($\mathrm{FlP}_{1,8,7,127}$).

(d) TF32 ($\mathrm{FlP}_{1,8,10,127}$).

(e) DLF ($\mathrm{FlP}_{1,6,9,31}$). (f) E5M2 ($\mathrm{FlP}_{1,5,2,15}$). (g) E4M3 ($\mathrm{FlP}_{1,4,3,7}$).

(h) MSFP12 ($\mathrm{FlP}_{1,8,3,127}$, *boundingbox* = 16).

Figure 2.6: FP-based format usable as alternatives to IEEE-754 formats, separated into their sign, exponent and mantissa fields.

The regular FP32 format is widely used both for its ease of use and its wide representation space (from $\approx 2^{-126}$ up to $\approx 2^{127}$). However, the width and the relative accuracy of this representation space may be ostentatious depending on the application. This is why a flurry of alternative data representations for non-integer numbers have been developed, some with the appropriate hardware support, to fit the needs of compute-intensive application not requiring the relatively high accuracy allowed by FP32.

The Brain Floating-Point (BF16) format (Figure 2.6c), also called bfloat16, is a 16-bit wide format first proposed by Google. It is defined as $\mathrm{FlP}_{1,8,7,127}$. It features the same range as the standard FP32 format, but with a reduced precision. BF16 are intended to be used in machine learning algorithms where range is more valuable than precision.

In terms of storage and representation, it is equivalent to the FP32 format with its 16 LSBs truncated. subnormal numbers can either be flushed to zero of properly handled depending on the implementation. Support for BF16 computation can either be done through software by truncation or with actual BF16-enabled FPUs. Compared to the FP16 format, BF16 have a wider representation space and a higher relative error.

The TensorFloat-32 (TF32) format (Figure 2.6d) has been introduced by NVIDIA in 2020, with hardware support in tensor cores of its GPUs, starting from the Ampere architecture [Nvi20]. It can be represented as $\mathrm{FlP}_{1,8,10,127}$, and essentially equivalent to the truncation of FP32 down to its 19 MSBs. Despite its name, the TF32 format uses 19 bits during computation but is still stored as a FP32, on 32 bits. The TF32 format is not directly usable by the end-user as any other data-type, but rather internally used inside the GPU for intermediate computations. The TF32 format combines the 8 exponent bits from the BF16 format with the 10 mantissa bits from the FP16 format. This enables TF32 to have the range of BF16 with the precision of FP16.

Two variations of an 8-bit FP format (Figures 2.6f and 2.6g) have been introduced by NVIDIA in 2022 for AI training and inference with tensor cores of its Hopper [Nvi22b] and Ada Lovelace [Nvi22a] architecture [Mic+22]. The two formats can be defined as $\mathrm{FlP}_{1,5,2,15}$ and $\mathrm{FlP}_{1,4,3,7}$. As with the TF32 format, these two formats referred by NVIDIA as $\mathrm{FP8}_{\mathrm{E5M2}}$ and $\mathrm{FP8}_{\mathrm{E4M3}}$ are not for use by the end-user but for internal computation within a tensor core. The accumulation result from FP8 matrix multiplication is still stored with an FP32 or FP16.

The customization can be done on the length of the exponent and the mantissa, as well as different representation, like Posit arithmetic (also called Type III Unums), which claim to have a larger dynamic range and a higher accuracy [GY17]. This alternative to IEEE-754 floating-point arithmetic was shown in [Din+19] to be efficient in areas like machine learning or graphics rendering, while being worse than floating-point in areas such as particle physics simulations. This flexibility in the implementation of the floating-point arithmetic allows finer optimizations and a better compliance with the constraints previously stated, as well as the possibility to trade between accuracy and performance, as shown on FPGA in [MS13]. However, a specific study is needed when using a custom floating-point data-type to ensure the validity of the result in the targeted application.

A method for performing FP computation on FxP processors is the use of Block Floating-Points (BFPs). The BFP format consists in grouping and storing multiple FP values without their exponents, and storing a single shared exponent for the whole block.

Microsoft introduced the Microsoft Floating-Point (MSFP) format (Figure 2.6h), a type of BFP, specifically dedicated for Deep Neural Network (DNN) workloads [Dar+20]. Values sharing an exponent are grouped in a block called a *bounding-box*, usually with a size ranging between 16 and 128 elements. The *bounding-boxes* can use a tile-based partitioning for efficient data representation. The configuration of the MSFP format is variable depending on the needs, and is expressed as MSFPN, with N the bit-width (exponent included). For $N \in [8, 11]$, MSFPN is represented as $\text{FlP}_{1,5,N,15}$ and for $N \in [12, 16]$, MSFPN is represented as $\text{FlP}_{1,8,N,127}$. Because the exponent value is shared across all data within a *bounding-box*, representing 0, which requires every bit apart from the sign-bit to be at 0, would lead to the loss of the whole block. Consequently, there is no implicit bit in the MSFP representation, all mantissa bits are represented explicitly. Computations involving data within a *bounding-box* can be performed with FxP operators. Specific hardware support for MSFP enables a reduction both in terms of area and energy.

In [Agr+19], the authors propose the DLFloat (DLF) format (Figure 2.6e), a 16-bit floating-point format designed for deep learning applications, composed of a sign bit, 6 exponent bits and 9 mantissa bits, midway between the FP16 and BF16 format. The article states that deep learning accelerators using FP16 suffer from software overhead, while those which use BF16 have limited power savings. The proposed DLF format is simplified compared to the IEEE floating-point specifications to allow simplification of the FPU logic. The first binade is used to represent normal numbers instead of subnormal numbers, which are not supported. `NaN`s and *infinity* are fused into a single ***NaN**-Infinity* symbol at the top of the last binade, at value `0x7FFF` or `0xFFFF`, the rest of the last binade being used for normal numbers. *Zero* and ***NaN**-Infinity* are unsigned, meaning the sign-bit is ignored when handling these values. Finally, the rounding mode is fixed to *round-nearest-up*. Deep Learning network training using the DLF format produces results close to the FP32 reference.

A comparison between FP32 and FP16, BF16, MSFP8 and Posits for deep leaning inferences has been done in [Res+20]. The model were stored in memory with different data-types, but the actual computation were still performed on FP32, with the intent of reducing the requirement in terms of memory capacity and bandwidth. The FP16 and BF16 formats produce similar results compared to FP32. The MSFP8 format gave insufficient results, and Posits tended to perform well, with some exceptions.

Table 2.3: Common floating-point formats and derivatives and their associated *s.e.m.b* parameters.

	Data-type	Sign	Exponent	Mantissa	Exponent Bias	Format
IEEE-754	FP16	1	5	10	15	$\text{FlP}_{1.5.10.15}$
	FP32	1	8	23	127	$\text{FlP}_{1.8.23.127}$
	FP64	1	11	52	1023	$\text{FlP}_{1.11.52.1023}$
	FP128	1	15	112	16383	$\text{FlP}_{1.15.112.16383}$
	FP256	1	19	236	262143	$\text{FlP}_{1.19.236.262143}$
Non IEEE-754	BF16	1	8	7	127	$\text{FlP}_{1.8.7.127}$
	DLF	1	6	9	31	$\text{FlP}_{1.6.9.31}$
	$\text{MSFP}N_{N\in[8,11]}$	1	5	$N-6$	15	$\text{FlP}_{1.5.N-6.15}$
	$\text{MSFP}N_{N\in[12,16]}$	1	8	$N-9$	127	$\text{FlP}_{1.8.N-6.127}$
	TF32	1	8	10	127	$\text{FlP}_{1.8.10.127}$
	FP8_{E5M2}	1	5	2	15	$\text{FlP}_{1.5.2.15}$
	FP8_{E4M3}	1	4	3	7	$\text{FlP}_{1.4.3.7}$

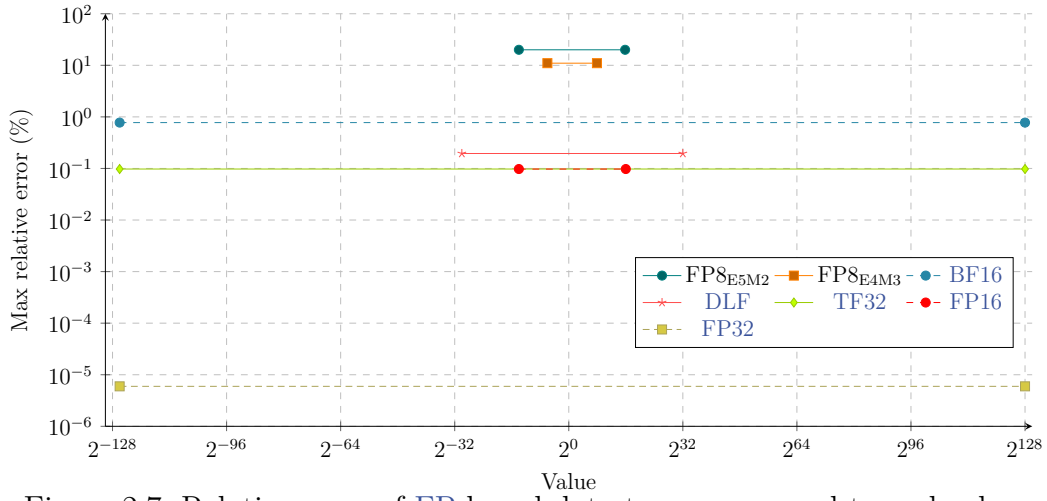


Figure 2.7: Relative error of FP-based data-types compared to real values.

Table 2.3 summarises the different FP-based data representation detailed in Section 2.2 along with the corresponding $\text{FlP}_{s.e.m.b}$ notation, while Figure 2.7 shows the expected maximum relative error and range associated with these representations.

These datatypes are primarily designed to optimize computations with reduced requirements regarding results accuracy. Conventional FPUs only requires minor modification to support these alternative formats. For example, TF32s and BF16s only consist in a truncation of FP32 data, as the FP8_{E5M2} is a truncation of the FP16 format. These datatypes are not specifically intended to reduce the memory footprint, and as such, are still dependent of the 2^N data sizes and memory alignment.

The MSFP format constitute an exception to these limitations as it is aimed at FPGA acceleration.

Conclusion

This chapter does a non-exhaustive summary of diverse existing AxC technique, segmented depending on whether they impact the data, the computation, or the hardware.

It appears that very few AxC techniques exist for the purpose of reducing the memory requirements of data processing application. Techniques that enable memory requirements reductions seem to do so as a side effect.

The memory system is already a limiting factor for large scale data processing application as well as for low-resources embedded systems. Hence the objective of this thesis to develop AxC-based methods to reduce the memory constraints.

Integration of AxC techniques into data processing application requires a deep-dive on the specific aspect to optimize, which is usually time-consuming. A possible option to reduce the AxC implementation delay can be to rely on programming models with native support for specific AxC techniques or provide a representation inherently well-suited for AxC implementations.

CHAPTER 3

Programming Models

Introduction

The hardware resources reduction methods discussed in this thesis rely on, but are not exclusive to, Synchronous Dataflow (SDF)-based Models of Computation (MoCs), which by nature expose the parallelism of applications and impose a clear separation between computations and data. It is mainly this clear separation that motivates the use of SDF-based MoCs. Consequently, this chapter also presents other parallel-oriented programming models, but the contributions of this thesis are not exclusive to parallel applications and are completely usable in purely sequential contexts.

This chapter presents the parallel programming paradigm usually used to take full advantage of parallel hardware architectures. One goal of programming models is to ease the development of parallel application making full use of the available hardware resources by hiding the complexity behind an abstraction layer.

Various programming models have been developed to target various hardware/software architectures. The most commons parallel programming Application Programming Interfaces (APIs) can be segmented into categories: Task-based programming models and Accelerator-based programming models, but other models may pose themselves as an abstraction layer on top of existing ones.

Task-based programming models refer here to APIs used to scatter a computational workload across homogeneous Processing Elements (PEs), usually on a shared memory architecture. Accelerator-based programming models refer here to APIs used to prepare and offload a computational workload on a hardware accelerator, optimized for specific workloads.

This chapter also present a hardware-agnostic paradigm as an additional abstraction layer, in the form of dataflow graphs.

Graph-based representations are commonly used to describe software architectures and data structure, or to represent the high-level behaviour of an application. The Unified Modeling Language (UML) [RJB04] is tailored for high-level specification and conception of software solutions but is not a programming model. Industrial automatons usually rely on graph-based model based on Finite-State Machines (FSMs) such as Ladder logic, or Petri Nets such as Graphcet [13] or Sequential Function Chart (SFC) [13]. These models are control-flow models [All70] and are out of scope of this thesis.
Commercial graph-based models for systems design and simulation include Matlab Simulink [Mat97] from Mathworks and LabVIEW [Joh97] from National Instrument. While these models may partially be assimilated as dataflow models [KKM16], they feature control-flow elements and are therefore out of scope of this thesis.

In this thesis, we mainly focus on dataflow MoCs, more specifically the SDF MoC and its derivatives.

First, common task-based programming models are presented in Section 3.1. Then, common accelerator-based programming models are presented in Section 3.2. Finally, dataflow-based models of computation are presented in Section 3.3.

3.1 Task-based Programming Models

We call here task-based the explicit coarse-grain division of a computation workload into a limited set of computation sub-workload. This division of workload can either be done manually by the programmer of left to the parallel programming API.

A task programming model is applied on top of symmetric multiprocessing architecture (also called shared-memory multiprocessing) consisting of a pool of independent homogeneous PEs sharing common resources through an inter-connection layer.

Task management, including mapping and scheduling, is handled by the Operating System (OS) but can be strongly hinted by the programmer through the parallel programming API, or by the API itself.

```
1  #define N 1000
2  float a[N], b[N], c[N];
3  void main(int arc, char* argv[]){
4      // Lets assume that array a[] and b[] contain data
5      for (int i = 0; i < N; i++){
6          c[i] = a[i] + b[i];
7      }
8      // Lets assume that array c[] is used
9  }
```

Listing 3.1: Simple example.

Listing 3.1 show a simple example program code adding two arrays `a` and `b` into an array `c`. This example will be used in the following section to illustrate the use of multithreading APIs.

Section 3.1 presents the basics of scattering a computational workload across a globally homogeneous pool of PEs, with a focus on the CPU-level. This section quickly goes over the two main APIs used to parallelized a workload. Section 3.2

3.1.1 Processes

A process is an instance of an application being executed. It includes the program code along with the resources in use. A process is run by an OS and hosts one or more computation threads. Processes are isolated from one another; their memory space is separated and this memory-isolation is enforced for the Memory Management Unit (MMU). For the purpose of multiprocessing application, a parent process can spawn (`fork`) child processes, either to handle orthogonal tasks or to scatter the computational workload. A child process is created with a copy of its parent process, including program code and memory space. Memory spaces are usually only symbolically separated and duplicated at `fork`-time. Actual memory separation is OS-dependent and usually occur only when modifying data, with strategies such as *copy-on-write* [Rod08]. Because of the process isolation, inter-process synchronizations and communications require specific mechanisms, such as signals, message passing (Message Passing Interface (MPI)) or explicit shared memory. The process isolation and the reliance on specific OS mechanisms create a computational overhead.

3.1.2 Threads

A thread is a sequence of instruction operated within a process. Multiple threads from a process can be executed concurrently. Threads within the same process share their memory, removing the explicit need for OS-dependent mechanisms for synchronization and communication. The OS usually provides thread-handling functions (creation, termination, mapping, ...), but scheduling and mapping of threads can be done without relying on the OS, reducing the computational overhead.

Multi-threaded application development is dependent on both the OS and the programming language. Lower-level languages such as `C` may use an OS-specific implementation while high-level languages such as `Java` exposes threading through a language-specific abstraction layer. Multiple APIs have been proposed to unify multi-threaded application development with low-level languages, such as PThreads and OpenMP.

3.1.2.1 POSIX Threads

POSIX Threads (PThreads) [18] is an API enabling multi-threading through a standardized abstraction layer. The PThreads API is widely used, directly on indirectly, for multi-threaded computations thanks to its native support in most Unix-based OSs or implementation as an abstraction layer on top of an OS specific multi-threading API, such as *pthreads4w*[1]. It provides a set of `C` constants, functions, and types to handle thread management (creation, termination, mapping, ...), mutexes, condition variables, and synchronization. Although not strictly part of the same standard, the PThreads API also interoperates with the POSIX semaphore API.

```
1     #define N 1000
2     #define NB_THREAD 4
3
4     float a[N], b[N], c[N];
5
6     void* computationTask(void *arg){
7         int index = N/NB_THREAD * *((int*) arg);
8         for (int i = index; i < index + N/NB_THREAD; i++){
9             c[i] = a[i] + b[i];
10        }
11    }
12
13    void main(int arc, char* argv[]){
14        pthread_t threads[NUM_THREADS];
```

1. pthreads4w - https://sourceforge.net/projects/pthreads4w/

```
15          int args[NUM_THREADS];
16          // Lets assume that array a[] and b[] contain data
17          for (int i=0; i < NB_THREAD; i++){
18              args[i] = i;
19              pthread_create(&thread[i], NULL, task, &args[i])
20          }
21
22          for (int i=0; i < NB_THREAD; i++){
23              pthread_join(&thread[i], NULL)
24          }
25          // Lets assume that array c[] is used
26      }
```

Listing 3.2: PThreads example.

Listing 3.2 shows the example computational workload parallelized on 4 threads with PThreads. PThreads-specific functions are called to properly create and terminate threads, and the workload needs to be packaged in a function.

The PThreads API gives the opportunity to fine-tune the multi-threaded execution and threads synchronizations, allowing the implementation of both simple and complex parallel design patterns (divide-and-conquer, master/slave, fork/join, pipelining, map/reduce, thread pool, ...). This flexibility comes at the cost of an increased complexity of application parallelization.

3.1.2.2 OpenMP

OpenMP [CJV07] is a multithreading API using compiler directives in addition to specific functions and environment variable. Support for OpenMP is mainly determined by compiler support and is already compatible with compilers from a variety of vendor/-source. While designed to work on shared memory architecture, OpenMP can be used in conjunction with inter-node communication API such as MPI to handle the distributed memory side.

The main advantage of OpenMP is the ability to simply and quickly make sequential code run with multiple thread, with minor modification to the source code. While OpenMP provides the tools for specific parallel behaviour, most use-cases can be decently optimized with as little as a single compiler directive.

```
1   #define N 1000
2   #define NB_THREAD 4
3
4   float a[N], b[N], c[N];
5
6   void main(int arc, char* argv[]){
7       // Lets assume that arrays a[] and b[] contain data
8       #pragma omp parallel for num_threads(NB_THREAD)
9       for (int i = 0; i < N; i++){
10          c[i] = a[i] + b[i];
11      }
12      // Lets assume that array c[] is used
13  }
```

Listing 3.3: OpenMP example.

Listing 3.3 shows the example computational workload parallelized on 4 threads with OpenMP. A single compiler directive is enough to scatter the computation of the `for` loop across the 4 threads. While this may not yield the best performance possible, the simplicity of use of OpenMP makes it at worse a compelling starting point to parallelize an application.

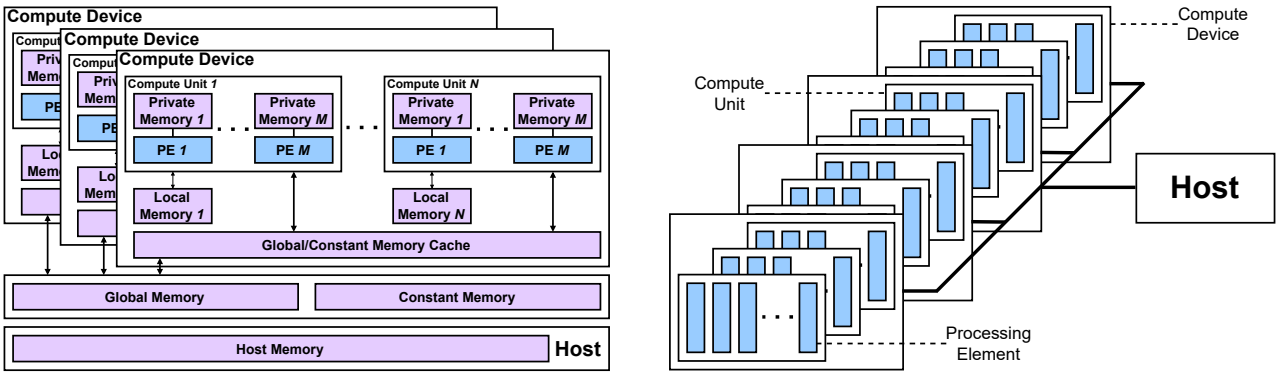
OpenMP is able to properly handle data sharing, mapping/scheduling, code segmentation, with a limited set of keywords. Fine-tuning is still required to take full advantage of the hardware architecture.

The actual implementation of OpenMP may rely on PThreads under the hood [AAB18].

3.2 Accelerator-based Programming Models

Programming models focusing on the preparation and offloading of a computation workload to a specific hardware resource are referred to as accelerator-based. Hardware accelerators can be GPUs, FPGAs, Data Processing Units (DPUs) or any other kind of ASIC. The use of accelerators usually relies on a host/device role, with the host preparing and offloading the workload to the device, and retrieving the result at the end.

The use of GPUs as hardware accelerators *i.e.* for tasks other than graphic computations, lead to the concept of General-Purpose computing on Graphics Processing Unit (GPGPU). Multiple APIs have been developed to facilitate the use of accelerators into a computation workflow, the most commons being OpenCL and CUDA.



(a) OpenCL memory model.

(b) OpenCL execution model.

Figure 3.1: OpenCL memory and execution models.

3.2.1 OpenCL/CUDA

OpenCL and CUDA are parallel programming APIs to offload the execution of compute kernels onto hardware accelerators. The CUDA API is a proprietary solution by Nvidia for use exclusively on its GPUs, while support for OpenCL is dependent on whether a hardware manufacturer provided an implementation. This section will mainly focus on OpenCL but the concept and models are still applicable to CUDA. As an open standard, OpenCL is widely supported on most CPUs (AMD, ARM, Intel), GPUs (AMD, ARM, Intel, Nvidia, Qualcomm), FPGAs (Altera, Xillinx), as well as on more exotic platforms such as the Kalray MPPA. The OpenCL framework provides a programming language based on `C/C++` (with specific keywords and functions) and an API enabling the host to control and execute code on the device.

The OpenCL framework revolves around a platform model and a memory model, as shown in Figure 3.1.

The platform model (Figure 3.1b) describes a virtual computing architecture where the host controls one or more *compute devices*, composed of *compute units*, containing PEs. The correspondence between the actual hardware and the platform model is dependent on the manufacturer implementation, but is usually done in such a way that a SIMD lane is exploited as a PE. The host interacts with compute devices through a command queue with commands for kernel execution, memory transaction, and synchronization.

OpenCL also provides an execution model, dividing the workload into *work-items*, bundleable into *work-groups*. A work-group is typically assigned to a compute unit.

The memory model (Figure 3.1a) describes the four memory regions of a compute device. Memory regions impact the locality of the data as well as its visibility from PEs. Data placement in memory regions is determined by the associated `C/C++` keyword in the device code. The *global* memory (`__global`) is a region accessible by all PEs from any devices. The *constant* memory (`__constant`) is a read-only region within the global memory. The *local* memory (`__local`) is a region accessible only by work-items from the same work-group. The *private* memory (`__private`) is a region only accessible by its work-item.

```
1  __kernel void device_computation(__global float *a,
       __global float *b, __global float *c){
2      // Arrays a, b and c have already been allocated and
           populated on the device by the host
3      size_t i = get_global_id(0);
4      c[i] = a[i] + b[i];
5      // Array c can be retrieved by the host
6  }
```

Listing 3.4: OpenCL device example.

Listing 3.4 shows the example computational workload parallelized as an OpenCL compute kernel. In this example, the `for`-loop from Listing 3.1 is completely unrolled to expose the finest granularity possible to take advantage of the massively parallel nature of GPUs. While in theory OpenCL implementations are generic and done according to reference models, compute kernels require hardware-specific optimization to achieve optimal performance, breaking the genericity.

The CUDA API is a proprietary solution but uses an execution model, a platform model and a memory model similar to that of OpenCL (Figure 3.1). The main difference between OpenCL kernels and CUDA kernels is the terminology and the corresponding host code, making CUDA-based application subject to the same kind of constraints and design decisions than GPU-oriented OpenCL applications. CUDA-based applications tend to be slightly faster than their OpenCL-based equivalent [Su+12]. The main attractions of the CUDA API over OpenCL are the CUDA-specific libraries.

3.2.2 OpenACC/OpenMP 4.0

OpenACC is a parallel programming standard which, as OpenMP, uses compiler directive along specific functions. OpenACC aims to be merged into OpenMP, which has partially

been done since OpenMP 4.0. The aim of OpenACC is to provide the ability to offload a computational workload to an accelerator with minimal source code modification. The simplicity of use of OpenACC leads to a lack of architecture-specific optimization, leading to performance gap reaching up to 100x on certain workload compare to an optimized CUDA implementation [KT21]. The merge of OpenACC into OpenMP makes it, from version 4.0 upward, a parallel programming model capable of both a task-based and an accelerator-based approach.

3.3 Dataflow-based Models of Computation

Dataflow models are commonly used to represent data processing applications in an abstract manner, potentially highlighting specific optimization opportunities. These graph-based representations may also simply be used as they are well-suited for the kind of processing to perform or for the targeted hardware architecture. Graph-based models may be used to represent the behaviour of a program, its architecture, data structure, communication, synchronization mechanisms, etc.

This section will focus on dataflow Models of Computation (MoCs) [Sav97], specifically on the Synchronous Dataflow (SDF) and Parameterized and Interfaced Synchronous Dataflow (PiSDF) MoC.

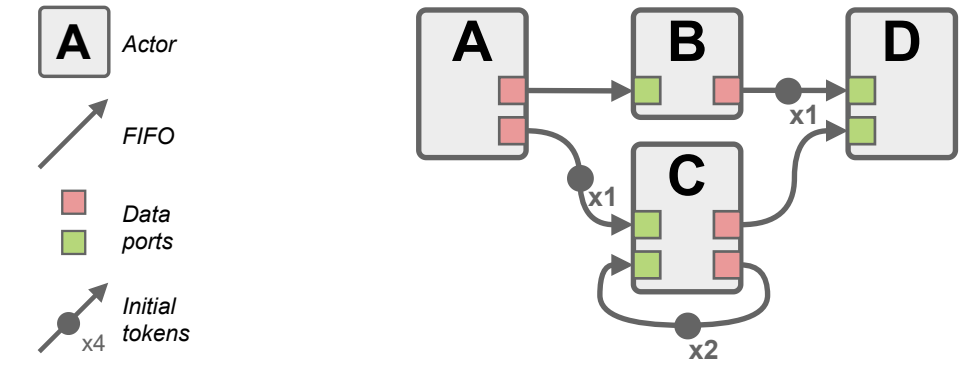
More details on the dataflow MoCs can be found in [Des14].

3.3.1 Process Network

An application defined with the Kahn Process Network (KPN) [Kah74] MoC is composed of a set of concurrent tasks interconnected by unbounded directed First-In First-Out queues (Fifos). A Fifo connecting two tasks creates a data dependency. Writing to a Fifo is non-blocking, but reading from it is blocking. A data-token is indivisible, produced exactly once and consumed exactly once. The KPN model is independent of the architecture; it is capable of exploiting parallelism but does not require it. The KPN MoC is deterministic; the same input will produce the same result.

The Dataflow Process Network (DPN) [LP95] MoC is a specialization of the KPN MoC. It attempts to provide a formal semantic for dataflow MoCs, defined as follows:

It defines a DPN as a graph composed of vertices, called actors, and edges, called Fifos. An actor is associated with sets of input and output data ports, a set of *firing*



(a) DPN Semantics. (b) DPN graph.

Figure 3.2: DPN semantic and graph example.

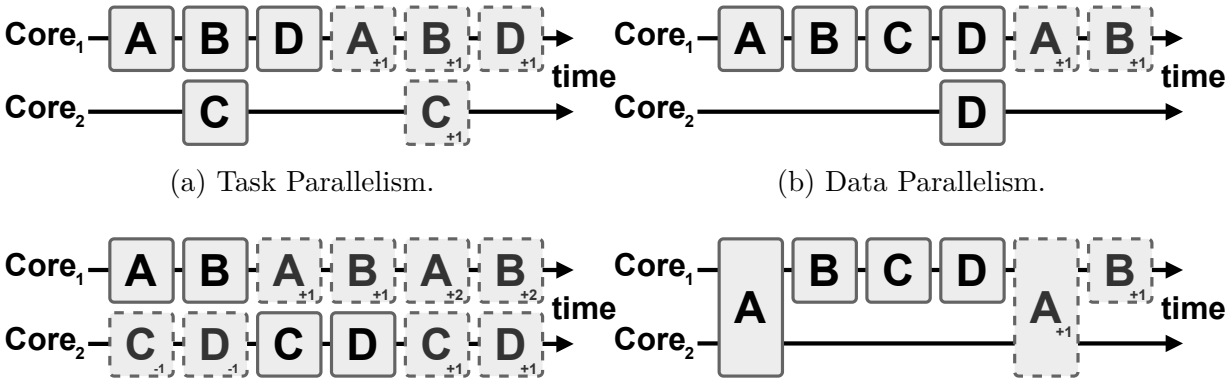
rules, and a set of data-token production and consumption rates associated with the *firing rules*. An edge is an unbounded directed FIFO transmitting data-tokens from an actor to the next. It is associated with a source of a producer actor and the sink port of a consumer actor, and to a number $n \in \mathbb{N}$ of data-tokens present at initialization.

Figure 3.2 shows the semantic of the DPN MoC (Figure 3.2a) along with an example graph (Figure 3.2b). This demonstration graph is composed of 4 actors linked together by 5 FIFOs. The FIFOs between actors `A` and `C` and between actors `B` and `D` contain 1 initial data-token each, while the FIFO cycling back on actor `C` contains 2 initial data-tokens.

3.3.2 Parallelism with Dataflow Model of Computation

Describing a data processing application with a dataflow-based MoCs allows for an exposing of opportunities of parallelism. The dataflow graph of the application can be parallelized in 4 different manners [Zho+13]: task parallelism, data parallelism, pipeline parallelism and parallel actor parallelism. Figure 3.3 shows the Gantt diagrams obtained from the different way to parallelize the graph from Figure 3.2b, mapped on 2 PEs.

- **Task Parallelism:** Actors linked by a FIFO have a data dependency, requiring the producer to be executed before the consumer (unless enough initial data-token are present in the FIFO). The dependency chain can be extended across the whole graph to obtain a data-path. Actors belong to parallel data-paths can be executed in concurrently. On the example graph from Figure 3.2b, 2 dependency chain can be extracted: `A-B-D` and `A-C-D`. Actors `B` and `C` can therefore be executed on parallel, as seen on Figure 3.3a.



(a) Task Parallelism.

(b) Data Parallelism.

(c) Pipeline Parallelism.

(d) Parallel Actor Parallelism.

Figure 3.3: Exposition of parallelism in dataflow MoCs.

- **Data Parallelism:** An actor in dataflow graphs is only reliant on the data-tokens from its input FIFOs and does not have a state or context that would make it dependent of the previous execution. If enough data-tokens are available, an actor can be fired successively and executed concurrently. On the example graph from Figure 3.2b, if actors `B` and `C` produce enough data-tokens to enable 2 firings of `D`, then the 2 instances of `D` can be executed in parallel, as seen on Figure 3.3b. If the behaviour of an actor depends on the state of its previous execution, this dependency is explicitly represented in the dataflow graph by a self-loop FIFO with a delay, as shown with actor `C` from Figure 3.2b.

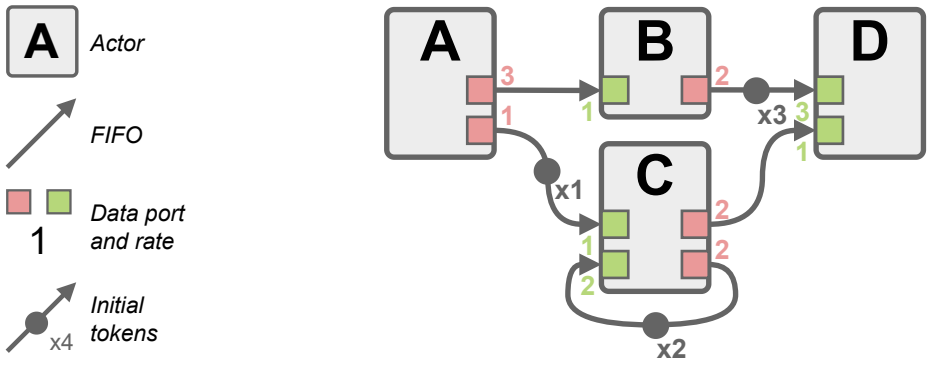
- **Pipeline Parallelism:** Application pipelining consists in starting a new iteration of a dataflow graph before the end of the previous one. Pipelining is possible when a dataflow graph do not have data dependency between its iterations. Pipelining opportunities can be exposed by separating the dataflow graph into stages with the addition of delays. On the example graph from Figure 3.2b, actors `C` and `D` have delays on all their input FIFOs, allowing their firing without prior execution of actor `A` and `B`, making 2 pipeline stages executable in parallel. Actors `A` and `B` of the current iteration can therefore be executed in parallel with `C` and `D` from the previous iteration, as seen on Figure 3.3c.

- **Parallel Actor Parallelism:** A parallel actor is an actor that internally make use of parallelism to make use of multiple PEs. This internal parallelism can either be directly provided by the host language (Sections 3.1 and 3.2) or with a hierarchical actor whose behaviour is described as a dataflow graph itself [PBR09; BB01; NL04;

Des+13]. On the example graph from Figure 3.2b, actor `A` is assumed to be a parallel actor executed on 2 PEs, as seen on Figure 3.3d.

3.3.3 Synchronous Dataflow (SDF)

The Synchronous Dataflow (SDF) [LM87] is a specialization of the DPN MoC; data-token productions and consumptions are fixed scalar values in the SDF graph. The SDF MoC is defined with the same set of rules as the DPN MoC with additional restrictions: An actor is associated with a unique firing rule, data-tokens production and consumption rate are static scalars. The firing of an actor can only occur if the required number of data-tokens are present in the input FIFOs.



(a) SDF Semantics. (b) SDF graph.

Figure 3.4: SDF semantic and graph example.

Figure 3.4 shows the semantic of the SDF MoC (Figure 3.4a) along with an example graph (Figure 3.4b). A complete execution of an SDF graph requires that every data-tokens produced during the iteration have been consumed, and the amount of initial data-token is maintained for the next iteration. With the example graph of Figure 3.4b, a complete valid iteration requires 1 firing of actor `A`, 3 firing of actor `B`, 1 firing of actor `C`, and 2 firing of actor `D`.

The SDF MoC popularity is due to its analysability, predictability and the exposition of parallelism opportunities, making it suitable for efficient execution on hardware architecture. As the SDF MoC is fully static and deterministic, it enables specific metric extraction and optimization, making it well suited for deployment on embedded systems.

The semantic of a dataflow MoC only describes interactions between actors with the *firing rules* and ports *rates* but does not specify the actual internal behaviour of said

actors. The nature of the actual computations performed by actors must still be specified by a host language. Imperative languages such as `C` or Java, or hardware description language such as VHSIC Hardware Description Language (VHDL) can be used. Specific languages have been proposed to describe both the graph and the internal behaviour of actors, such as the CAL Actor Language (CAL) [EJ03; Bha+11] or ΣC.

3.3.4 Parameterized and Interfaced Synchronous Dataflow (PiSDF)

The Parameterized and Interfaced Synchronous Dataflow (PiSDF) MoC is a modification of the SDF MoC by applying the Parameterized and Interfaced dataflow Meta-Model (PiMM) [Des+13] on top of the SDF semantic.

The PiMM reuses the hierarchical semantic from the Interfaced-Based Synchronous Dataflow (IBSDF) [PBR09] MoC with the addition of the parameter semantic from the Parameterized Synchronous Dataflow (PSDF) [BB01] MoC.

The IBSDF semantic enables the description of internal behaviour of an actor with an inner dataflow graph. This subgraph features input and output data interfaces for data-token transmission. To maintain the execution properties of the SDF MoC, data interfaces have a specific behaviour: Data input interfaces behave as *broadcast* actor, duplicating if necessary the data-token sent from the upper-level graph. Data output interfaces behave as *round buffers*, transmitting only the lasts data-token to the upper-level graph.

The PSDF semantic enables the definition of parameter which can be used in expression to define data-token rates, instead of static scalars.

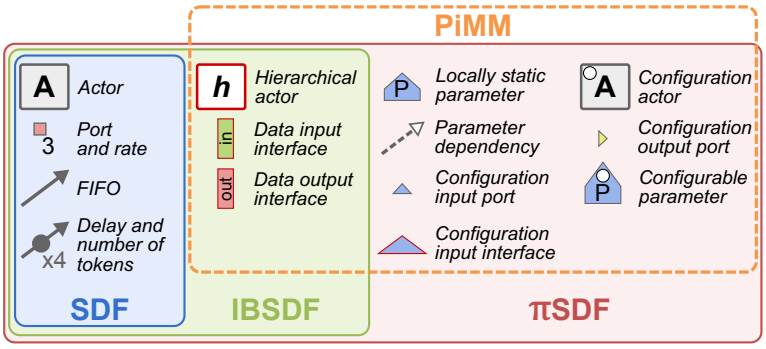
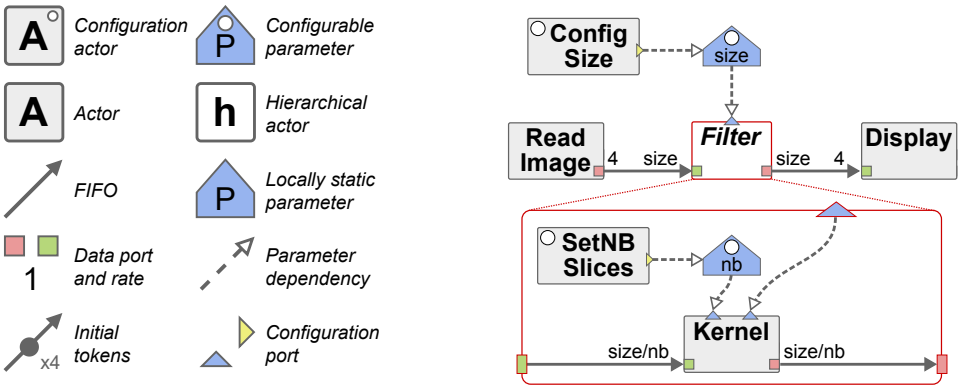


Figure 3.5: PiMM semantics.

Figure 3.5 shows a summary of the PiMM semantic.



(a) PiSDF Semantics. (b) PiSDF graph.

Figure 3.6: PiSDF semantic and graph example.

These additions make the PiSDF MoC reconfigurable at runtime. The reconfiguration is based on dynamic parameters which can change during execution. As the production and consumption rates of actor can be defined as expression depending on parameters, a dynamic parameter can change firing condition of actor, and the total number of firing during an iteration.

Figure 3.6 shows the semantic of the PiSDF MoC (Figure 3.6a) along with an example graph (Figure 3.6b). The example graph from Figure 3.6b is a dynamic graph. Actors *Config Size* and *SetNB Slices* are configuration actors capable of changing the value of a configurable parameter at runtime. A PiSDF graph without configuration actor is a static graph, and is as such eligible to any compile-time optimization applicable to conventional SDF graphs.

Dynamic graphs require running on top of a runtime manager to handle graph reconfiguration and on-the-fly optimizations.

Graph parameters can be used to determine various characteristics of an application, such as data-token production and consumption rates, delay depths, static integers for actors input, or can even be used to estimate the execution time and energy of an actor. Consequently, a dataflow graph can only represent a single configuration of the application; representing multiple configurations with the same base graph would either require the developer to change parameter values every time, or to create a separate graph for each configuration.

For this purpose, support for moldable parameters [Hon20] were added to the PiSDF MoC. Moldable parameters are able to hold multiple alternative expressions, enabling a single PiSDF graph to represent multiple configurations of the same application.

The best configuration depends on multiple criteria: throughput, latency, memory footprint, energy, or any QoS metric. Finding such a configuration for a graph with multiple moldable parameters with multiple expressions each quickly becomes time-prohibitive and represent a multi-criteria optimization problem, which can be resolved with an appropriate Design Space Exploration (DSE) algorithm [Hon+22].

Development of applications with the PiSDF MoC can be done with the Parallel and Real-time Embedded Executives Scheduling Method (PREESM) development framework.

Parallel and Real-time Embedded Executives Scheduling Method (PREESM)

PREESM[2] [Pel+14] is an application development framework design to represent data processing application using the PiSDF MoC. PREESM is capable of generating parallelized `C` code from a PiSDF graph.

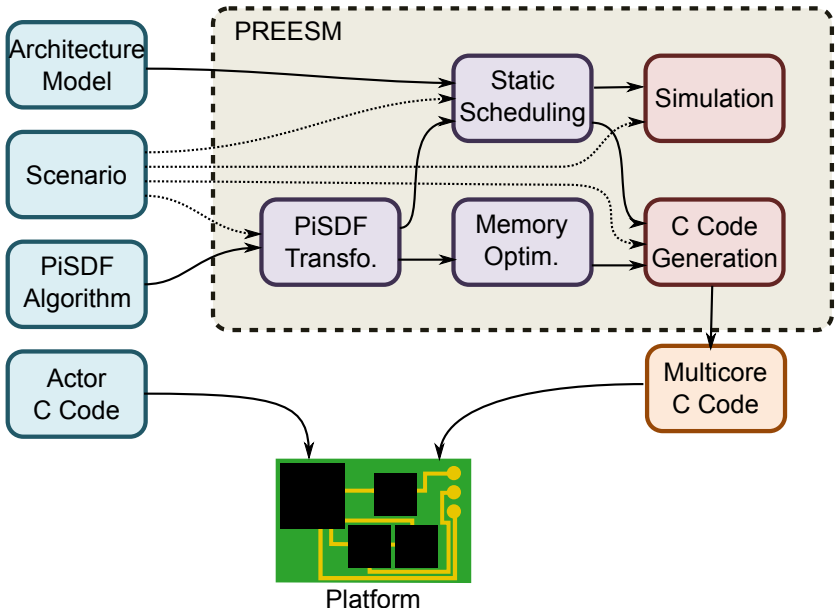


Figure 3.7: Visualization of the PREESM application development framework (Source [Heu15]).

The developer builds the PiSDF graph within PREESM and provides actors code, a model of architecture [Pel+09], and an execution scenario containing mapping constraints and timing values, as shown in Figure 3.7. From this information, PREESM handles the mapping and scheduling of actor onto the hardware architecture and memory buffer

2. https://preesm.github.io/

allocations, then generates the corresponding code. Parallel applications generated with PREESM rely on the PThreads API.

PREESM is able to target off-the-shelf multicore architecture running Windows or a Linux-based OS, as well as some specific hardware architecture, such as the C6x from Texas Instrument or the MPPA from Kalray [Has18].

Synchronous Parameterized Interfaced Dataflow Embedded Runtime (SPIDER)

The Synchronous Parameterized Interfaced Dataflow Embedded Runtime (SPIDER) [Heu+14] library is a runtime manager design to execute dynamic PiSDF graphs. The SPIDER runtime reconfigure the graph when a parameter changes, potentially leading to modification of actor firing rates. The newly configured graph is then scheduled and mapped for execution.

SPIDER relies on a *master/slave* paradigm. For each iteration, the *master* Global Runtime (GRT) reconfigures the graph, schedule actors execution, and dispatch the workload to *slaves* Local Runtimes (LRTs).

This runtime adaptability enables SPIDER to potentially remove entire data-paths from a graph. Like PREESM, SPIDER is supported off-the-shelf multicore architecture running Windows or a Linux-based OS, as well as some specific hardware architecture, such as the C6x from Texas Instrument or the MPPA from Kalray [Mio+17; Mio+18].

3.3.5 Existing tools for Dataflow Applications Design

Works in this thesis were done around the PiSDF MoC with the PREESM application development framework, but multiple other dataflow programming tools exist, supporting one or several dataflow MoCs.

Ptolemy [Buc+01; Eke+03] is a framework for modeling and simulation of applications with heterogeneous combinations of MoCs. Each level of hierarchy can be described according to a different MoC, such as DPN, SDF, or Heterochronous Dataflow (HDF) [GLL99].

The DSPCAD Lightweight Dataflow Environment (LIDE) [She+11] is a design environment for modeling, simulation and implementation of dataflow graphs on DSPs. LIDE rely on applications described with the Enable-Invoke Dataflow (EIDF) [Pli+08] MoC, a specialization of the DPN MoC.

Higher Order dataflow Coordination Language (HoCL) [Sér20] is a DPN description language designed for abstract and concise description of graph structures, including hierarchy and parameters. HoCL is independent of the target dataflow MoC and implementation, relying on dedicated back-ends. Supported back-ends include DOT for visualization, SystemC for simulation, PREESM for implementation, as well as Dataflow Interchange Format (DIF) [Hsu+04] and XDF for interfacing with other tools.

Data Activated 流 (Liu) Graph Engine (DALiuGE) [Wu+17] is a graph execution framework primarily designed for radio astronomy applications. It originated during prototyping activities of the Square Kilometre Array (SKA) Science Data Processor (SDP) Consortium, but aims to become a generic tool to target data-intensive application in general.

Conclusion

This chapter introduces parallel programming models commonly used to take full advantage of parallel hardware architectures. These programming models are separated depending on whether they are designed to scatter a workload across homogeneous PEs or to offload the computation to accelerators. Dataflow graphs are then presented as an abstraction layer usable on top of these programming models.

This thesis mainly focuses on SDF-based MoCs, which by design expose the parallelism of application and impose a clear distinction between the computation and the data it is performed on. This separation inherent to SDF-based MoCs also creates opportunities to develop AxC techniques whose impact can directly be evaluated while designing the application.

CHAPTER 4

Applications

Introduction

Dataflow models are already widely used for modelling of signal processing applications such as image processing or telecommunications [Pel+12]. Dataflow models are also well suited for deployment on embedded platforms with specific hardware and software constraints [Arr+; Sur+19; SB18; Hol+14].

In the last decade, the domain of machine learning has experienced a huge gain of popularity, leading to the development of various libraries and frameworks to ease training of deep learning models. For example, TensorFlow[1] is able to express its computation in the form of a dataflow graph to perform specific optimizations [Won+18].

Specific optimizations enabled by dataflow MoCs can be applied on distributed platform, whether on a smaller scale with clustered architectures [Has+17] or on a larger scale with High-Performance Computing (HPC) systems. Large scale multi-node architectures can range from complete processing chains from sensor acquisitions to centralized processing in HPC systems, to fine-grained distribution with edge computing [ZC20]. This creates specific opportunities for optimization of exascale data processing pipelines such as the SKA project, which has strict restrictions regarding real-time constraints and energy consumption.

1. https://www.tensorflow.org/

For example, Dask[2] is a Python library for parallel computing, splitting the computations of large sets of data and dispatching the workloads to distributed compute nodes. For this purpose, a Directed Acyclic Graph (DAG) is generated at runtime from the provided Dask collections, with vertices representing units of computations in Python functions, and edges representing data dependencies between vertices, constituting a dataflow graph. This runtime generation of the dataflow graph prevents the use of offline optimization. A similar solution is proposed by Google[3] [Aki+15].

The three applications used to demonstrate the memory reduction methods of this thesis are: 1) A basic image processing application composed of 2D-Discrete Wavelet Transform (DWT) followed by a 2D-Inverse Discrete Wavelet Transform (IDWT). 2) A *C* implementation of the SqueezeNet DNN. 3) An implementation of the SDP Imaging Pipeline, a compute-intensive process requiring a large amount of data.

The image processing application is a fairly simple wavelet filter, composed of one level 2D-DWT followed by a 2D-IDWT. Such filters can be used to de-noise 2-dimensional signals such as images. No actual de-noising is done in this experiment, as the aim is to evaluate the potential impact on output quality of memory reduction techniques as added noise.

The SqueezeNet DNN is a small Convolutional Neural Network (CNN) for image classification. Its main feat compared to other DNN is the small size of its model and its reduced overall memory footprint.

The SKA SDP Evolutionary Pipeline (SEP) is an implementation of a part of the SKA data processing pipeline that will eventually be deployed on an HPC architecture, requiring around 250 petaflops of processing power and generating 600 petabytes of data per year. The use of memory reduction techniques could have a significant impact, potentially reducing the memory-footprint requirements intra-nodes as well as communication inter-nodes.

4.1 2D Wavelet Filter

The Wavelet Filter is based on the 2D-Discrete Wavelet Transform (DWT), a transform commonly used in signal processing and compression. This first use-case is chosen as it is relative simple, has actual real-world applications, and produces an output with easily

2. https://www.dask.org/
3. https://cloud.google.com/dataflow

measurable quality. Typical applications of the 2D-DWT in image processing include denoising [CH98] and image compression format JPEG 2000 [SCE01].

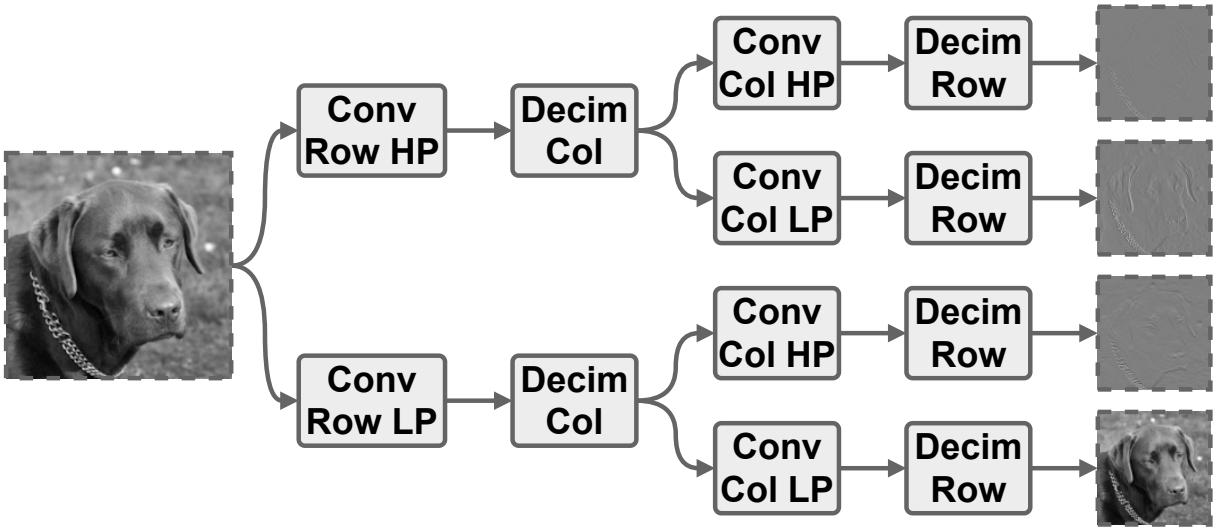


Figure 4.1: 2D-DWT diagram.

Figure 4.1 shows a diagram of the application of a 2D-DWT on an image. The process consists in applying a horizontal 1D-convolution with both a low-pass filter and a high-pass filter to obtain 2 intermediary results. Columns are downsampled by 2 before applying another pass of low-pass and high-pass filters. Lastly, these 4 intermediary results have their rows downsample by 2. The result is constituted of 4 images; 1 image consisting a downsizing of the input, and 3 images representing the horizontal, vertical and diagonal edges (from bottom to top in Figure 4.1).

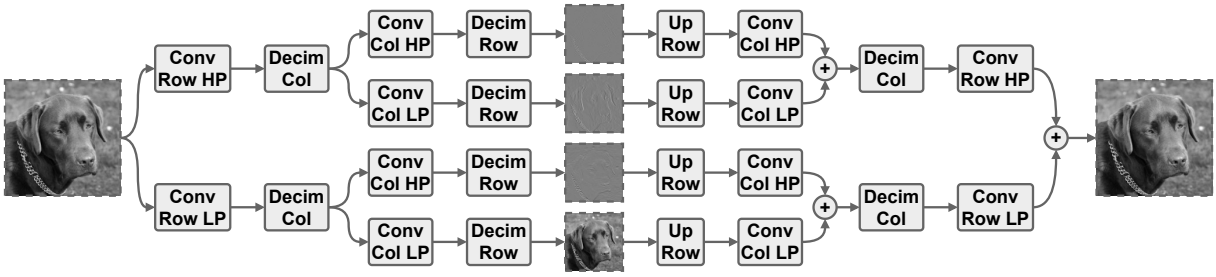


Figure 4.2: 2D-DWT diagram.

The application example chosen in this context consist in applying in succession a 2D-DWT followed by the corresponding 2D-IDWT, as shown on Figure 4.2. The *Decimation* and *Upsampling* actors shown in Figures 4.1 and 4.2 can be integrated into an adjacent *Convolution* actor but are kept separated in figures for illustration purposes.

This use-case is tested on a set of video sequences. The quality metric used is the Peak Signal-to-Noise Ratio (PSNR), comparing the expected output images of the application with the output images obtained while using AxC techniques presented in latter chapters.

The goal of this example is to show that image processing algorithms can be very resilient to precision reduction, and not just in terms of subjective perception.

4.2 SqueezeNet CNN

SqueezeNet [Ian+16] is a DNN for computer vision using a CNN architecture designed to perform image classification by associating for an input image its corresponding class.

CNNs are usually an input layer, multiple hidden layers, and an output layer. The input and output layers can be viewed as performing pre/postprocessing, feeding the image to the hidden layers, and formatting the results from hidden layers into a set of probability for each class. The configuration of hidden layers can vary both in terms of interconnection, dimensions and number of layers.

SqueezeNet achieves the same level of accuracy as the AlexNet [KSH12] CNN on the ImageNet [Den+09] dataset with 50x fewer parameters. The ImageNet dataset is composed of around 1.3 million images spanning across 1000 classes. SqueezeNet has a model of 1.25 million parameters with a size of around 4.8MB, but specific methods based on quantization and pruning enable a decrease its size, with a depth of 10 layers. For comparison, 8 out of the 10 current top performers models on the ImageNet dataset use more than 1.4 billion parameters.

The purpose of this use-case is to validate the genericity of the methods discussed further in this thesis. As such, the reference used for SqueezeNet is the default 4.8MB version of the model. The model is pre-trained and used as-is, it does not undergo any re-training. The objective is not to improve the classification rate of SqueezeNet, but rather to obtain the exact same results with a smaller overall memory footprint, as well as to provide the opportunity to selectively lower the classification accuracy for an additional reduction of the memory footprint.

The interest of this use case is that it widely differs from regular image processing application and serves as a representation of the global deep learning trend that has been growing for the last decade. It also serves to show that neural network are naturally well-suited for dataflow representation as the separation between data and computation is already applied with the layer representation.

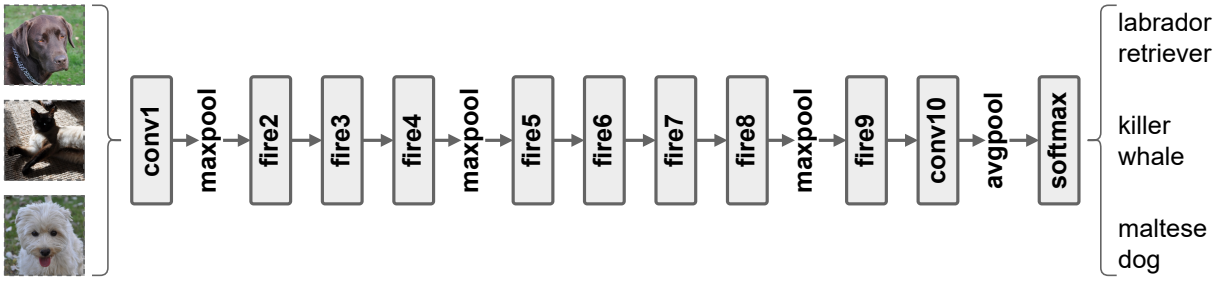


Figure 4.3: Simplified view of the Squeezenet architecture.

This use-case is tested by comparing the classification results on a set of input images with an unmodified implementation of SqueezeNet with the classification results obtained while using AxC techniques presented in latter chapters. The number of misprediction is used as the quality metric.

4.3 Square Kilometre Array Science Data Processor Implementation

The Square Kilometre Array (SKA) project is an international effort to build the world's largest radio-telescope, with a final collecting area of around one square kilometre. The SKA telescope does not use a single gigantic receptor but instead relies on interferometry, where multiple smaller receptors are coupled together in array. Receptors are located in two separate locations: 131,072 antennae in Australia for the lower frequency range, and 197 parabolic dishes in South Africa for the higher frequency range. These 130 thousands receptors will produce a combined 28 terabits per second of data, sent to remote Central Signal Processors (CSPs) producing *visibility* data and Science Data Processors (SDPs) creating 3 dimensional images. The two SDPs are expected to have a processing power of around 135 PFlops each. The annual amount of data to archive is expected to be between 300 and 700 petabytes.

SDP Evolutionary Pipeline

The SDP Imaging Pipeline[4] is an implementation of the SKA SDP by the HPC Research Laboratory at Auckland University of Technology (AUT), for the most compute intensive

4. SEP Pipeline Imaging - GitLab: https://gitlab.com/ska-telescope/sdp/sep_pipeline_imaging

task. It produces images from *visibility* data output from the CSP. This implementation of the SKA SDP has been chosen as our use case. The SDP Imaging Pipeline generates sky images with an iterative process, refining the image on every *major cycle*. Each of these *major cycles* goes through a *deconvolution* step, tasks with extracting relevant "bright" sources during each of its *minor cycles*. In this use case, the used dataset has been generated from a Galactic and Extra-galactic All-sky MWA (MWA) observation[5] in order to generate sky images that can be analysed in terms of quality. The used dataset is generated for a 15 minute observation with a visibility per baseline each 30 seconds for 512 antennas. In comparisons SKA will have hundreds of receivers producing billions of visibilities per second.

In the context of this thesis, the SDP Imaging Pipeline has been modelized for the first time as a PiSDF graph using PREESM[6]. The complexity to represent an algorithm like the SDP Imaging Pipeline lies in the fact that the algorithm is iterative. Figure 4.4 shows a simplified representation. The delay between the actors *Gains Calib* and *Gains Apply* is used to represent the *major cycles* of the SDP Imaging Pipeline and the delay on the actor *Deconvolution* is used to represent the *minor cycles*. The PiSDF graph of the SDP is made up of 150 actors and 280 Fifo buffers. This graph is then flattened by PREESM in a Single-rate Directed Acyclic Graph (SrDAG) for the analysis, the mapping-scheduling and the code generation steps. The SrDAG of the SDP is made up of 4430 actors and 11850 Fifo buffers using the parameters of the GLEAM dataset we used.

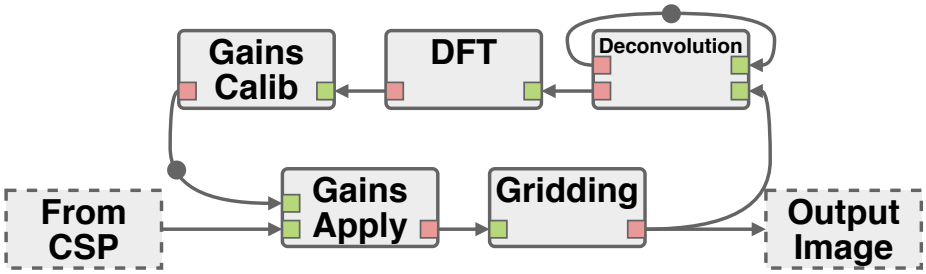
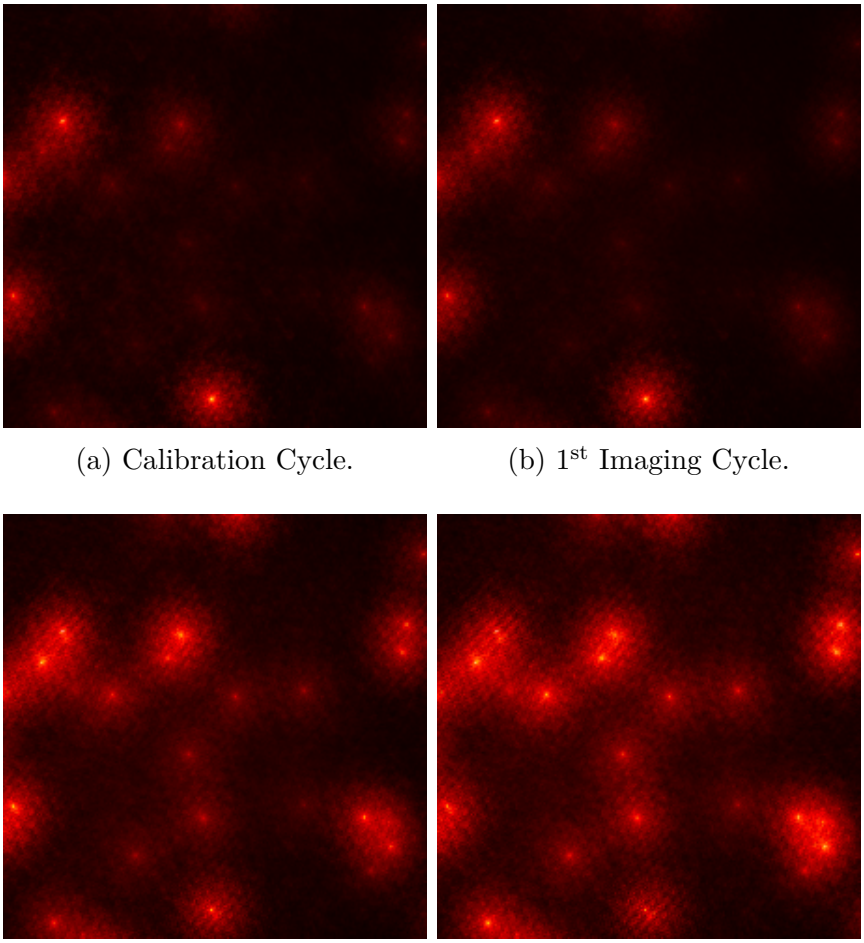


Figure 4.4: Simplified architecture of the SDP Imaging Pipeline.

The SDP Imaging Pipeline is an iterative process which in each major cycle produces a *dirty image*, containing interferometry artefacts, and cleans it with a series of minor cycles that detect sources and remove them from the image. In this use-case, the first major cycle is a calibration cycle used to correct antenna gains, followed by imaging

5. MWA - GLEAM-X Survey: https://www.mwatelescope.org/gleam-x
6. PREESM - GitHub: https://github.com/preesm/preesm/



(a) Calibration Cycle. (b) 1st Imaging Cycle.

(c) 2nd Imaging Cycle. (d) 3rd Imaging Cycle.

Figure 4.5: SDP Evolutionary Pipeline (SEP) *dirty* images output for 1 calibration and 3 imaging cycles.

cycles to detect fainter and fainter sources. A total of 4 cycles is processed, producing a *residual* dirty image each time in which fainter sources can be resolved. The reference used for output quality measurements are the four dirty images produced by taking the precise but computationally-expensive direct Fourier Transform.

The output of the SEP for 4 major cycles is shown on Figure 4.5. It is composed of 1 calibration cycle follow by 3 imaging cycles. The calibration cycle is necessary to adjust antennae gains and does not clean the output, hence the similarity between Figure 4.5a and Figure 4.5b. Successive imaging cycles on Figures 4.5b to 4.5d enable the detection of light sources that would otherwise be drowned in background noise.

Conclusion

This chapter presents the three applications used to evaluate the efficiency of the proposed techniques presented in the following chapters. It consists of a generic image processing application in the form of a 2D-DWT followed by a 2D-IDWT applied on a set of video sequences, a CNN performing image classification, and an implementation of a signal processing application for radio-astronomy.

These applications are chosen to be diverse enough to show the genericity of the presented memory reduction techniques.

Part II

Contributions

CHAPTER 5

Data Representation in Approximate Buffer

Introduction

Data processing applications are usually designed to perform computation with either IEEE-754 32-bit Single-Precision Floating-Point (FP32) or IEEE-754 64-bit Double-Precision Floating-Point (FP64) data. Data-intensive applications then require a memory system to match the computation capabilities. However, the level of accuracy provided by this extensive use of FP32 and FP64 formats is susceptible to be disproportionate compared to the initial data acquisition, or simply beyond the needs of the end-user. Consequently, alternative data formats (as presented in Section 2.2) have been developed with the goal of trading-off accuracy for a reduction of the memory footprint and, depending on hardware support, faster computations.

Memory related systems (caches, controller, interconnects, ...) already account for around 80% of chip area [Mut+23]. Moreover, data access to registers and caches can amount for more than 50% of a Central Processing Unit (CPU) energy dissipation, before taking into account external memory operations [Hor14]. Hence the idea to focus on storage paradigm designed to reduce the memory requirements of data processing applications.

This chapter introduces the concept of Approximate Buffer (AxB). It focuses on how bit-width reduction negatively impact the accuracy, and how data representation can

mitigate this loss when using AxBs. Storing data in AxBs with an alternative format, varying both data-width and representation, can potentially enable a reduction of the memory footprint required, as well as a reduction of the volume of transfer and the associated energy.

First, Section 5.1 presents the concept of AxBs. Then, Section 5.2 shows impact for truncating the Least Significant Bits (LSBs) from regular FP32 data. Next, Section 5.3 shows how the Fixed-Point (FxP) representation can be adapted with specific bit-width. Section 5.4 presents a format based on the IEEE-754 Floating-Point (FP) rule-set able to target any bit-width. Section 5.5 explains the uniform quantization concept and how it is used in a slightly different way than the norm. Finally, Section 5.6 applies the previous format to the set of test applications detailed in Chapter 4.

These notions of data-width and representation constitute the first half of the AxB concept. Parts of this contribution have been presented at the International Workshop on Signal Processing Systems (SiPS) 2020 [Mio+20].

5.1 The Concept of Approximate Buffer

The Approximate Buffer (AxB) concept belongs to the category of *Data-Level* Approximate Computing (AxC) technique that changes the representation data to reduce the memory footprint of data processing applications. It belong to the *Precision Optimization* category (Section 2.2) and in our case does not impact the nature of the computation.

The basic idea of AxBs is to store the data associated with data buffers using fewer bits than the original and no matter the data-type used during computations. As mentioned in Section 2.2, algorithms are usually designed with FP64 or FP32 arithmetic, as these data representations enable the use and computation of non-integer numbers. However, the available dynamic range and precision provided by these data-types can often be underused. Regardless of how much precision is required for a floating-point variable, it still occupies the same memory space. The same applies to integer variables, as standard data-types sizes tend to occupy either 8, 16, 32, 64 or 128 bits in memory.

Depending on the nature of the original data, a number N of bits, either LSB or Most Significant Bit (MSB), could be scrapped to reduce the size of the data-type associated to a First-In First-Out queue (Fifo) buffer, and thus of the whole buffer.

The set of data with N bits removed is then concatenated in memory to achieve the footprint reduction.

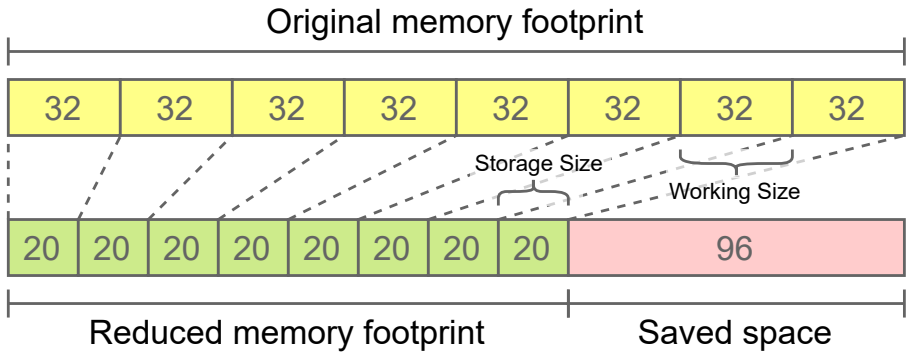


Figure 5.1: Simplified illustration of AxB, with data converted from 32 down-to 20 bits concatenated in memory.

Figure 5.1 shows a simplified representation of the AxB concept, with a set of originally 32-bit wide data converted to a 20-bit wide representation, and concatenated in memory.

AxBs are specifically targeted toward memory buffers used to transmit data from a computation bloc to the next, and is especially well suited to be applied on FIFOs from a dataflow graph. As described in Section 3.3, a dataflow graph representation clearly separates by design the computation from the data buffers, making the use of a technique such as AxBs straightforward.

An AxB stores the data using the same storage format and the same bit-width. These parameters need to enable the representation of every data within the AxB with an high-enough accuracy level.

As data stored within AxBs are altered by the bit-width reduction, the impact from using this AxC technique needs to be evaluated. This impact on overall quality, as well as the accuracy loss mitigations enabled by specific data representations are discussed in this chapter. Implementation details concerning data conversion and insertion/extraction from an AxB are explained in Chapter 6.

5.2 Data Truncation

Reducing the bit-width of a data will have varying effect depending on initial representation (integer or floating-point) and on whether the removal targets MSBs or LSB. With an (unsigned) *integer* representation on N bits, removing m MSBs reduces the range of values representable from 2^N down to 2^{N-m}, while removing m LSBs lower the accuracy by increasing the step-size from 2^0 to 2^m. Consequently, removing MSBs from an integer

data has no impact on the accuracy as long as the range stays wide enough to represent all values, after which it becomes unusable. On the other hand, removing LSBs will simply gradually degrade the data.

With FP-based representation, MSBs encode the sign-bit and the exponent field while LSBs encode the mantissa. Removing MSBs would almost immediately destroy the data while removing LSBs will progressively degrade it. As we aim at reducing data-width of FP value, we focus on LSBs removal.

Using FP32 variables, the actual value is encoded in the 23 LSBs, the rest serving as a scaling factor, and the sign bit. This means the data-width reduction opportunities are functionally limited to these 23 bits.

The average relative error compared to real values introduced by the use of data truncation is shown in Figure 5.2. Because the 9 MSBs of the FP32 representation correspond to the sign bit and the exponent, a total of 11 bits are necessary to encode a real value with an average error below 10% (1 sign bit, 8 exponent bits and only 2 mantissa bits), and 16 bits for an average error below 1%.

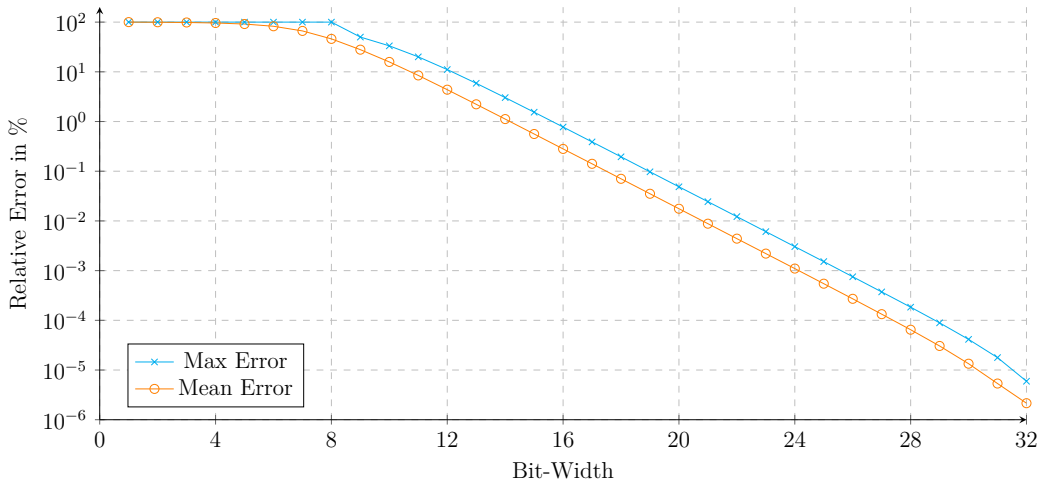
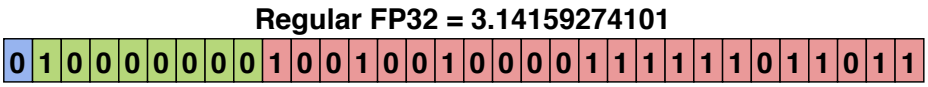


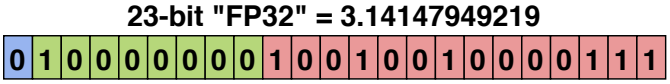
Figure 5.2: Mean and max relative error introduced by the use of truncation compared to real values.

This progressive loss of accuracy may be used to tune the memory storage to the data to store, by reducing the number of bits used to encode the value with a specific level of accuracy.

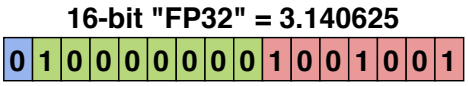
Figure 5.3 shows the loss of precision that comes with the data-width reduction with truncation. In this example, the regular FP32 allows an accurate representation of the value of the number π up to the 7^{th} decimal place. Using a 23-bit FP32 truncation, π is



(a) π encoded on a regular FP32.



(b) π encoded on a truncated 23-bit FP32.



(c) π encoded on a truncated 16-bit FP32 (equivalent to BF16).

Figure 5.3: Precision loss of π depending on data-width.

only accurate up to the 4^{th} decimal place, and only up to the 3^{rd} decimal place using a 16-bit FP32 truncation. For reference, an FP64 representation is accurate to the 15^{th} decimal place, an FP128 to the 34^{th}, and an FP256 is accurate to the 70^{th}, while an accuracy to the 38^{th} decimal place is enough to measure the circumference of the observable universe with a precision down to one atom of hydrogen.

Reducing the width of FP32 data for use with an AxB can be done by truncation, but the impact on the accuracy quickly becomes significant. The reduction of the bit-width by truncation from 32 bits to 12 bits, a 2.7 bit-width reduction, degrades the relative error of the FP32 representation from less than $10^{-5}\%$ up to 10%. This accuracy loss can be mitigated by converting the data to another representation with a bit-width more suited for values to represent.

5.3 Fixed-Point Representation

The first data conversion considered is the Fixed-Point (FxP) format. As explained in Section 2.2, FxP variables are presumed to have their exponent values known at compile time, meaning the order of magnitude of variables stored in the AxB should be known beforehand. Multiple data with the same bit-width can have a completely different $Q_{m.n}$ format. Unfortunately this may not be known in practice. To mitigate this issue, every data stored within an AxB are encoded with the same $Q_{m.n}$ format. This common format is stored as metadata for the AxB, alongside the *WorkingSize* representing the original bit-width

of the data, and the *StorageSize* representing the bit-width of data within the AxB. This allows FP variables to be stored in even fewer bits than the truncation method described in Section 5.2.

The determination of an appropriate common $Q_{m.n}$ format for the AxB can either be done offline during the setup of the AxB technique, or live at runtime. If the number of bits allocated to the integer part is too low, there is a risk of overflow. If it is set too high, the accuracy will be reduced.

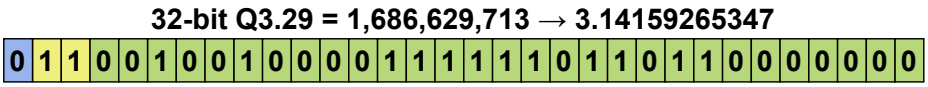
The offline determination can be done with interval arithmetic for relatively simple application such as image processing algorithms. This method may become complicated with more complex and compute intensive applications. In such cases, a simpler solution is to track the ranges of values stored into a buffer during dry runs of the application for a wide range of inputs.

The online method consists in checking during insertion if the current $Q_{m.n}$ format of the AxB allows for storing the data without overflow. If not, every data within the AxB can be reformatted with a new $Q_{m.n}$ format. While this means that the previously stored data will lose some precision, it enables storing values that would otherwise end up overflowing.

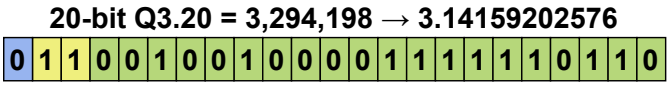
Additionally, if the content of the AxB is known to be always positive or always negative, the sign-bit (which is included in m) can be removed and instead kept as an AxB parameter. Consequently, the FxP format notation can be extended as $Q_{s.m.n}$, with n the number of bits for the fractional part, m the number of bit for the integer part without the sign-bit, and s to represent the sign, with a value of '+' or '-' if the sign is fixed, or '1' if an actual sign-bit is required. The $Q_{m.n}$ notation is still usable and is assumed to have a sign-bit included in the parameter m.

The main drawback of this storage method with an FxP representation is that AxBs containing values across a wide range might have their smallest values stored with a high relative error or rounded to zero, which might be problematic with some applications.

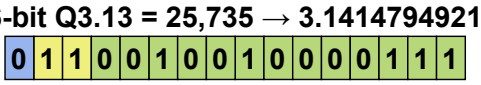
Figure 5.4 shows the loss of precision that comes with the data-width reduction with the FxP representation. As in Figure 5.3, π is used a the reference value. The raw integer value is shown, along with the associated FxP interpretation with the corresponding $Q_{m.n}$ format. The $Q_{m.n}$ FxP format is set to better encode the value of the number π, with $m = 3$ and $n = nbBit - m$. In this example, the $Q_{3.29}$ FxP format allows an accurate representation of π up to the 9^{th} decimal place. Using a $Q_{3.20}$ FxP representation, π is still accurate up to the 7^{th} decimal place, the same as the FP32 format, and to the 4^{th}



(a) π encoded with $Q_{3.29}$ format.



(b) π encoded with $Q_{3.20}$ format.



(c) π encoded with $Q_{3.13}$ format.

Figure 5.4: Precision loss of π depending on data-width with FxP representation.

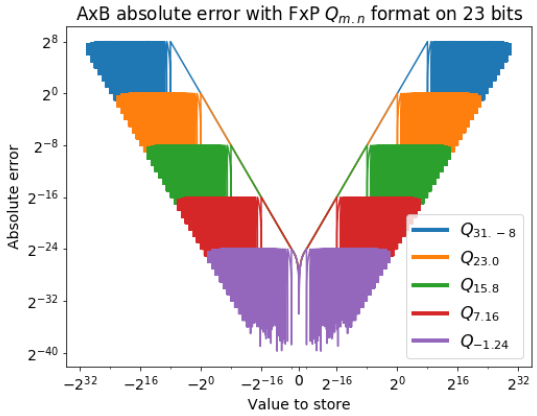
decimal place using a $Q_{3.13}$ FxP representation. In this example, the 16 bits $Q_{3.13}$ FxP format offer the same level of precision as the 23-bit truncated FP32. These $Q_{3.n}$ formats can represent values from $[-2^{3-1}, 2^{3-1} - 2^{-n}]$.

Precision

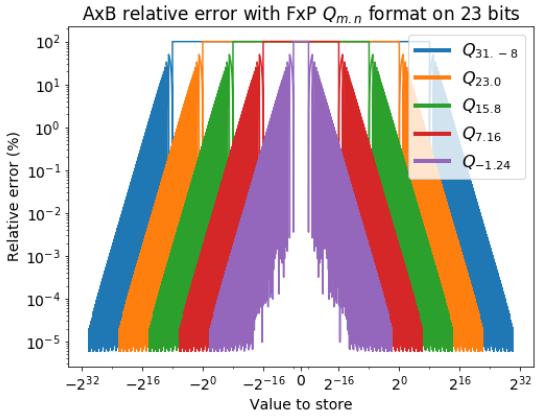
As previously indicated, the main drawback of the $Q_{m.n}$ FxP format is its limited range, roughly spanning from -2^m to 2^m. The absolute $|x_{real} - x_{FxP}|$ and relative $\left|\frac{x_{real} - x_{FxP}}{x_{real}}\right|$ error introduced by the use of AxBs with the FxP representation on 23 bits is shown on Figure 5.5 for various $Q_{m.n}$ formats.

As the step-size is equal to 2^{-n}, values close to the lower bound have a higher relative error, which in return guaranties a maximum absolute error of 2^{-n}. Out of the whole $[-2^{m-1}, 2^{m-1} - 2^{-n}]$ representation space of the $Q_{m.n}$ format, values in the interval $]-2^{-n}, 2^{-n}[$ shows the worst relative error at 100%, as values in this range are flushed to zero.

This enables the $Q_{m.n}$ FxP format to have a higher precision than the same-sized FP format near the upper representable limit and gives a clear 2^{-n} maximum absolute error.



(a) Absolute error from using AxBs with a 23-bit wide FxP format.



(b) Relative error from using AxBs with a 23-bit wide FxP format.

Figure 5.5: Error from using AxBs with a 23-bit wide FxP format with different $Q_{m.n}$ parameters.

5.4 Custom Floating-Point Representation

The regular FP32 format is able to, thanks to its exponent field, represent values ranging from $\approx 2^{-126}$ up to $\approx 2^{127}$ (excluding subnormal numbers), while keeping a precision between 6 and 9 significant decimal digits. This uniform precision across this wide range is possible as the format is equivalent to a juxtaposition of 254 representation spaces, ranging from 2^e to $2^{(e+1)-q}$, e being the exponent value and q being the step-size, equal to 2^{e-23} for FP32. The maximum absolute error is below 2^{e-23}, and the maximum relative error below $6.10^{-6}\%$.

The basic implementation of AxBs, using no alternative data representation, truncates LSBs (as described at the beginning of Section 5.2) from floating-point values, which is equivalent to increasing the step-size. Alternative representations derived from the reference FP32 format will be designated as Custom Floatind-Point (cFP) by the notation $cFP_{s.e.m.b}$, with s the sign-bit, e the number of exponent bits, m the number of mantissa bits, and b the exponent bias. Unless specified otherwise, the representation space is centred around the value 1, with the exponent bias $b = 2^{e-1} - 1$. The regular FP32 format is equivalent to $cFP_{1.8.23.127}$.

The cFP implementation featured in AxBs uses the $cFP_{s.e.m.b}$ format to adapt the internal storage representation of the AxB to the data as well as possible.

As with the use of FxP representation in AxBs described in Section 5.3, using an appropriate storage format is necessary to ensure a proper representation of values. The same strategies can be use for offline and online determination of the s, e, m and b parameters.

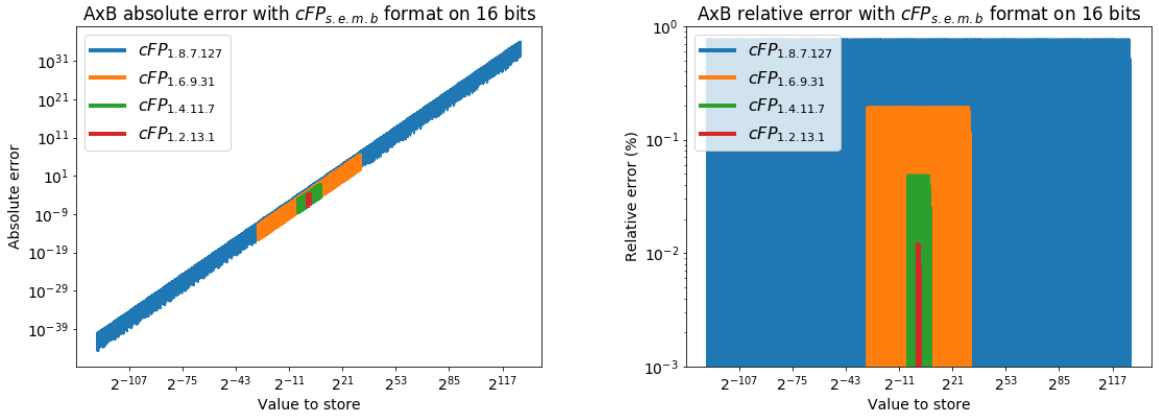
However, while the $Q_{m.n}$ format has a single parameter controlling the representation space (the number of integer bits m), the $cFP_{s.e.m.b}$ format has two. The parameter e corresponding to the number of bits for the exponent field has a similar effect on the representation space as the parameter m from the $Q_{m.n}$ format, but the parameter b corresponding to the exponent bias controls its centre point.

By default, the representation space is centred around 1. If, as an example, the values within the AxB were to be in the interval $[-1, 1]$, then half of the representation space corresponding to positive exponent values would be unused. In such case, the exponent bias b can be decreased to exclude positive exponent values, enabling a reduction of the exponent field by an additional bit.

To go further, adjusting the exponent bias can also be done to purposely flush to zero the lowest values. The number of binade available is equal to 2^e. If, as an example, the values to store within the AxB spanned across 20 binades, the safest value for the number of bits e assigned to the exponent field would be 5, provided $2^5 = 32 \geq 20$ binades. It would guaranty for every data to be stored without issues, but at the same time leaving 4 binades unused. The other possibility is to purposely discard the 4 lower binades (consequently flushed to zero) by allocating $e = 4$ bits to the exponent field and adjusting the exponent bias b to match the 16 higher binades. The spare bit can then be assigned to the mantissa m to improve the storage accuracy of every other values of the AxB. The impact of this modification is shown in Section 5.6.1.

The absolute and relative error introduced by the use of AxBs with the $cFP_{s.e.m.b}$ cFP format on 16 bits is shown on Figure 5.6. Note that while the negative part is not shown on Figure 5.6 because of the logarithmic scale, it exists and is symmetric to the positive part as the s parameter is equal to 1.

The use of the $cFP_{s.e.m.b}$ format allows a fine-tuning of the representation space of the AxB. The e parameter controls the width of the representation space, the m parameter the precision, and the b parameter the centre of the representation space, which by default is equal to $2^{e-1} - 1$, centred around 1 (on a logarithmic scale).



(a) Absolute error from using AxBs with different 16-bit wide cFP formats.

(b) Relative error from using AxBs with different 16-bit wide cFP formats.

Figure 5.6: Error from using AxBs with a 16-bit wide cFP format with different $cFP_{s.e.m.b}$ parameters compared to FP32.

The cFP implementation in AxBs does not support some features from the regular IEEE-754 reference, such as *infinity*, *NaN*s and subnormal numbers. Consequently, the first and last binade are fully used to represent normal numbers. Data above the representation space of an AxB will be stored with an incorrect exponent value, hence the need to properly dimension the *s.e.m.b.* parameter set, while data below the representation space will be flushed to zero.

5.5 Uniform Quantization

The use of alternative representations with AxBs previously presented in Section 5.3 and Section 5.4 mainly consist in reducing the representation space as much as permitted (the parameter m for FxP, and e for cFP), and using the remaining bits to increase precision (n for FxP, and m for cFP). It has the advantage of only requiring to know the approximate range of data within the AxB, but the new representation space may still be partly unused. This is because both FxP and cFP representations have their range determined with a power of 2.

The principle of the uniform quantization is to redefine the usual 2^N-based *step size* separating consecutive values. For the $Q_{m.n}$ format, the $stepSize = 2^{-n}$. Note that regular integer data-types can be expressed as $Q_{m.0}$, with m equal to 8, 16, 32 or 64. For the

$cFP_{s.e.m.b}$ format, the $stepSize = 2^{E-b} \times 2^{-m}$ with E the exponent value associated with a binade. This is also valid for the normal numbers of FP-based data-types from Section 2.2.

With uniform quantization, the *step size* is dependent on the range of values to encode and on the number of step permitted by the bit-width $nBit$, such as $stepSize = \frac{max - min}{2^{nBit}}$. The quantized data corresponds to the number of step to add to get back to the original value, offsetted by $offset = -min$, essentially consisting in mapping the $[0, max - min]$ interval to $[0, 2^{nBit}]$.

The regular quantization encoding is done with:

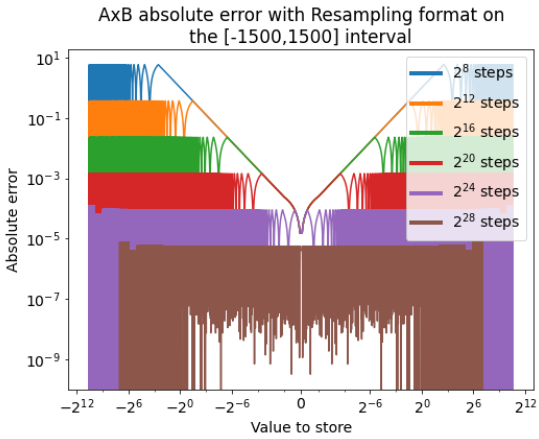
$$x_{UQ} =< \frac{x + offset}{stepSize} > \tag{5.1}$$

with $<\ldots>$ a rounding-to-nearest operation, and the quantization decoding with:

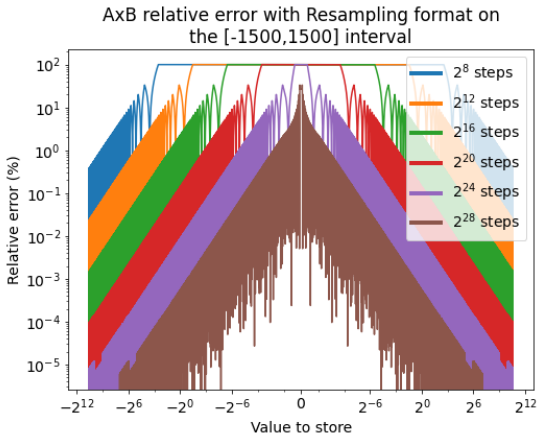
$$x = x_{UQ} \times stepSize - offset \tag{5.2}$$

This method exhibit a similar kind of imprecision as the FxP representation with the $Q_{m.n}$ format detailed in Section 5.3. The maximal absolute error is constant, and the maximal relative error increases for smaller values. However, while the FxP format have its maximal relative error at the smaller side of its representation space, the quantized representation have its maximal relative error near zero, which is not necessarily part of the chosen representation interval. This is a consequence to the way the step-size is defined. With the $Q_{m.n}$ format, the step-size is equal to 2^{-n}, which also corresponds to the lower bound of the representation space, regardless of the available number of bits. With uniform quantization, the step-size is adjusted to the representation space as well as the available number of bits.

Figure 5.7 shows that the quantized format behaves similarly as the FxP representation, with the maximum absolute value corresponding to the *step size* and the maximal relative error reaching 100% in the neighbourhood of 0. However, this high relative error do not happen if the quantized representation space does not contain 0, and the farther it is from it. Additionally, the conventional uniform quantization encoding is unlikely to properly represent the value 0 which is instead decoded as a small but non-zero value. This may have a significant impact on signal processing application relying on the actual value 0. Furthermore, the round-to-nearest operation may cause positive values close to zero to be mapped to the closest step during encoding, which may then be decoded as a small negative values, propagating an erroneous sign.



(a) Absolute error from using uniform quantization on $[-1500, 1500]$ interval.



(b) Relative error from using uniform quantization on $[-1500, 1500]$ interval.

Figure 5.7: Error from using uniform quantization for different data-width in the neighbourhood of zero compared to FP32.

The solution chosen for the implementation of quantization in AxBs is to slightly modify the conventional encoding and decoding method.

The modified quantization encoding is done with:

$$x_{UQ} =< \frac{x}{stepSize} + offset > \tag{5.3}$$

with $<\ldots>$ a truncation operation, and the modified quantization decoding with:

$$x = (x_{UQ} - offset) \times stepSize \tag{5.4}$$

The *step size* is still defined with $stepSize = \frac{max - min}{2^{nBit}}$, but the *offset* now corresponds to the number of step to reach the lower bound of the representation space such as $offset = \frac{-min}{stepSize}$. This modification guaranties the proper encoding of 0 with a dedicated step. Additionally, the replacement of the round-to-nearest operation by a truncation leads to negative values close to 0 to be encoded and decoded as minus 1 step instead, removing the previously mentioned encoding imprecision causing small values to be decoded with the opposite sign.

The impact of this modification is shown in Section 5.6.1.

5.6 Experimental Results

The impact of the data representation explained in this chapter is measured on the application set presented in Chapter 4. As detailed in Chapter 5, the aim of this chapter is to demonstrate how the use of alternative data representation can mitigate the accuracy loss inferred by data-width reductions.

5.6.1 2D Wavelet Filter

The first use-case is an image processing pipeline consisting of a 2D-Discrete Wavelet Transform (DWT) followed by an 2D-Inverse Discrete Wavelet Transform (IDWT), as shown in a simplified way in Figure 5.8. The goal of this example is to show that image processing algorithms can be very resilient to precision reduction, and not just in terms of subjective perception.

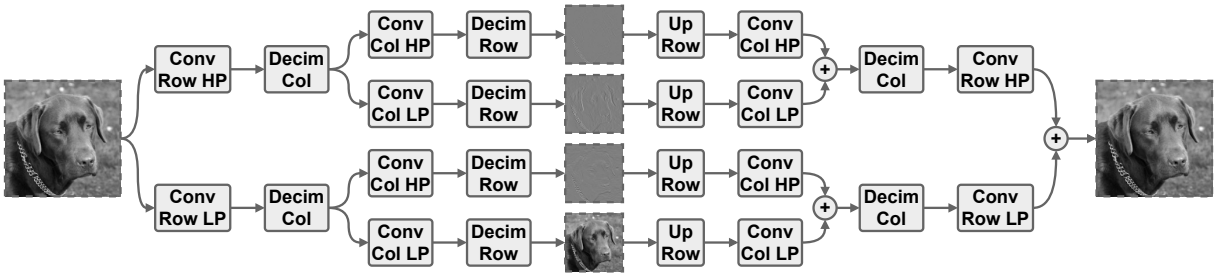


Figure 5.8: Simplified representation of the application. Every FIFO from the first *Conv Row HP* and *Conv Row LP* are AxBs.

This experiment applies the 2D-DWT to a video stream, followed by a 2D-IDWT, on the luminance component Y of each image of a set of video sequences. Every buffer of this application is an AxB (aside from buffers holding chrominance components `U` and `V`), using the same bit-width and the same representation, but with a format adapted to their content, for a total of 15 AxBs.

The quality metric used in this example is the Peak Signal-to-Noise Ratio (PSNR), comparing the result frame obtained with and without AxBs. The final quality used to qualify bit-width/representation couples corresponds to the worst PSNR value obtained on any frame of any video sequence.

The graph on Figure 5.9 shows that the use of alternative representation has clear benefits over a simple truncation of the base FP32 format.

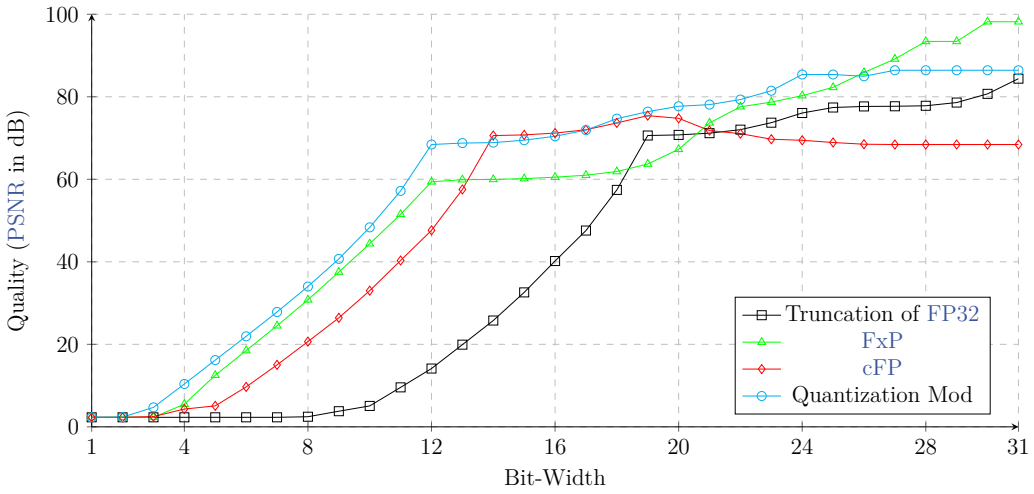


Figure 5.9: Graph of the worst PSNR dependant of the bit-width for various representations.

The modified quantization representation tops most of the chart. This lead comes with the constraint of having to specifically tune the proper range for each AxB, instead of simply relying on 2^N-based orders of magnitude.

These results validate the concept of using alternative data representations to mitigate the accuracy loss from reducing the bet-width of data, with every tested alternative representations maintaining a quality above 60dB with a simple generalized use of AxBs with a width of 14 bits.

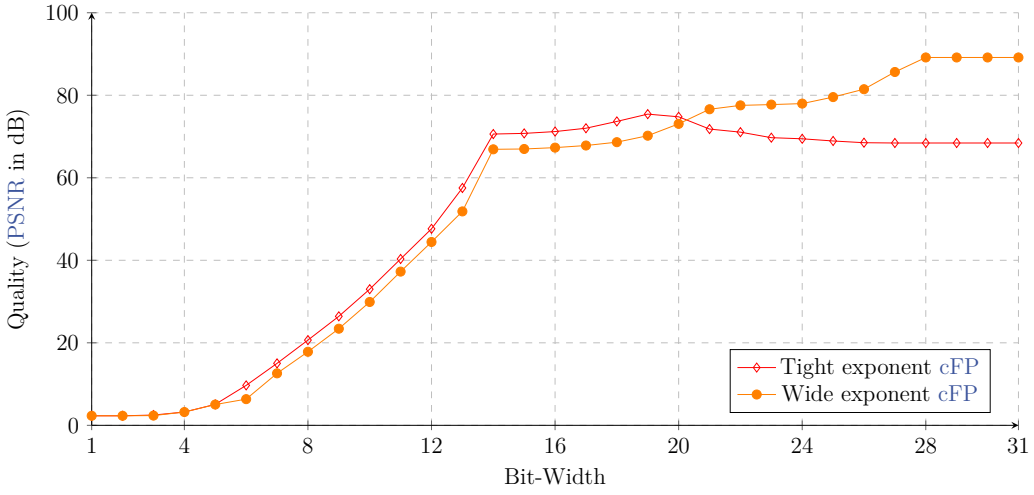


Figure 5.10: Comparison between tight and wide exponent range.

The graph on Figure 5.10 illustrates the impact of removing the lower binades, as mentioned in Section 5.4. Adjusting the exponent bias e and the width of the exponent field e, lower binades are flushed to zero, saving up an additional bit to allocate to the mantissa field m. This enables the format with the tighter exponent field to be more accurate on narrower bit-width. In return, this format with a tighter exponent field is less accurate than its "wide exponent" counterpart which achieves a better accuracy from 20 bits upward in this example.

On a side note, it can be observed on Figure 5.9 and Figure 5.10 that the best accuracy is not necessarily achieves with the widest bit-width, as the tight cFP representation reaches it best quality value with 19 bits.

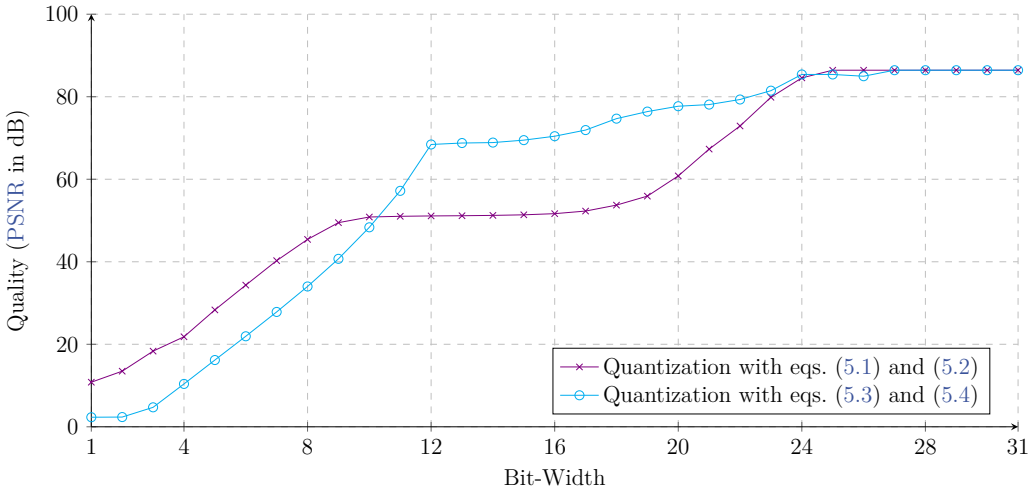


Figure 5.11: Graph of the worst PSNR dependant of the bit-width with the regular uniform quantization and to modified quantization.

The graph on Figure 5.11 illustrates the impact modifying the conventional uniform quantization encoding and decoding methods, as mentioned in Section 5.5. As expected, the modification provide less accurate results for bigger *step sizes* (due to the narrower width), which causes negative values close to zero to be encoded to the lower step instead of the closest one, although this issue becomes less and less impactful as the number of steps increases. In return, the accuracy is greatly increased from 11 bits and up, with an improvement reaching 20 dB. This is thanks to small data no longer being susceptible to changing their sign when decoded, as well as the value 0 being preserved.

5.6.2 SqueezeNet Deep Neural Network

The second use-case is the SqueezeNet [Ian+16] Convolutional Neural Network (CNN). It takes images as input and gives corresponding class. This application is presented in Section 4.2. The goal of this example is to show that CNN can be very resilient to precision reduction, validating the relevance of the AxB concept on this class of application.

The quality metric used in this example is the error rate of TOP-1 classification compared to a regular implementation of the SqueezeNet CNN with a dataset of 300 images.

In this example, the same logic of using the same data representation and bit-width from Section 5.6.1 is used, but buffers containing the model weights and buffers used between layers are treated independently.

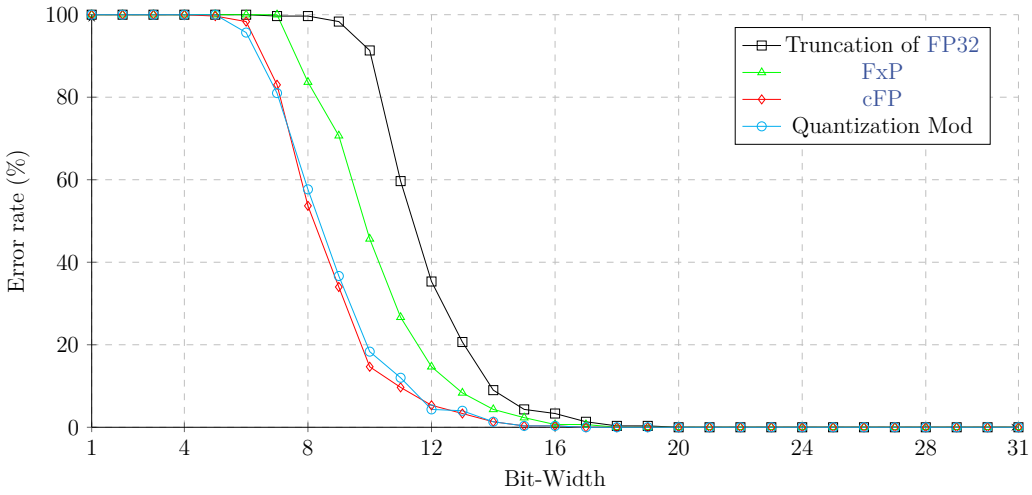


Figure 5.12: Squeezenet prediction error rate using AxBs to store data between layers, compared to regular SqueezeNet.

The graph on Figure 5.12 shows that reducing the bit-width of interlayer buffers by truncation down to 20 bits does not introduce any miss-prediction. Using alternative data representation enable the bit-width to be reduced even further without error, down to 18 bits with the FxP representation, and down to 17 bits with cFP and uniform quantization.

The graph on Figure 5.13 shows that reducing the bit-width of model weights by truncation down to 20 bits does not introduce any miss-prediction. Using alternative data representation enable the bit-width to be reduced even further without error, down to 19 bits with the uniform quantization, down to 17 bits with cFP and to 16 bits with FxP.

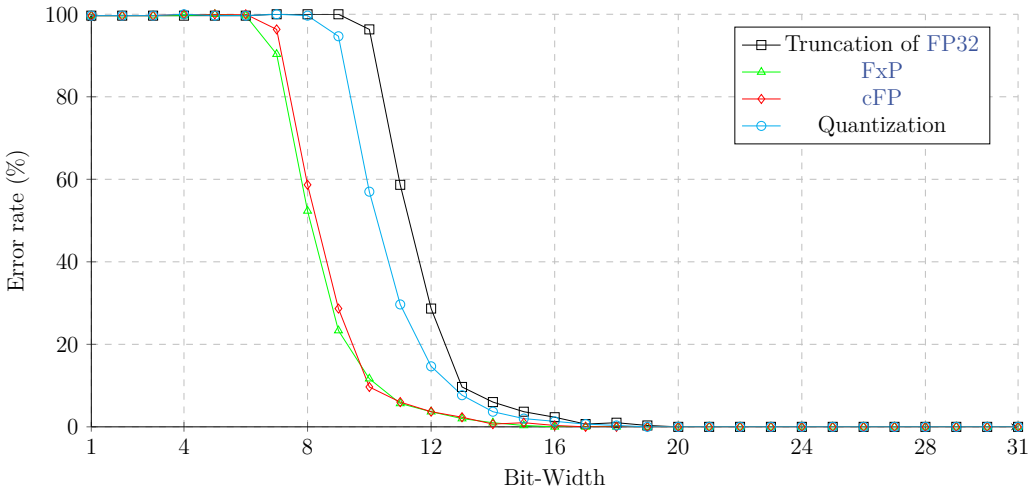


Figure 5.13: Squeezenet prediction error rate using AxBs to store weights, compared to regular SqueezeNet.

5.6.3 SDP Evolutionary Pipeline

The last use-case is the Science Data Processor (SDP) Imaging Pipeline[1].

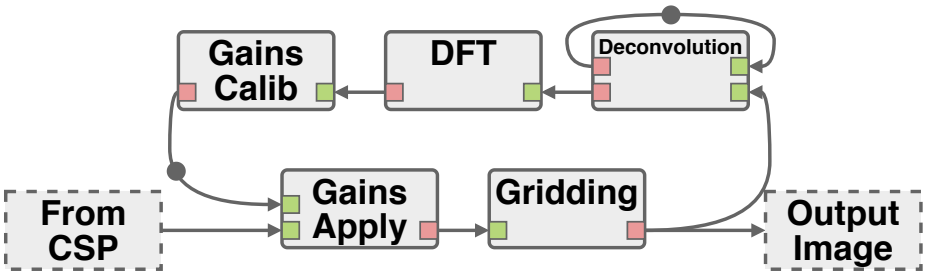


Figure 5.14: Simplified architecture of the SDP Imaging Pipeline.

Figure 5.14 shows a simplified representation of the application.

The goal is to show that the concept of AxBs can also be applied on specialized signal processing applications. Results from previous use-case show that the truncation of FP32 is irrelevant.

As this application is complex, a reduced set buffer is considered to study the impact of alternative representation as a mean to mitigate accuracy loss from bit-width reduction. The selected buffers for the study are the output buffers of actors shown in Figure 4.4, as

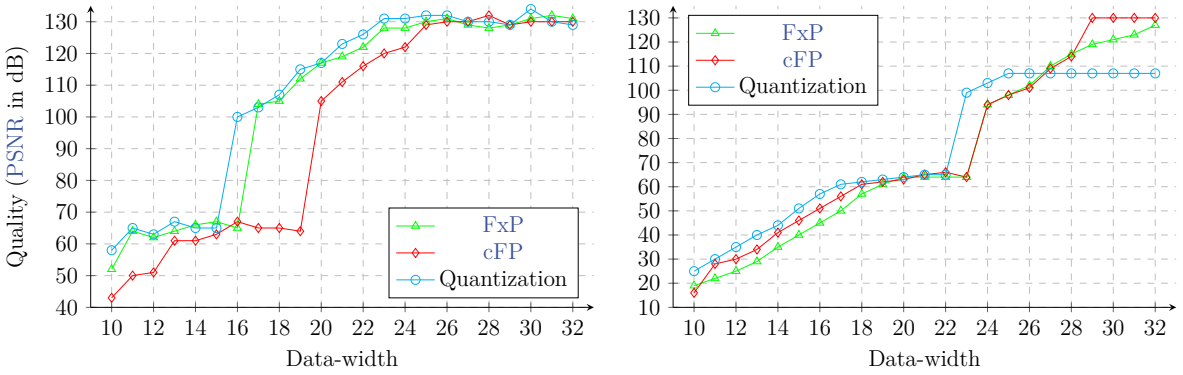
1. SEP Pipeline Imaging - GitLab: https://gitlab.com/ska-telescope/sdp/sep_pipeline_imaging

well as static buffers corresponding to the sky model and the point spread function used for deconvolution.

For this use-case, the impact of the bit-width and the representation of data is measured for each buffer independently.

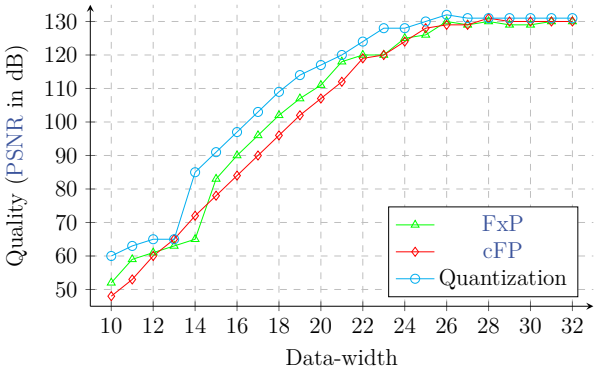
The results of this analysis are shown in Figure 5.15i. Figure 5.15a and Figure 5.15b correspond to the point spread function and the sky model, Figure 5.15c to the input of the *gridding* actor, Figures 5.15d to 5.15g are internal to the *gridding* actor, Figure 5.15h to the output of the *gridding* actor which is also the image output, and Figure 5.15i to the output of the *DFT* actor.

Graphs of Figure 5.15 show that every data representation is not fit to every buffer, and that the behaviour of the output PSNR can vary wildly depending of the data-width and the data representation used within the AxB. While Figure 5.15c, Figure 5.15g or Figure 5.15i show that different data representation can have similar results, Figure 5.15d and Figure 5.15e show that there can be clear incompatibilities depending of the content of the buffer, even when the AxB is properly dimensioned. Note that the PSNR dip in Figure 5.15h when using an AxB on 18 bits with FxP representation is an edge case where an additional erroneous source is detected during the calibration cycle, resulting in a slightly incorrect antennae gain calibration. This leads to a 65dB final result instead of the expected 100-ish dB.

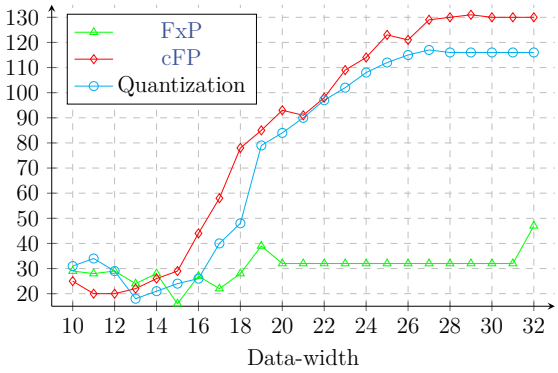


(a) PSF

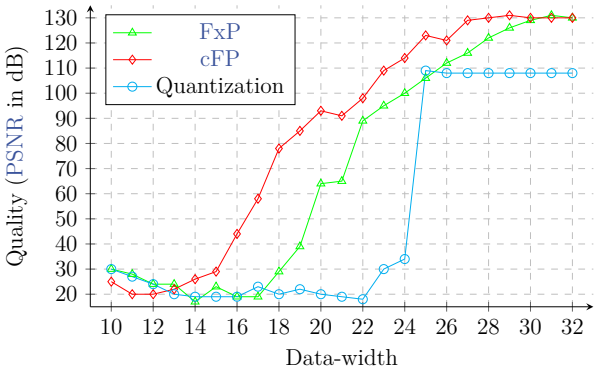
(b) vis_uvw



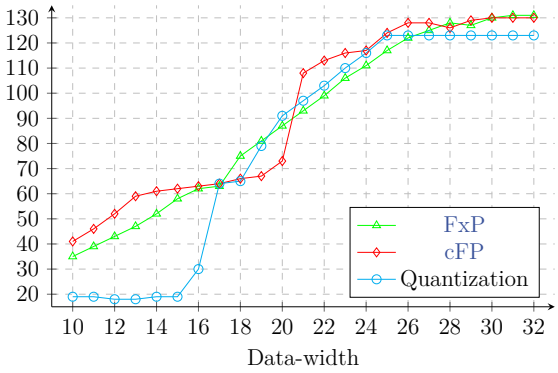
(c) Gain



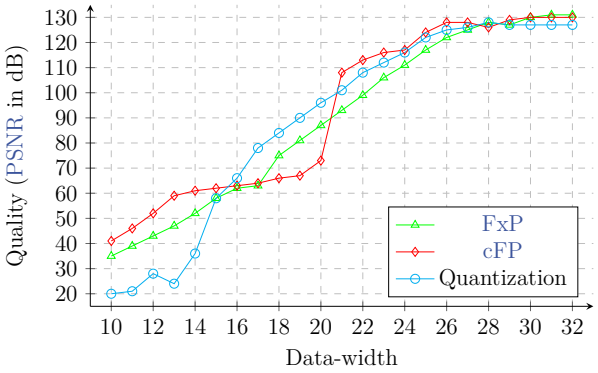
(d) Gridding



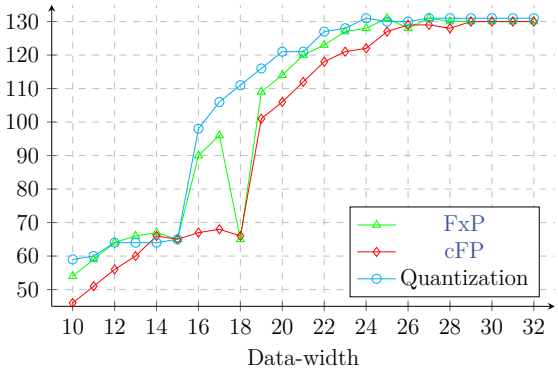
(e) FFT C2C



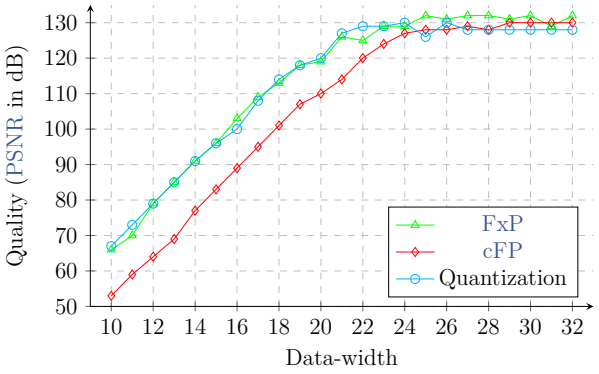
(f) FFT



(g) FFT C2R



(h) Conv Correction



(i) DFT

Figure 5.15: PSNR dependant of AxB data-width on individual parts of the SDP chain.

Conclusion

This chapter shows how data processing applications can be resilient to data-width reduction, as well as how alternative data representations can be used to further this resiliency. Three alternative data representation are presented: the FxP representation, the cFP representation, and a variation of the uniform quantization encoding. The expected limitations and accuracy losses are detailed. There is no representation appearing to be universally better than others in every situation. The AxB concept presented in this chapter is design to target entire data buffer by storing its content with an alternative data representation using a well-suited format. The following chapter will focus on how the implementation of the AxB can enable memory footprint reductions.

CHAPTER 6

Implementation for Approximate Buffer

Introduction

Chapter 5 showed how the concept of AxB is able to change the storage representation of data in applications to reduce the number of bits encoding the information, while maintaining a sufficient output quality. Moreover, as an AxC technique, it offers a tunable trade-off between quality and data bit-width. However, Chapter 5 does not provide a reference implementation model enabling the materialization of the memory resources reductions advertised.

This chapter provides examples of implementations of the AxB concept to reduce the memory resources requirements of data processing applications depending on the nature of the target architecture. The main focus is set around CPU-based architectures, but the concept of AxB can also be exported hardware design and especially to Field-Programmable Gate Array (FPGA) applications.

Figure 6.1 shows a simplified representation of the data storage in memory with AxBs. Data tokens on 20 bits are concatenated in memory to effectively reduce the footprint of the buffer.

One of the objectives of AxBs is to be as least intrusive as possible with few modifications to the original application. Indeed, while methods such as FxP arithmetic would

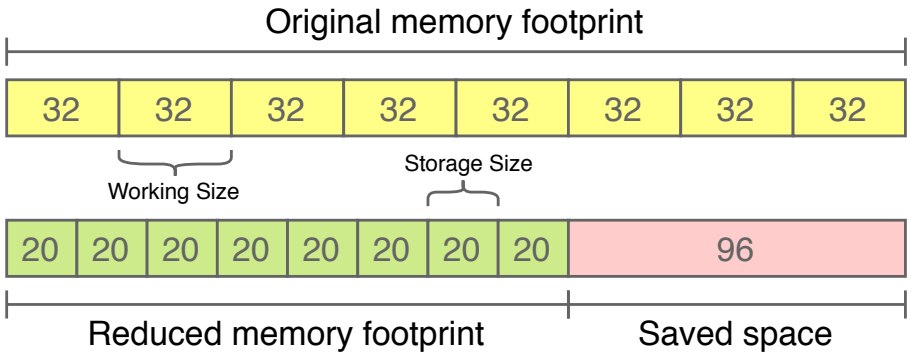


Figure 6.1: Simplified representation of an AxB, with 20 bits wide data tokens being concatenated in memory, compared to a bit-width of 32.

require the algorithm to be wildly redesigned, the use of AxBs only require modifications on how the data is read and written in memory.

The concept of AxBs has initially been developed for data processing applications described with Synchronous Dataflow (SDF)-based Models of Computation (MoCs) (Section 3.3) due to the inherent separation between computations and data transactions, but is not limited to this programming model. Consequently, support for AxBs has been added to the Parallel and Real-time Embedded Executives Scheduling Method (PREESM) application development framework, for both CPU and FPGA architecture. Support within PREESM for CPU architectures is limited to shared memory platforms, and as of writing to single core execution.

Section 6.1 details the implementation logic to enable data to be read from and written to memory with an arbitrary bit-width in a relatively seamless manner. Section 6.2 shows how the general concept of AxBs can be used on FPGAs to reduce the use of specific hardware resources.

6.1 Software Implementation for CPU

To correctly handle data insertions and extractions from an AxB, relevant parameters concerning the data representation and format within the AxB is kept as metadata. This metadata is used to keep track of the original bit-width of the data, the storage bit-width, the size of the buffer in data-tokens, as well as information on the specific format used for storage. This metadata represents an overhead of around 20 bytes per AxB.

In case of a C-like syntax, the data-writing process would be modified from `array[index] = value;` to `AxB_Write(&axb, index, value);`. Data accesses through AxBs can be made transparent with programming languages allowing for operator overloading such as C++, where simply providing an overload for the array subscript operator of the AxB class is enough. The only other modification required is to provide the computation function with a pointer to the AxB in place of a pointer to the original buffer.

6.1.1 Data conversion

For an efficient use of AxBs, data-tokens can be converted into one of the alternative representation described in Chapter 5.

The conversion from FP32 to FxP and back is explained by Equations (2.2) and (2.4) in Section 2.2.2. Similarly, the quantization encoding and decoding is detailed by Equations (5.3) and (5.4) in Section 5.5. The conversion result needs to be shifted left, as the insertion process reduces the token width by removing LSBs. Inversely, the token retrieved by the extraction process needs to be shift right.

The conversion from FP32 to cFP (and back) consists in a bit manipulation to move and resize the exponent and mantissa field and is therefore more verbose, as shown in Listing 6.1.

```
1  int32_t convertFrom_fp32_to_cFP(AxB* axb, float data){
2      // Convert FP32 data to cFP(s.e.m.b)
3      if (data_in == 0)
4          return 0;
5
6      uint32_t* ptr_fp_bit = (uint32_t*) &data;
7
8      // If sign bit is stored, then fetching it, else leave
          sign at 0 to negate impact
9      uint32_t s = 0;
10     if (axb->s == SIGN_UNDEF)
11         s = *ptr_fp_bit & 0x80000000;
12
13     // Get exponent e, on 8 bits or less
14     uint32_t e = ((*ptr_fp_bit >> 23) & 0xff) - 127;
15
16     // Apply AxB exponent bias b
17     e += axb->b;
18
19     // Crop exponent e MSB
20     e &= ((1 << (axb->e)) - 1);
```

```
21      // Set exponent e in position
22      e <<= (axb->m + (axb->workingW - axb->storageW));
23
24      // Get mantissa m
25      uint32_t m = *ptr_fp_bit & 0x007FFFFF;
26      // Crop mantissa m
27      m >>= (FP32_MANTISSA_SIZE - axb->m);
28      // Set mantissa position
29      m <<= (axb->workingW - axb->storageW);
30
31      // Assemble cFP
32      *ptr_fp_bit = s | e | m;
33
34      return *temp;
35  }
```

Listing 6.1: Convertion function from FP32 to $cFP_{s.e.m.b}$.

The different fields are isolated, resized as needed, and assembled into the new format to be store into the AxB.

Performing computations with data from AxBs may produce results outside the range used for AxB format determination, especially when using quantization. To solve this issue, a clamping operation may optionally be performed before conversion.

6.1.2 Insertion/extraction

AxBs store data of arbitrary width, far from the conventional byte-based access schemes of processor-based systems. Data is then concatenated in memory, allowing the memory footprint reduction.

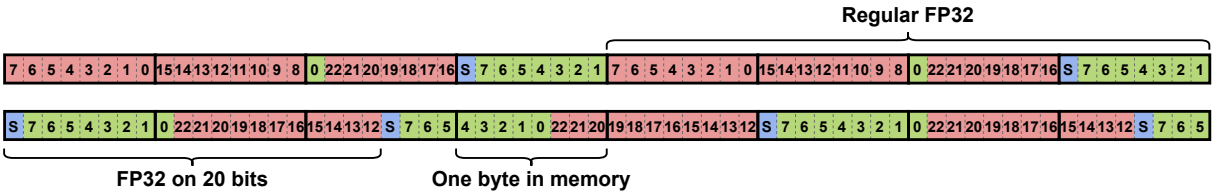


Figure 6.2: Memory representation of 20-bit truncated FP32 data concatenation, along with regular FP32 storage.

Figure 6.2 shows an example of how 20-bit truncated FP32 data can be concatenated in memory compared to regular FP32 storage. The top line shows 2 FP32 data stored on 8 bytes, and the bottom line shows 3 20-bit truncated FP32 concatenated with the first

4 bits of a 4th one. As it is not possible to perform 20-bit wide memory operations on a conventional CPU architecture, specific procedures for data insertion and extraction from an AxB need to be devised.

Insertion Process

The process of writing a data-token into an AxB is not straightforward, as data accesses to memory do not have a granularity down to a bit, and the width of these accesses is limited to 2^N bytes.

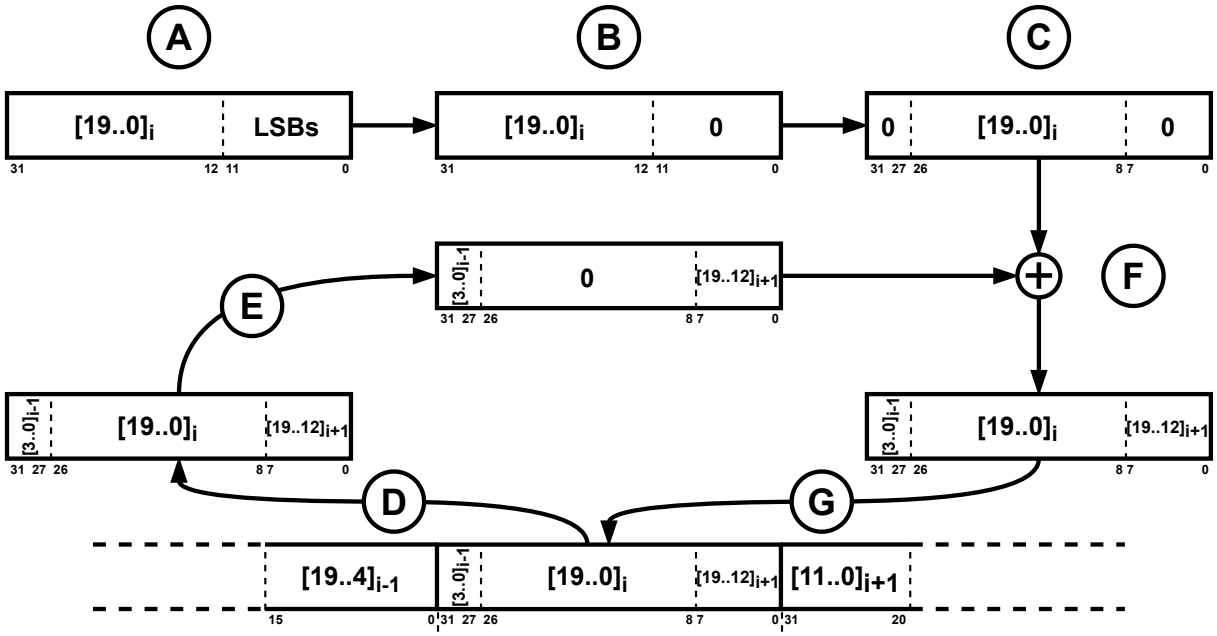


Figure 6.3: Step-by-step process for data insertion into an AxB, for a bit-width of 20.

Figure 6.3 shows the process of inserting a data-token into an AxB. This example considers an insertion of a 20-bit data token. Actual memory accesses are still performed with a width conventionally supported by CPUs. In this case, memory operations are performed by 4-byte wide blocks, with a 2-byte alignment. Although misaligned memory accesses can in theory result in performance penalties, tests showed that performing a single *blockWidth*-wide access on a $\frac{blockWidth}{2}$ alignment was equivalent, if not slightly better, to 2 consecutive $\frac{blockWidth}{2}$wide accesses with a $\frac{blockWidth}{2}$ alignment.

Because data-tokens are concatenated into the AxB, a single block of memory (*i.e.* 4 consecutive bytes) contains bits from multiple data-tokens. When writing a data-token

to memory, it is essential that neighbour bits belonging to adjacent data-tokens stay unchanged.

To ensure that only bits belonging to the data-token are modified, the insertion process is decomposed in 7 steps detailed below:

A) The data-token i is provided to the insertion function. The data representation is irrelevant as the 20 bits of the data-token are treated as a senseless payload.

B) LSBs are removed through bit-masking so that only the 20-bit payload remains.

C) The 20-bit payload is shifted to the position it will hold in the memory block.

D) The block is read from memory.

E) Bits from the block to be replaced by the payload are zeroed through bit-masking.

F) The payload is inserted into the block.

G) The block in written back to memory.

Because of the byte-ordering in memory, little-endian architectures need to perform a `bswap` operation on the result of (**C**) as well as on the bit-mask used in (**E**).

The non-atomicity of this process, especially because of the block read (**D**), block modification (**E** and **F**) and block write (**G**) steps, enables potential race conditions for multithreaded execution. These race conditions can be prevented by the use of atomic operations, replacing steps (**D**) to (**G**) by an atomic `and` with the bit-mask to clear the bits to replace, followed by an atomic `or` to write the payload. This results in a significant performance penalty because of the increased number of memory transactions from 2 to 4, as well as latency from lock acquisition for the atomic operations. The other solution is to accept the possibility race conditions happening (which could even be considered as an additional AxC technique implementation from Section 2.1.3.2). These race conditions are likely to happen in case of multithreaded random memory writes, but are virtually a non-issue when each thread only interacts with a segment of the AxB. When the workload is parallelized such as each of the N^{th} threads has only access to an umpteenth of the AxB, a race condition can only occur if a thread is writing its lasts data-tokens while the next thread is writing its firsts, which is highly unlikely to happen. Program-code generated by the PREESM application development framework fit this scenario.

Extraction Process

The process of reading a data-token from an AxB is simpler.

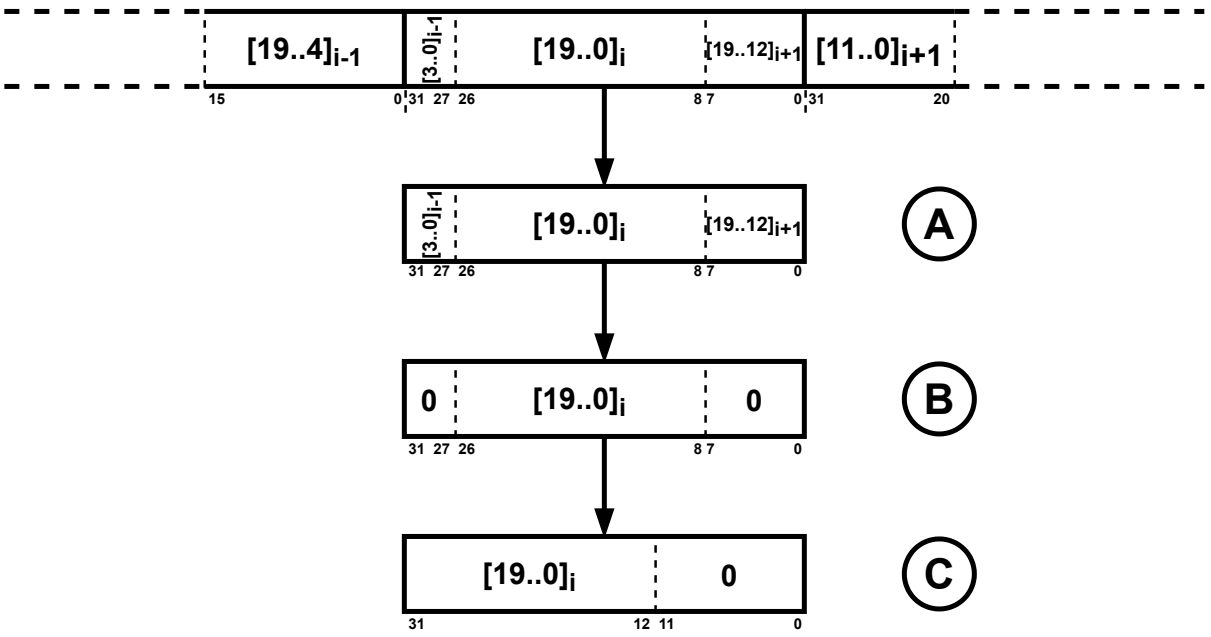


Figure 6.4: Step-by-step process for data insertion into an AxB, for a bit-width of 20.

Figure 6.4 shows the process of extracting a data-token from an AxB. This example still considers the extraction of a 20-bit data-token. Comments regarding memory alignment made during the explanation of the *Insertions Process* are still applicable. There is however no longer any issues regarding modifications of bits belonging to neighbour data-tokens.

The extraction process is decomposed in 3 steps detailed below:

A) The 4-byte wide memory block containing the 20-bit payload is read from memory.

B) Bits from neighbour data-tokens are removed through bit-masking so that only the 20-bit payload remains.

C) The 20-bit payload is shifted left to its final position.

The extracted data-token can then be converted from the storage representation to the representation used for computations.

As with the insertion process, a `bswap` operation is required on little-endian architectures at the end on step (**A**).

The bit-mask operation from step (**B**) may optionally be skipped, causing MSBs from the next data-token to be used as LSBs. This causes a slight quality drop with no measurable performance impact. Consequently, the operation from step (**B**) is kept.

The time to insert and extract data-tokens with AxBs has been measured. This measurement is done with the knowledge that no particular specific and dedicated effort has been done for optimizations. A read and a write operation in an AxB takes approximately an additional 2.1ns without alternative data representation. The conversion from FP32 to another representation adds 0.6 to 1.6ns to the write delay, while the conversion from an alternative representation back to FP32 adds another 0.1 to 2.9ns to a read operation. These additional delays on data transaction are expected to have more noticeable impact on data processing applications with a lower arithmetic intensity. Enhancement to the insertion and extraction processes and data conversions could be performed to reduce the additional delay, either with specific code optimizations or with hardware implementation, but these are not covered in this thesis.

6.1.3 Experimental Results

The memory footprint reductions enabled by the use of AxBs are demonstrated on the application set presented in Chapter 4. The performance hit, in the form of longer execution time, is mainly dependent on the number of AxBs used and on the arithmetic intensity of the application itself.

6.1.3.1 2D Wavelet Filter

The Wavelet Filter is applied on every frame of a set of video sequences. Only the luminance component `Y` is processed, chrominance components `U` and `V` are left untouched.

Figure 6.5 shows the memory footprint of the 2D wavelet filter with the corresponding output quality. Every buffer of this application is an AxB (aside from buffers holding chrominance components `U` and `V`), using the same bit-width and the same representation, but with a format adapted to their content, for a total of 15 AxBs. The configurations of AxBs used for this plot are the best result from the quality study in Section 5.6.1 (Figure 5.9). The desired output quality can be tuned depending on the memory footprint constraints, maintaining for example a PSNR above 60dB with a 40% reduction of memory footprint. The large number of AxBs coupled with the low arithmetic intensity lead to a particularly high performance penalty of around 56%.

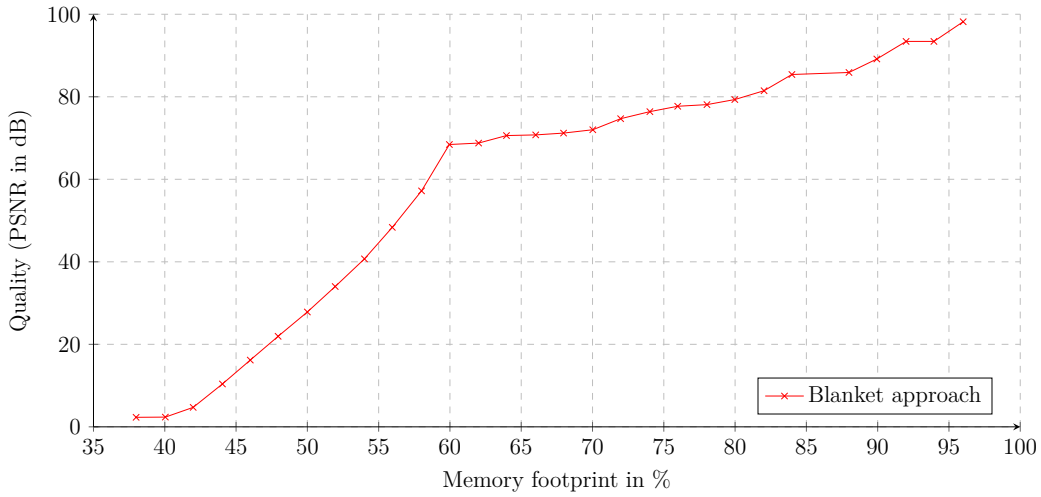


Figure 6.5: Graph of the memory footprint of the 2D wavelet filter with the corresponding output quality.

6.1.3.2 SqueezeNet

The SqueezeNet CNN has a memory footprint of around 9.2 MB. Results from Section 5.6.2 show that using AxBs to store all model weights with the same bit-width and format enables a preservation of the TOP-1 accuracy, using 16 bits and the FxP representation (Figure 5.13), for a memory footprint reduction of 25.5%, from the initial 9.2 MB down to 6.85MB.

By combining this result with the data-width reduction from using AxBs on inter-layer data (Figure 5.12), the total memory footprint of SqueezeNet could potentially be reduced to 5.07MB, a 44.9% reduction. This is however not directly possible, as adding additional AxBs aggravate the propagated error. This would require additional specific measurements.

A better solution is to reduce the amount of buffer considered for an AxB implementation. Figure 6.6 shows the memory footprint needed for each part of the SqueezeNet neural network.

Figure 6.6 shows that while some actors require a significant amount of memory (*maxPool1*, *conv1*, *fire3*...), most of them have comparatively small memory requirements (from *fire4* onward). This shows that the highest memory requirement is reached by the *maxPool1* layer, followed by *conv1* and *fire3*. Using AxBs on input/output buffer of layer from *fire4* onward is therefore useless to achieve a memory footprint reduction. Contrary to model weights which are kept in memory for the whole duration of the execution, input

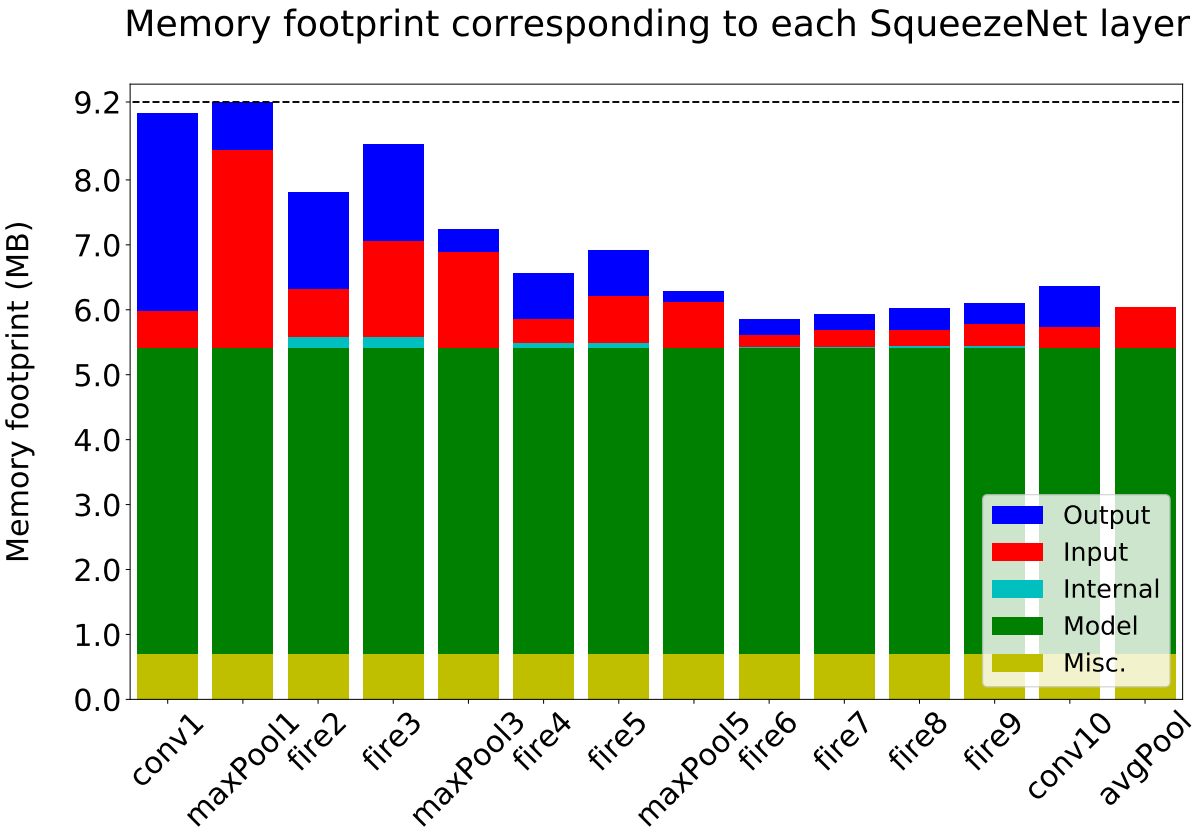


Figure 6.6: Memory footprint corresponding to different parts of the SqueezeNet neural network.

and output buffers of each layer are allocated during a limited time. This implies that a broad approach consisting of using the same AxB parameters for every interlayer buffer is inadequate.

A better solution is to reuse a similar approach as before, but this time specifically selecting the buffer which are the most susceptible to have an impact on the overall memory footprint. The interlayer buffers are still sharing the same AxB parameters, but this time only buffers from *conv1* to *fire3* are considered.

Table 6.1 shows the memory footprint dependent on both the bit-width and format used for model weights AxBs as well as by using the same AxB parameters for input and output AxBs from layer *conv1* to layer *fire3*, with the constraint of producing the same classification results as the regular version of SqueezeNet.

The lowest memory footprint is achieved by storing the model on 16 bits into an AxB using the $Q_{2.14}$ format and the working data on 14 bits in AxBs using the $cFP_{+.5.9.15}$

Model \ Data	Truncation		FxP		cFP		Quantization	
Truncation	18 / 20	FP18	18 / 20	$Q_{12.6}$	14 / 20	$cFP_{+.5.9.15}$	15 / 19	15
	FP20	5.77MB	FP20	5.77MB	FP20	5.30MB	FP19	5.42MB
FxP	18 / 16	FP18	18 / 16	$Q_{12.6}$	14 / 16	$cFP_{+.5.9.15}$	16 / 16	16
	$Q_{2.14}$	5.19MB	$Q_{2.14}$	5.19MB	$Q_{2.14}$	4.71MB	$Q_{2.14}$	4.95MB
cFP	18 / 17	FP18	18 / 17	$Q_{12.6}$	14 / 17	$cFP_{+.5.9.15}$	15 / 17	16
	$cFP_{1.5.11.24}$	5.33MB	$cFP_{1.5.11.24}$	5.33MB	$cFP_{1.5.11.24}$	4.86MB	$cFP_{1.5.11.24}$	5.10MB
Quantization	16 / 18	FP16	18 / 19	$Q_{12.6}$	14 / 19	$cFP_{+.5.9.15}$	15 / 20	15
	18	5.24MB	19	5.63MB	19	5.15MB	20	5.42MB

Table 6.1: Minimal data-width and memory footprint of SqueezeNet dependant on data representation, while still reaching 100% accuracy compared to SqueezeNet with FP32. AxBs are used from the input image up-to *fire3* actor, including its output, with the same configuration.

format (the + indicates that there is no signbit stored, but all values within the AxB are considered positive when converted out of the AxB).

The best results obtained from this approach allows a reduction of the SqueezeNet memory footprint to 4.71MB, a 48.8% reduction. The outcome of this approach can be used as a starting point for a more in-depth tuning of AxB parameter.

	Model	Input to conv1	conv1 to maxpool1	maxpool1 to fire2	fire2 to fire3	fire3 to maxpool3	fire4 to fire5
OG Size	4.70MB	0.57MB	3.06MB	0.74MB	1.48MB	1.48MB	0.71MB
New Size	2.35MB	0.27MB	1.15MB	0.28MB	0.60MB	0.60MB	0.53MB
Data-width	16	15	12	12	13	13	24
Format	$Q_{2.14}$	$Q_{9.6}$	$cFP_{+.5.7.15}$	$cFP_{+.5.7.15}$	$cFP_{+.5.8.15}$	$cFP_{+.5.8.15}$	$cFP_{+.6.18.31}$

Table 6.2: Minimal data-width and memory footprint of SqueezeNet dependant on data representation, while still reaching 100% accuracy compared to SqueezeNet with FP32. AxBs are used from the input image up-to *fire3* actor, including its output, as well as the output of *fire4* actor.

Table 6.2 shows the AxB parameters used to reduce the memory footprint of SqueezeNet down to 4.47MB, for a 51.4% reduction. This result was achieved by tuning the parameters of AxBs by hand, significantly increasing the time required to reach a better result than the configurations from Table 6.1. The performance penalty in this application is around 7%.

The memory footprint of SqueezeNet could be reduced even further by separating the AxB configuration of model weight in 10 individual sets of parameters, but this separation implies an explosion of the design space. This exploration is not reasonably doable manually and requires a specific Design Space Exploration (DSE) method, presented in Chapter 7.

6.1.3.3 Science Data Processor

The impact of each AxB with different data representation, individually applied on different part of the SDP pipeline, is shown on Figure 5.15. This is aimed at showing the individual impact on the output quality of using AxBs on different part of the pipeline and can be used as a guide for the implementation of multiple AxBs at once.

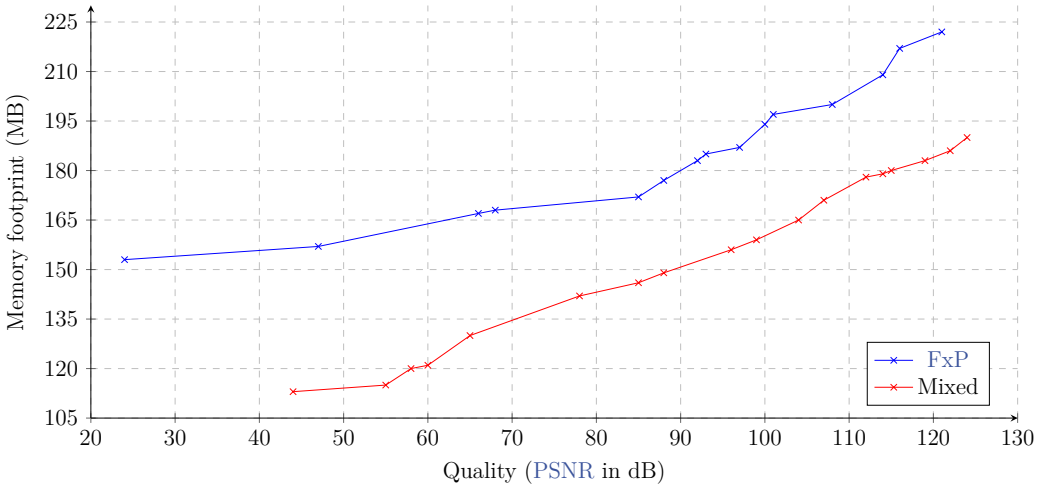


Figure 6.7: Graph of the memory footprint dependant of the output quality.

Figure 6.7 shows memory footprint reduction dependent of the output quality on the SDP use case (Section 4.3).

The memory footprint of the base version of the SDP is 235 MB.
When using only FxP representation, the memory footprint can be reduced to 222 MB (-5.5%) with a PSNR of 121dB and down to 170 MB (-27.7%) while keeping a PSNR value of 70dB. From this point, reducing data precision any further prevents the SDP Imaging Pipeline from correctly finding the expected number of sky sources, leading to the generation of erroneous intermediate *dirty images*, hence the precision dip on the blue curve.

Using multiple data representation at the same time enable a significant improvement over blue curve, with a memory footprint of 186 MB (-20.85%) for a PSNR of 122dB, and a footprint of 142 MB (-39.57%) for a PSNR of 78dB.

As the SDP Imaging Pipeline is a complex application, it is not possible to precisely predict the extent of the quality degradation that comes with the use of Approximate Buffers. Reducing data-storage accuracy of one buffer can lead to a significant decrease in the output quality, but applying the reduction to a second one can, in some case, almost completely negate this degradation. Fine-tuning the data-width in complex applications can be a lengthy but necessary process. This method leads to an execution time between 0.6 and 4.3% longer in this application, depending of how many buffers are impacted.

Obtaining these results required a hand-made DSE guided by the findings from Figure 5.14, emphasizing the need for an automated DSE method.

6.2 Hardware Implementation for FPGA

This section is specifically talking about implementation of the AxB concept for SDF-based applications on Xilinx FPGAs.

In the context of implementation of dataflow applications on an FPGA, every actor is executed concurrently. In accordance with the SDF MoC, an actor can start its execution when enough data-tokens are available on its inputs.

On CPU-based implementations of SDF graphs, both input and output Fifos require to be allocated in their entirety in the form of memory arrays, where data-tokens are considered available only when the actor finishes its execution. On FPGA-based implementations, data-tokens are produced progressively and can be consumed right-away, lowering the memory requirements. Consequently, Fifos on FPGAs are not required to hold the total amount data-tokens produced by an actor execution, but are rather only required to be deep enough to hold the maximal amount of token that the consumer actor will be unable to process right away [Vlu+19; Gho+12] (in the context of a throughput maximization approach).

Fifos on FPGAs can use different kind of hardware resources, such as Flip-Flops (FFs) or Block RAMs (BRAMs). Fifos required to hold a large-enough amount of data are implemented with BRAMs. BRAMs are discrete components of an FPGA used to store data. A BRAM is a dual-port RAM module capable of holding 18k bits (or 36k depending on configuration) on Xilinx FPGAs. As all other kinds of hardware resources

(Digital Signal Processors (DSPs), FFs, Look-Up Tables (LUTs), ...), there is a finite number of BRAMs available within an FPGA. BRAMs are in general used to store a large amount of data, *i.e.* more than a kilobit, such as data from peripherals, read-only data, large LUTs (not to be confused with hardware LUTs), Fifos, etc. BRAMs can also be used to transfer data between clock domains. The dual-port nature of BRAMs enable them to simultaneously perform read and write accesses, making them well-suited for Fifo operations. BRAMs have the particularity of having customizable width and depth. For example, a BRAM containing 18k bits can be setup with a depth of 1k with 18-bit wide data, 2k with 9-bit wide data... Multiple BRAMs can be pooled together to increase the storage capacity.

In this context, the aim of implementing the AxB concept on FPGAs is to reduce the number of BRAM allocated for Fifos. The details on how many BRAMs are allocated to a Fifo depending on its depth and on the width in bit of a data-token is not provided by the manufacturer, but has been reversed-engineered in the High-Level Synthesis (HLS) compiler Vitis HLS.

The amount of BRAM allocated for a Fifo is given by the equation:

$$\texttt{BRAM} = \left\lceil 2^{\lceil \log_2(depth) \rceil} \times \frac{tokenWidth}{BRAM\_size} \right\rceil \tag{6.1}$$

with *depth* the maximal amount of data-token the Fifo is expected to hold, *tokenWidth* the width in bits of a data-token, and *BRAM_size* the size in bit of a BRAM.

Figure 6.8 shows the BRAM allocation dependent on the Fifo depth and the token-width corresponding to Equation (6.1).

This equation indicates that the depth of the Fifo is a major factor of the BRAM allocation. More specifically, because the depth is rounded up to the next power of 2, an increase of the Fifo depth by a single token (from $depth = 2^N$ to $depth = 2^N + 1$) will lead to doubling the number of BRAMs allocated. Moreover, Equation (6.1) has some specific unexpected quirks. By default, a BRAM has a size of 18 kilobits ($BRAM\_size = 18 \times 1024$ bits), unless the Fifo depth is superior or equal to 4096, in which case only 16 kilobits are used per BRAM. Additionally, if data-tokens within a Fifo have certain bit-width, the amount of BRAM allocated is rounded to the next even number. The bit-width values concerned by this rounding rule are 13, 21, 22, and 28 and up, but only if the Fifo depth is in the interval $]2048; 4096]$.

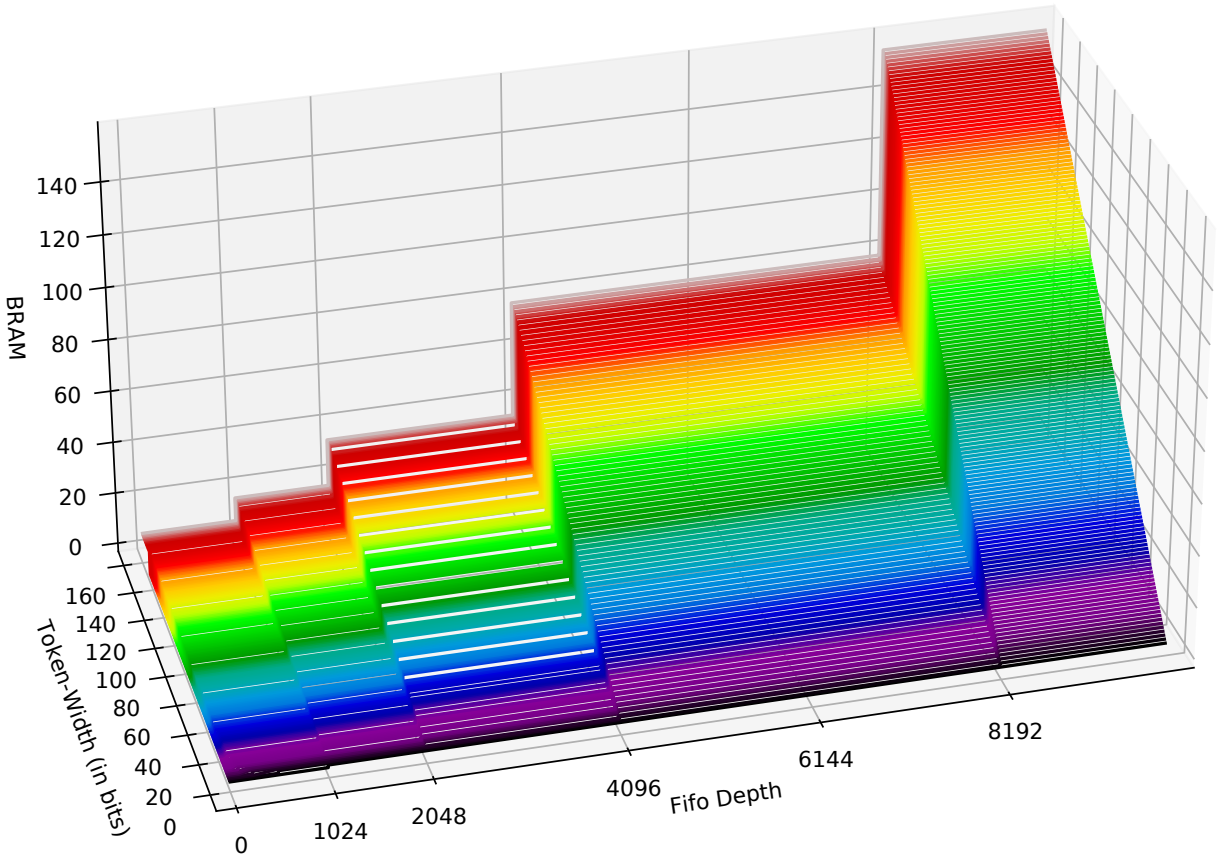


Figure 6.8: BRAM allocation dependent on the Fifo depth and the token-width corresponding to Equation (6.1).

Data-Packing

The concept of AxBs consisting in concatenating data-token of a whole buffer in memory is only relevant on processor-based architecture, where bits of memory are grouped in bytes by design. However, the general idea can still be applied on FPGA platforms.

As shown on Equation (6.1), the depth of a Fifo has a huge impact on the BRAM allocation, as the addressing of the content is bound to the next power of 2. The idea is to group data-tokens into data-packets. Using a BRAM with a bigger width leads to a reduction of its depth.

Consequently, by packing P data-token into a data-packet, the corresponding depth is divided by this factor P. The goal is to find the packing ratio P leading to a packed-depth as close as possible but still below $2^N + 1$ threshold. Additionally, using a packed-depth inferior or equal to 4096 enables the use of the whole 18k bits of a BRAM, instead of 16k as explained above.

For use of data-packing in an application described with an SDF-based MoC, special *Packing* and *Unpacking* actors can be inserted on each side of the Fifo to pack. With the packing ratio P, the *Packing* actor consumes 1 data-token per clock cycle and produces a data-packet every P^{th} cycle. The *Unpacking* actor does the opposite, consuming a data-packet every P^{th} cycle and producing a data-token every cycle.

The addition of these *Packing* and *Unpacking* actors on a Fifo creates a delay equivalent to the additional number of clock cycles required to produce the data-packet (by consuming P data-tokens), and an additional clock cycle to unpack the first data-token. The added latency is equal to $P + 1$. Parallel branches may need to account for this additional latency by increasing the depth of one of their Fifos by $P + 1$, with a process similar to retiming [Liv+07].

To ensure a proper execution of a graph iteration, the packing ratio P needs to be a divisor of either the production or consumption rate.

The process of deploying a data processing application described with an SDF-based MoC on FPGA is shown inFigure 6.9.

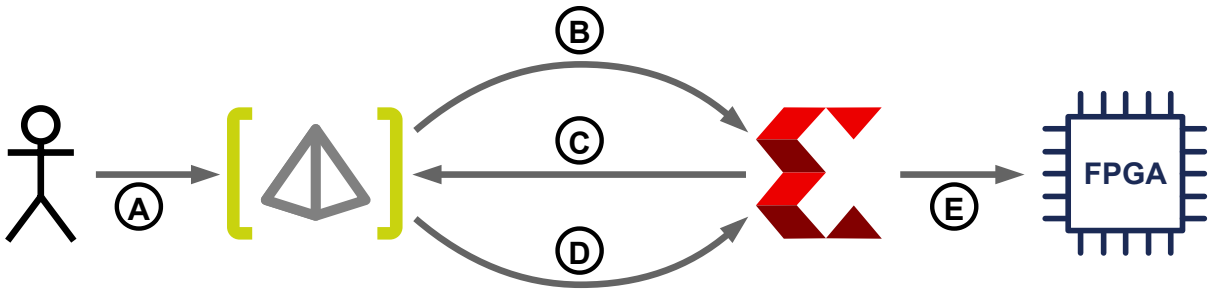


Figure 6.9: Process for deployment of SDF-based application on FPGA using PREESM and Vitis HLS.

The deployment from SDF graph to FPGA requires a back and forth between PREESM and Vitis HLS:

A) The application designer modelizes the graph in PREESM.

B) PREESM generates the HLS code with dummy execution times and Fifo depths.

C) Vitis HLS analyses the HLS code to determine precise actors' execution times.

D) PREESM uses these precise execution times to determine the appropriate Fifo depths, with Data-Packing if desired.

E) Vitis HLS uses this newly generated HLS code for FPGA deployment.

This data-packing method has been implemented in the PREESM application development framework.

6.2.1 Results on FPGA

In this experiment, only the FxP representation is considered, as it is inherently well suited for FPGAs computations. The application chosen to demonstrate the data packing method is a 2D-DWT, constituting the first half of the 2D Wavelet Filter application described in Section 4.1 and is shown on Figure 6.10. The image resolution is 1920x1080.

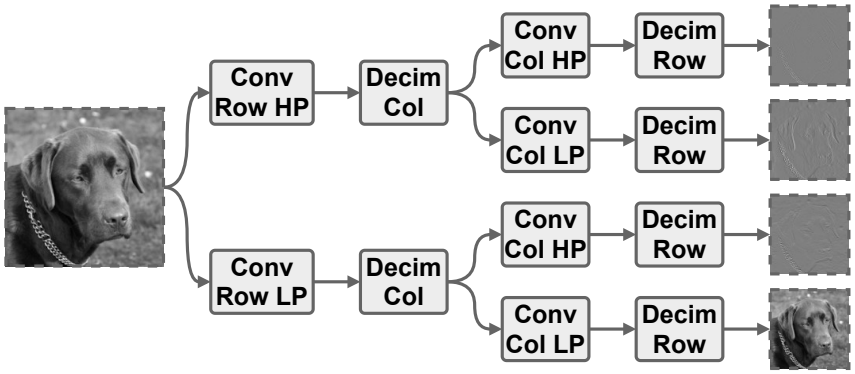


Figure 6.10: Simplified representation the 2D-DWT application.

This experiment combines the gains from the use of Data-Packing with the AxC paradigm to give the possibility to trade off output quality for hardware resources reductions. The approach is the same as in Section 6.1.3.1, with the same bit-width used for all FIFOs used with Data-Packing.

Figure 6.11 shows the amount of BRAMs required for FIFOs dependent of the output quality, with and without Data-Packing. The number of BRAMs used with Data-Packing is up to 25% smaller.

The use of Data-Packing enables a reduction of the BRAM usage at the cost of a latency increase along with other types of hardware resources.

Table 6.3 shows the hardware resources usage with and without Data-Packing for 10-bit wide data-tokens for the whole application, FIFOs and actors, corresponding to a PSNR value of 50.81dB on Figure 6.11. The use of Data-Packing enables a global BRAM utilization reduction of 13.6% at the cost of a small increase in FFs and LUTs usage.

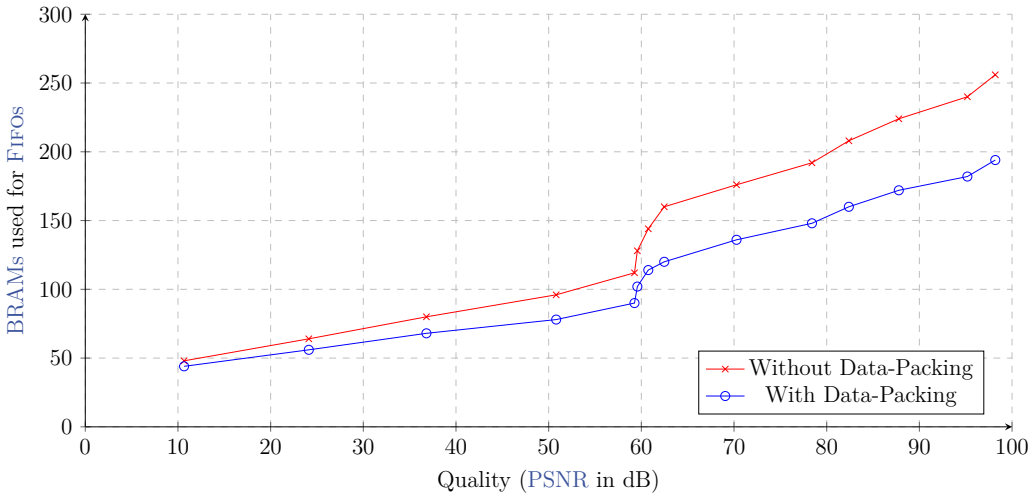


Figure 6.11: Graph of the quantity of BRAMs used for FIFOs with the 2D-DWT dependent on the output quality.

	BRAMs	DSPs	FFs	LUTs
Without Data-Packing	132	90	34682	56220
With Data-Packing	114	90	36540	58525
Difference	-13.6%	0%	+5.1%	+3.9%

Table 6.3: FPGA hardware resources usage with and without Data-Packing with 10-bit wide data-tokens for both FIFOs and actors, according to Vitis HLS.

Additionally, as previously stated, the use of Data-Packing adds a latency of $P + 1$ cycles for each packed FIFOs on the critical path, with P the packing ratio. In the example of Table 6.3, the packing ratio varies between 5 and 8, with a total additional latency on the critical path of 15 cycles. For comparison, the execution time of the *Conv Row HP* actor takes 2,079,917 cycles, making the additional latency from Data-Packing negligible.

Conclusion

This chapter details the implementation of the AxB concept on both hardware and software architectures. It shows how AxBs can enable significant memory footprint reductions while maintaining a high quality with various types of data processing applications. The benefit of AxBs is demonstrated with memory footprint reductions between 40 and 50% depending on the application.

This chapter also explains how the concept of AxBs can be applied on FPGAs in the form of Data-Packing. This technique enables a reduction of the BRAMs usage, a hardware resources dedicated to memory storage, at the cost of a small increase in general purpose hardware resources usage. Data-Packing also has a negligible impact on latency.

This results emphasize the need for a specific DSE method to automate the configuration of AxBs by minimizing the memory footprint while maintaining a quality constraint, as doing so manually can take from days to weeks depending on the complexity of the application.

The techniques presented in this chapter has been integrated into the PREESM application development framework.

CHAPTER 7

Design Space Exploration for Approximate Buffers

Introduction

Chapter 6 shows that the use of AxBs, a data-driven AxC technique applied on data buffers of a data processing application, enables a reduction of the memory footprint, traded-off by a controlled degradation of the overall output quality. AxBs are implemented in `C` code, by replacing data accesses from arrays `tab[i]` by a function call. Representing data using alternative data representations and an arbitrary bit-width enables AxBs to concatenate data-tokens in memory while still maintaining an acceptable accuracy. It enables a memory footprint reduction of around 40% on the application examples with little to no degradation of the output quality on CPU implementations, as well as a reduction of up to 25% of the hardware resources for Fifo storage on FPGA.

However, the size of the design space to explore increases exponentially with the expression N^{B^R}, with N the number of bit-width possible (usually 32), B the number of AxBs, and R the number of alternative data representation to consider. Consequently, the AxC-oriented AxB Design Space Exploration (DSE) cannot be done exhaustively and is thus usually performed empirically. The selection of buffers to consider in this approach may take days, up-to weeks on complex applications.

This chapter presents an automatic and generic DSE method to optimize the bit-width and format of data for use with AxBs. The proposed DSE method leverages the

specific memory allocation of FIFOs to find a configuration of AxBs reducing the memory footprint while maintaining a given quality constraint.

This contribution has been presented at the International Conference on Application-specific Systems, Architectures and Processors (ASAP) 2022 [MNM22].

7.1 State of the Art on Design Space Exploration for Approximate Computing

Our contribution is a data-driven AxC technique considering precision optimization. The goal is to find the binary representation of manipulated data with the smallest number of bits, while preserving the accuracy of the final result.

DSE methods have been proposed to ease the implementation of AxC-based techniques. A DSE method is proposed in [EWT18] for boolean function approximation on FPGAs, boasting up-to 20% resource reduction for a maximum error rate of 0.05%. In [Cas+20], a set of tool is presented to perform a DSE for the implementation of approximate operators (mainly additions and multiplications). This allows a HLS tool to properly generate approximate accelerators while maintaining a quality constraint. Energy savings up-to 30% are advertised. A simulation-based DSE is presented in [Pai+20], for the design of approximate adders within video encoding application. The presented technique is evaluated on the H.265/HEVC coding standard, showing up-to 45% power reduction with a minimal quality degradation. In [Bar+21], the automated framework E-𝔻𝔼𝔸 has been developed to apply AxC techniques on `C/C++` applications through source-to-source conversion. The framework is able to perform the DSE depending on the criteria to optimize, such as power, area or computation time, with regard to a quality constraint.

While current DSE methods enable the implementation of AxC-based techniques on applications, these methods are either architecture-specific or application-specific, but also does not take into account the memory footprint improvement achievable with AxC. To the best of our knowledge, there is no automatic and generic DSE algorithm for memory footprint reduction method based on data precision optimization AxC techniques like AxBs.

7.2 Automatic Approximate Buffer Configuration

7.2.1 Memory Footprint Minimisation Algorithm

This generic DSE algorithm is designed to minimize the memory footprint of an application, while maintaining the result above a specified threshold of a desired quality metric. For this purpose, the DSE algorithm is given a set of buffers for which to find the smallest bit-width and representation. This generic method relies on the AxB concept presented in Chapters 5 and 6, allowing data to be represented and stored in memory on an arbitrary number of bits, without the expected 2^n memory alignment requirements.

One of the main feature of our generic method is the distinction between **global buffers** and **local buffers**. Global buffers are buffers which are allocated for the whole duration (lifespan) of an application iteration, such as the result vector of an iterative process or a reference set of value that needs to be compared against. Local buffers are buffers that are not required for the whole duration of the application and whose memory space can be reused [Des+16], such as buffer storing intermediate results between different part of the processing pipeline. It is supposed that local buffers with non-overlapping timespan may share the same memory space.

This clear distinction between **global buffers** and **local buffers** gives the possibility to opportunistically increase the bit-width of local buffers, as long as the total memory footprint is unchanged, reducing the number of configuration to explore by ignoring obvious sub-optimal configurations. This operation is later on referred to as buffer inflation.

The proposed algorithm takes as input the quality constraint $q^{constraint}$ to fulfill, the list of global buffers n^{global} and local buffers n^{local} on which to apply the AxB technique on, the model to measure the memory footprint, and a way to evaluate the output quality q^{test}.

A first run of the application is performed to monitor the variation of values to store within AxBs. This enables the determination of the base storage format to be used for each representation, and the degradation to apply when reducing the bit-width.

Algorithm 1 shows the global view of our method and its separation into three subsections detailed in Algorithm 2, Algorithm 3 and Algorithm 4.

7.2.1.1 Min Value Determination

The first part (Algorithm 1:1), detailed in Algorithm 2, consists in finding, for each individual buffer, the minimal number of bits b^{best} and the associated data-type which satisfy

Algorithm 1: Global view of the DSE algorithm.

```
1 w^min ← minValueDetermination // Algorithm 2
2 q^test ← runWithTestVector(w^min)
3 if q^test < q^constraint then
4     w^min ← Iterative_Process(w^min) // Algorithm 3
5 return bitScraping(w^min) // Algorithm 4
```

the quality constraint $q^{constraint}$, while every other buffer are kept at their original data-type. The goal of this step is to find an appropriate starting vector $\boldsymbol{w}^{min}$. This step is based on a dichotomy search in order to limit the number of times the application is executed. Each bit-width n^{bit} is tested with each candidate data-type. Intermediary results are kept aside for latter parts of the DSE, as well as potential future explorations with different quality constraints, and the best bit-width b^{best} is added to $\boldsymbol{w}^{min}$. This vector of individual minimums $\boldsymbol{w}^{min}$ is then tested to check if it satisfies the quality constraint (Algorithm 1:2-4). If not, the algorithm continues with the second part (Algorithm 1:4) detailed in Algorithm 3.

Algorithm 2: Min Value Determination.

```
   Input: List of buffer to process
   Output: List of minimal sizes
1  foreach j ∈ bufGlob ∪ bufLoc do
2      q^test ← ∞
3      n^bit ← MAX_NB_BIT
4      b^best ← MAX_NB_BIT
5      for i ← log2(MAX_NB_BIT) - 1 downto 0 do
6          if q^test ≥ q^constraint then
7              n^bit ← n^bit - 2^i
8          else
9              n^bit ← n^bit + 2^i
10         q^test ← findBestType(j, n^bit)
11         if (q^test ≥ q^constraint) AND (n^bit < b^best) then
12             b^best ← n^bit
13     w_j^min ← b^best
14 return inflateBufferBitWidth(w^min)
```

7.2.1.2 Iterative Process

The main part of Algorithm 3 is an iterative convergent process. On each step, the algorithm tries to find an acceptable buffer configuration $\boldsymbol{w}^{test}$ with a better ratio $q^{Mem} = \dfrac{\Delta quality}{\Delta memory}$ (quality variation over footprint increase) until a solution satisfying the quality constraint is found.

The general idea of this part is to progressively increase the output quality of the application by selectively adding bits to data buffers, using the vector of individual minimums $\boldsymbol{w}^{min}$ as a starting point.

For this purpose, on every step, a bit-budget w^{budget} and a w^{offset} are defined to determine the number of bits to add to data buffers. Global buffers are handled as independent buffer entities, while local buffers are all considered as a single unique buffer entity. With n^{global} the number of global buffers and n^{local} the number of local buffers, the number of buffer entity to consider for the definition of w^{budget} and w^{offset} is $n^{entity} = n^{global} + \min(n^{local}, 1)$. The number of bits to add is defined such as w^{budget} corresponds to the number of buffer entity to receive a single additional bit, and w^{offset} to the number of bits added to each buffer entity. Consequently, during each step, there are multiple ways to apply w^{budget} on the vector of buffer bit-width $\boldsymbol{w}^{iter}$.

When processing a step, the algorithm sets up a list $\boldsymbol{W}^{test}$ of all the ways to apply w^{budget} and w^{offset} on $\boldsymbol{w}^{iter}$, the base vector for the current iteration. Bit-width vector of $\boldsymbol{W}^{test}$ are inflated, creating potential duplicates to remove. $\boldsymbol{W}^{test}$ is then ordered by increasing memory footprint. Finally, the application is executed with every entry of $\boldsymbol{W}^{test}$.

If no satisfying $qMem$ ratio can be found *i.e.* a positive value, the bit-budget is increased, until it reaches a value of $n^{global} + \min(n^{local}, 1)$. w^{budget} then goes back to 1 bit, and $\boldsymbol{w}^{offset}$ is incremented and applied on every buffer (Algorithm 3:11). While global buffers are all treated equally, local buffers are handled as a separate group. Indeed, local buffers with non-overlapping lifespans are completely independent from one another. Consequently, as the maximal memory footprint is achieved in a specific section of the application, local buffers that are not contributing to this maximum can have their bit-width increased up until the point it either reaches the memory ceiling, or reaches the number of bits used for the original data (usually 32 or 64 bits). An example is shown in Section 7.2.3.

To simplify the representation of the method, some of its parts have been offloaded in the form of functions, detailed hereafter, the first two being user-provided:

Algorithm 3: Iterative Process.

Input: List of buffer to process
Output: A set of parameter that satisfies $q^{constraint}$

```
1  q^iter ← q^test
2  w^iter ← w^min
3  bestQMem ← 0
4  w^budget ← 0, w^offset ← 0, exit ← False
5  while exit = False do
       // If no satisfactory step has been found, add an additional bit to the pool
6      if bestQMem ≤ 0 then
7          w^budget++
8      else
9          w^budget ← 1, w^offset ← 0
       // If an additional bit has been added to every buffer, increase the pool
          offset
10     if w^budget > n^entity then
11         w^budget ← 1, w^offset++
       // Provides a list of test vectors ordered by increasing footprint
12     W^test ← generateVectorList(w^iter, n^global, n^local, w^budget, w^offset)
13     foreach w^test in W^test do
14         q^test ← runWithTestVector(w^test)
15         qMem ← Δquality ÷ Δmemory
           // Check if satisfactory solution
16         if q^test ≥ q^constraint then
17             exit ← True, Break
           // Check if satisfactory next step
18         if qMem > bestQMem then
19             bestQMem ← qMem
20             q^iter ← q^test
21             w^iter ← w^test
           // If buffers are already at max size, then exit without solution
22         if ∑ w^test = n^entity × MAX_NB_BIT then
23             exit ← True, Break
24 return w^min
```

- **qualityEvaluation**(): Evaluate the quality of the output in comparison to a reference. The quality metric to use needs to be adapted to the application to optimize and its type of output, such as PSNR, SSIM, MSE, error rate, etc... This function is user-provided.

- **appFootprint**($\boldsymbol{w}^{test}$): Returns the application footprint corresponding to the $\boldsymbol{w}^{test}$ parameter set. This function can either be user-provided or performed through an application development framework such as PREESM.

- **findBestType**($\boldsymbol{w}^{test}{}_j$, n^{bit}): If not previously done, runs the application with a single AxB for buffer $\boldsymbol{w}^{test}{}_j$ on n^{bit}, with each available data-type. Returns the best quality and save the result (and associated type) for future uses. Uses **qualityEvaluation**.

- **inflateBufferBitWidth**($\boldsymbol{w}^{test}$): Gradually increases the size of local buffers until it reaches the memory ceiling of the application. Uses **appFootprint**.

- **runWithTestVector**($\boldsymbol{w}^{test}$): Checks the known individual best type for each values in $\boldsymbol{w}^{test}$, running **findBestType** if required. Then runs the application with the $\boldsymbol{w}^{test}$ parameter set. Returns the output quality value. Uses **findBestType** and **qualityEvaluation**.

- **generateVectorList**($\boldsymbol{w}^{min}$, n^{global}, n^{local}, w^{budget}, w^{offset}): Generates a list $\boldsymbol{W}^{test}$ of all the ways to add a bit to w^{budget} number of buffers with w^{offset}, inflating local buffer when possible, removing any duplicates, and ordering the list by memory footprint. Returns $\boldsymbol{W}^{test}$. Uses **appFootprint** and **inflateBufferBitWidth**.

The result from this step is a set of bit-width and types to use with AxBs which satisfies the quality constraint. However, the more the constraint is overshot, the higher the memory footprint is likely to be. For this purpose, our DSE features a bit scraping step.

7.2.1.3 Bit Scraping

After the main part is completed (Algorithm 3) and a result satisfying the constraint is obtained, a last pass is performed. This last step is the bit-scraping step (Algorithm 4).

It consists in trying to remove additional bits from $\boldsymbol{w}^{min}$ (obtained from Algorithm 3). The result vector obtained from Algorithm 3 is able to satisfy the quality constraint, but

Algorithm 4: Bit Scraping.

Input: Vector $\mathbf{w}^{\min}$ respecting constraint $\mathrm{q}^{\text{constraint}}$
Output: Optimised buffer, New quality value

```
exit ← False
while exit = False do
    for i = 0 to n^buffer do
        w^test ← w^min
        w_i^test ← w_i^test - 1
        if appFootprint(w^test) < appFootprint(w^min) then
            q^test ← runWithTestVector(w^test)
            if q^test ≥ q^constraint then
                w^min ← w^test
                q^last ← q^test
                exit ← True
return w^min, q^last
```

because of the way Algorithm 3 is designed, this solution may not be optimal depending on the buffer configuration of the application.

The bit-scraping method consists in iterating on every buffer of $\boldsymbol{w}^{min}$ trying to successively remove bits. If the removal allows both a reduction of the memory footprint and maintaining of the quality constraint, the change is committed, otherwise it is discarded. If no bits can be removed from buffers, the bit-scraping process is considered completed.

7.2.2 Complexity Analysis

On each step and considering that the best storage format is known for every buffer, the maximum number of tries to find a suitable result is given by the combination $\binom{n}{k} = \frac{n!}{k!(n-k)!}$, with k the bit-budget b^{budget} and n the total number of buffer ($n^{buffer} = n^{global} + n^{local}$). However, the presented generic method handles separately global buffers and local buffers.

While the effect on adding a bit to a global buffer is always measured, increasing the footprint of a local buffer may allow another local buffer to be inflated without any impact on the memory footprint. This means that instead of blindly testing the impact of all the different ways to apply the bit-budget to the buffer (hence the $\binom{n}{k}$ number of possibilities), when adding a bit to the local buffer pool, all other local buffers are inflated (as long as the overall memory footprint remains unchanged). As a result, some of the

test vectors related to the increase of local buffer size end up being identical, allowing the rejection of duplicates. The more local buffers there are, and the more they can share their allocated space, the more duplicates will be rejected.

The main drawback of our method is, as the memory footprint is modified by small increments, the number of test runs potentially required to find a suitable result. However, it has an upper bound. The first step (Algorithm 2) is guaranteed to complete in at most $n^{run} = \log_2(MAX\_NB\_BIT) \times n^{buffer} \times n^{type}$. The second step (Algorithm 3) has its maximum number of test runs per iteration bounded by $\binom{n}{k}$ (with $k = b^{budget}$ and $n = n^{global} + n^{local}$), but as previously explained, the more local buffers there are, the closer n is to the number of global buffers, such as $n = \lim_{n^{local} \to \infty} n^{buffer} = n^{global}$.

It is important to note that the **findBestType** function saves the information of which data-type is best suited for each bit-width, and this for every buffer. This means that after a result has been obtained for a desired quality constraint, the next execution of the algorithm (possibly with a different quality constraint) will potentially be substantially faster.

As the AxB concept our method relies on has a small impact on the execution time of the application, a test run of the application takes a similar amount of time as a regular run.

7.2.3 Example

An example application is shown on Figure 7.1. For the sake of simplicity, this example uses a SDF-like representation [LM87]. In this representation, the application is represented as a dataflow graph. The actual computations take place in graph vertices, called actors, and data is moved from one actor to the next through the graph edges, which are memory buffers.

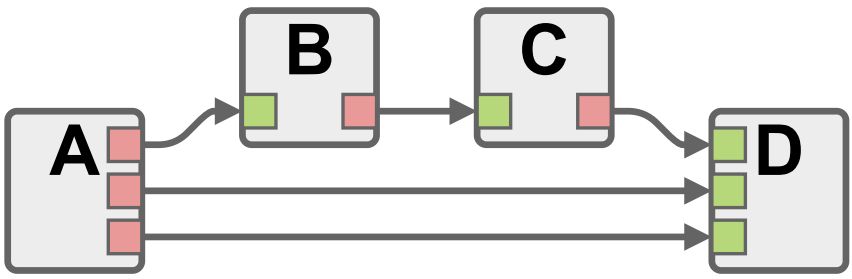


Figure 7.1: Simple application example with 4 actors and 5 buffers.

This application example is composed of 4 actors (here called **A**, **B**, **C** and **D**), 2 global buffers ($\mathbf{AD_1}$ and $\mathbf{AD_2}$) and 3 local buffers (**AB**, **BC**, **CD**). A buffer only needs to have its dedicated memory allocated from when its data is produced, to when its data is consumed. When an actor is about to be executed, both its input and output buffers need to be allocated. Figure 7.2 shows which buffers are allocated at each step of the application execution. **AB** and **CD** share the same memory space.

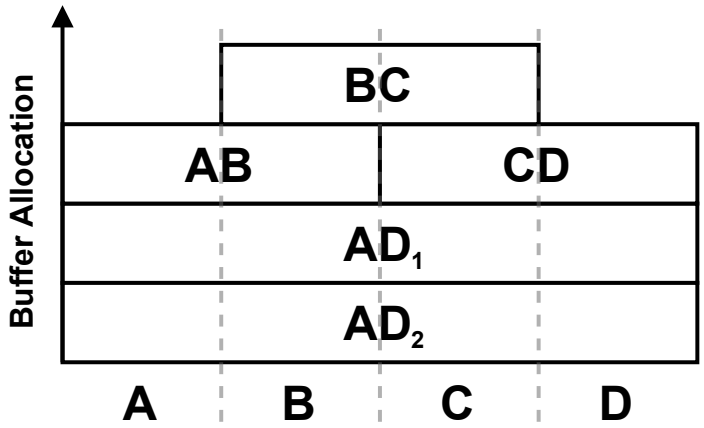


Figure 7.2: Graph of buffer allocation requirements dependent of actors execution.

In this example, the largest memory footprint is achieved when executing actors **B** and **C**. The maximum memory footprint is calculated as $mem = mem_{\mathbf{AD_1}} + mem_{\mathbf{AD_2}} + \max(mem_{\mathbf{AB}} + mem_{\mathbf{BC}}, mem_{\mathbf{BC}} + mem_{\mathbf{CD}})$.

Table 7.1 shows the different possible allocations of w^{budget} on the example application. Because there are 2 global buffers and *some* local buffers, the highest possible value of w^{budget} is 3. The addition of another bit to the budget leads to the increase of w^{offset} by 1, and bringing w^{budget} back to 1.

		$w^{offset}=0$							$w^{offset}=1$						
Bit budget		$w^{budget}=1$			$w^{budget}=2$			$w^{budget}=3$	$w^{budget}=1$			$w^{budget}=2$			$w^{budget}=3$
Global buffers	$\mathbf{AD_1}$	1	0	0	1	1	0	1	2	1	1	2	2	1	2
	$\mathbf{AD_2}$	0	1	0	1	0	1	1	1	2	1	2	1	2	2
Local buffers		0	0	1	0	1	1	1	1	1	2	1	2	2	2

Table 7.1: Bit budget allocation for the example application in Figure 7.1.

Regarding the number of runs required to complete the algorithm (considering 32-bit wide data and 3 storage representation formats): the first step (Algorithm 2) is done after

at most $\log_2(32) \times 5 \times 3 = 75$ while, as buffers **AB** and **CD** share the same allocation space, each iteration of the second step (Algorithm 3) is done in either $\binom{4}{1} = 4$ or $\binom{4}{2} = 6$ plus the occasional single-buffer best type evaluation performed by the **findBestType** function.

7.3 Experimental Results

The proposed DSE method was tested on the application presented in Chapter 4 to demonstrate its genericity and to illustrate the possibilities of memory footprint reduction.

7.3.1 2D Wavelet Filter

The DSE method is tested with a quality constraint from 20 to 100dB with 5dB increment.

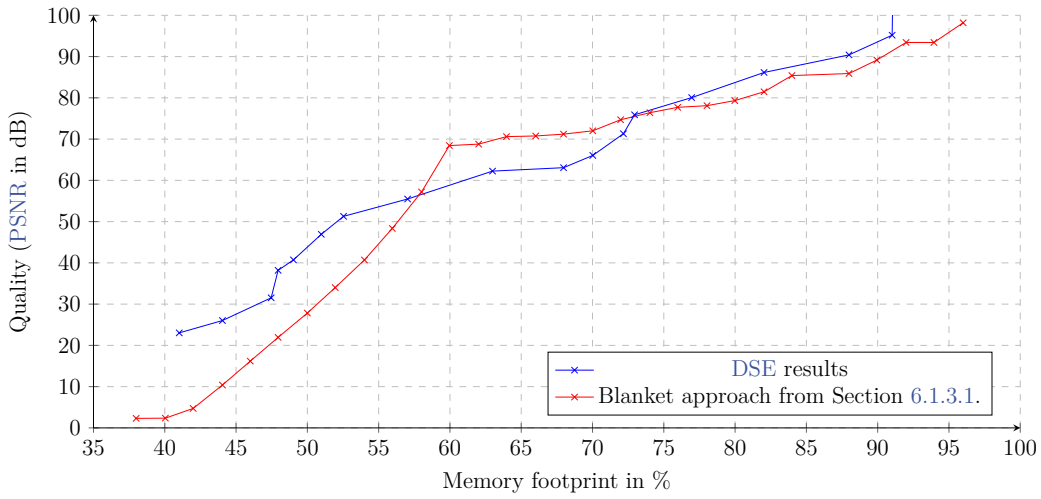


Figure 7.3: Graph of the memory footprint dependant of the output quality.

Figure 7.3 shows the results of the DSE method on the 2D Wavelet Filter presented in Section 4.1, along with the results from the simpler approach from Section 6.1.3.1. The DSE is able to find a solution satisfying the quality constraint while achieving a lower memory footprint for constraints below 55dB or higher than 75dB. In the interval between 55 and 75dB, the blanket approach from Section 6.1.3.1 is better. This inability to get good results on this interval comes from the *Bit Scraping* step Algorithm 4. In this specific application, multiple buffers are responsible at the same time for the highest memory footprint, making the *Bit Scraping* unable to reduce the bit-width of concerned

buffers, as such reduction has no impact on the overall memory footprint. A solution to this issue would be to give the *Bit Scraping* step the ability to know which set of buffer are responsible for this maximal memory footprint, reducing the bit-width of the whole set instead of one buffer at a time.

Despite this weakness, the proposed DSE is still able to reduce the footprint from 58% to 52.5% with a PSNR above 50dB on this application.

7.3.2 SqueezeNet CNN

SqueezeNet [Ian+16] is a Deep Neural Network (DNN) for computer vision using a CNN architecture, presented in Section 4.2. The memory footprint of the reference implementation is composed of around 4.7MB of model parameters, around 3.8MB of buffers to store data between layers and around 0.7MB of other memory allocations, for a total of 9.2MB.

The buffers used to test the DSE method are the 10 global buffers holding the model of the CNN, and the local buffers used between layers, from the input of *conv1* to the output of *fire5* (minus the output of *maxPool3*), for a total of 17 buffers.

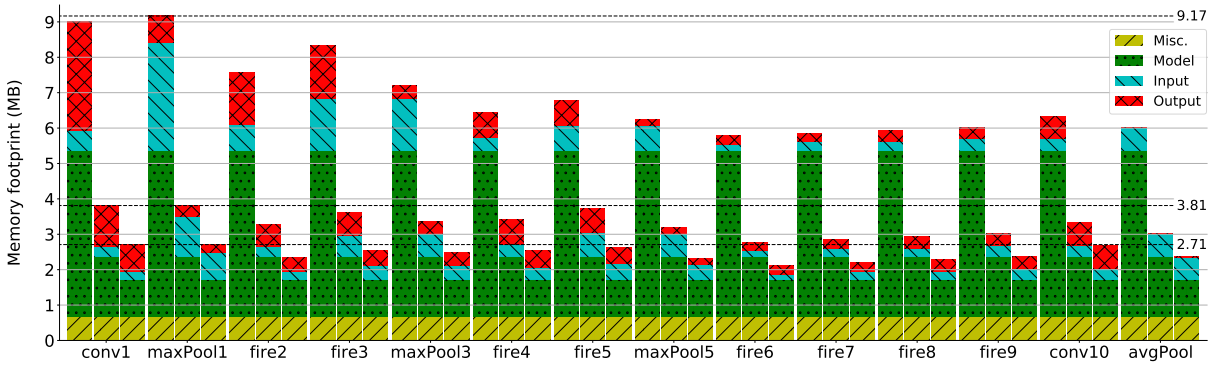


Figure 7.4: Memory allocation at different stage of the SqueezeNet CNN with the reference implementation (left columns) and with our method with a **100%** accuracy (middle columns) and a **90%** accuracy (right columns).

Figure 7.4 shows the memory allocation for the SqueezeNet CNN without AxBs (left columns), with AxBs for a 100% identical TOP-1 accuracy (middle columns), and for a 90% identical TOP-1 accuracy (right columns).

Using the proposed DSE algorithm, the memory footprint of the application can be reduced to 3.81MB, a 58.6% reduction, while maintaining the TOP-1 classification accuracy. This result is obtained by exploring only 300 of the theoretical maximum $32^{17} \approx 3.87 \times 10^{25}$

possible configurations. The memory footprint obtained with our method are displayed on Figure 7.4 (right columns).

Furthermore, as shown on fig. 7.4, lowering the classification accuracy constraint from the original implementation down to 90% allows a memory footprint reduction to 2.73MB, a 70.3% reduction. This result is obtained by exploring 724 configurations. Inference time when using AxBs is around 7% slower. The time for our DSE method to find a solution is overwhelmingly dominated by the processing time of the application. Considering that a single execution of the application with the 300 images dataset takes around a minute on **CPU**, our DSE method is able to provide a satisfying AxB configuration in approximately 5 hours, depending on the quality constraint.

7.3.3 SDP Imaging Pipeline

The SDP Evolutionary Pipeline (SEP) Imaging Pipeline[1] is an implementation of the Square Kilometre Array (SKA) SDP for its most compute intensive task, presented in Section 4.3.

The normal memory footprint is 234.9MB. A quick analysis of the application shows that it already uses memory footprint reduction techniques, such as in-place computation, which has to be taken into account to correctly evaluate the memory footprint reduction.

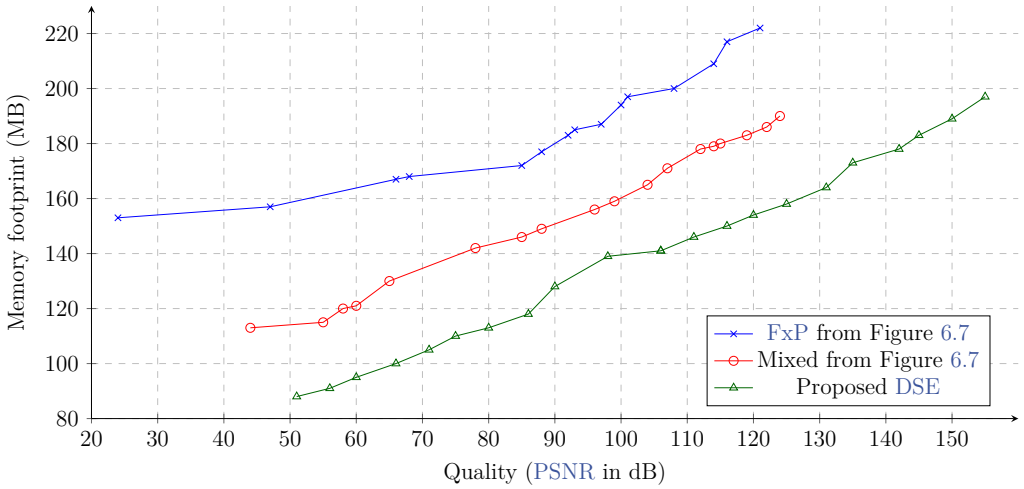


Figure 7.5: Graph of the memory footprint dependant of the output quality.

1. SEP Pipeline Imaging - GitLab: https://gitlab.com/ska-telescope/sdp/sep_pipeline_imaging

Figure 7.5 shows the evolution of the memory footprint of the application dependant of the output quality, tested with quality constraints from 50dB to 155dB by 5dB increments, obtained with the presented DSE method.

On this application, using the proposed DSE method with a 105dB constraint yields an output quality of 106dB with a memory footprint of 141MB, a 39.7% reduction. This result is obtained by exploring only 7 of the maximum $32^9 \approx 3.52 \times 10^{13}$ possible configurations, with a runtime of around 170 seconds per configuration. The use of AxBs on this application leads less than 1% longer execution times. Depending on the end-user requirements, this level of accuracy may be higher than needed, allowing the memory footprint to be reduced even further.

The proposed DSE method is able to produce a parameter set to properly configure AxBs on this application in a matter of minutes to hours rather than weeks and providing at the same time better quality results than the hand-made DSE.

In both this use-case and the previous one, there is no data-type that is exclusively more accurate regardless the bit-width. Instead, some data-types tend to be better on certain range of bit-width, hence the need to test multiple different configuration. Furthermore, the DSE algorithm is able to provide the AxB configuration for a given quality constraint in a matter of minutes to hours.

7.3.4 2D-DWT on FPGA

In the case of an FPGA application, as every actor is executed concurrently, the hardware resources required for implementing FIFOs are always in use, making every buffer a global buffer. This creates a situation where the DSE method presented in this chapter is not used in its intended setting. While it is suboptimal, the DSE method is still able to provide a set of token-width satisfying the quality constraint.

Figure 7.6 shows the amount of BRAM used for FIFO in order to satisfy a given quality constraint. The DSE method was set up to minimize the amount of BRAM without taking into account the data-packing technique presented in Section 6.2. The resulting bit-width are then inputted into PREESM to perform data-packing, and BRAM usage is confirmed with Vitis HLS.

Aside from the bit-widths configuration obtained from the DSE method with a 56.3dB quality, the proposed DSE provides results at worst identical to Section 6.2.1.

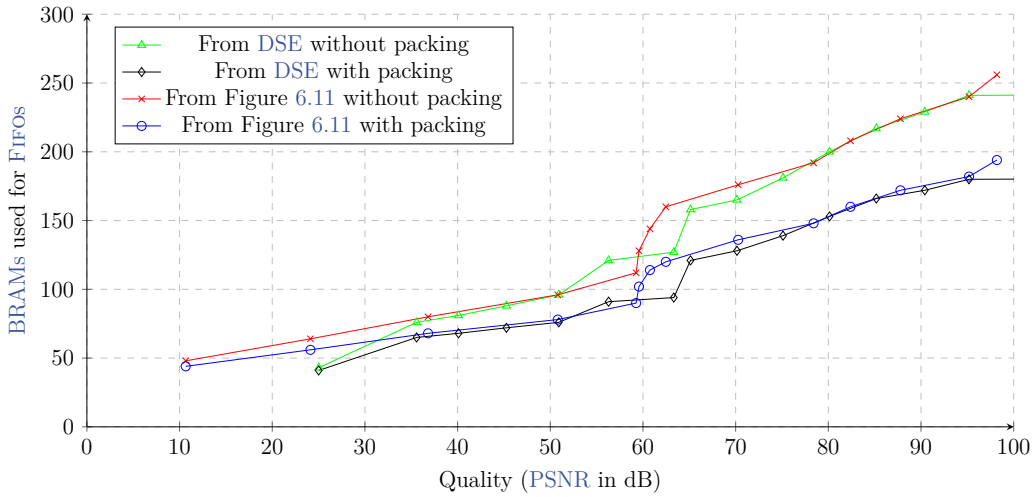


Figure 7.6: Graph of the quantity of BRAMs used with the 2D-DWT dependent on the output quality.

Conclusion

This chapter presents an automatic and generic DSE method to optimize the bit-width and format of AxBs, a memory-oriented AxC technique. It only requires to be provided with initial memory information of the application to optimize as well as the quality evaluation method, and uses this information to automate the AxC-oriented DSE.

We have shown that the proposed DSE method is capable of finding the required buffer configuration to reduce the memory footprint of data processing application. The genericity and efficiency of the proposed DSE method has been shown on the application set from Chapter 4, a generic image processing application, an off the shelf CNN, and a specific signal processing application. It shows the possibility to substantially reduce the memory footprint of applications with little to no significant impact on the output quality.

An interest of our work is to simplify the implementation of memory oriented AxC techniques, enabling applications to be deployed on previously unfit architectures, which usually requires a deep-dive into the inner-working of the targeted application. The precision optimization of actual computations, actors internal behaviour in case of a dataflow application, is not in the scope of the proposed method. Existing approaches focusing on the processing part optimization are complementary and can be combined with our approach.

Additionally, the proposed DSE method has been tested on an SDF-based FPGA application to reduce the hardware resources usage, providing optimal bit-widths for data-tokens in FIFOs. The Data-Packing technique presented in Section 6.2 can then be applied to further reduce the requirements of hardware resources.

CHAPTER 8

Conclusion

8.1 Summary

The memory requirements of digital signal processing and multimedia applications have grown steadily over the last several decades. From embedded systems to supercomputers, the design of computing platforms involves a balance between processing elements and memory capabilities to avoid the memory wall.

The contributions presented in this thesis are mainly aimed at reducing the memory footprint of data processing applications with the AxC paradigm. These contributions were developed with the objective of targeting data processing applications described with an SDF-based MoC, but are not limited to these MoCs and can be applied on any kind of application with a clear segmentation between data and computations. While the contributions were initially aimed toward CPU-based architectures, the concepts presented can be adapted to reduce the hardware resources on FPGAs. Most of the contributions of this thesis have been implemented within the PREESM application development framework.

In Chapter 5, a study is conducted to evaluate the accuracy loss associated with the reduction of the number of bits used to represent data. The impacts of the use of multiple alternative data representations are assessed as a way to mitigate the accuracy loss from bit-width reduction, as well as their limitations. Experimental results showed

opportunities to reduce the bit-width of data in storage up to 50% without compromising the overall accuracy.

In Chapter 6, a reference implementation is provided to enable data-tokens of Fifo buffers of unusual bit-width to be stored in memory in a concatenated manner. This enables a reduction of the memory footprint associated with these Fifos. The concept is also demonstrated in the context of FPGA applications, enabling a reduction of specific hardware resources. Experimental results validated to memory footprint reductions, with an emphasis on the possibility to trade off the accuracy for reduction in hardware resources requirements.

In Chapter 7, a generic DSE method is proposed to find the appropriate bit-width and data representation minimizing the memory footprint of an application while satisfying a given level of accuracy. The proposed DSE method leverages the knowledge of buffers allocation lifespan in memory to avoid reducing the bit-width when unnecessary. Experimental results showed that the proposed DSE is able to find solutions to efficiently reduce hardware requirements of data processing applications.

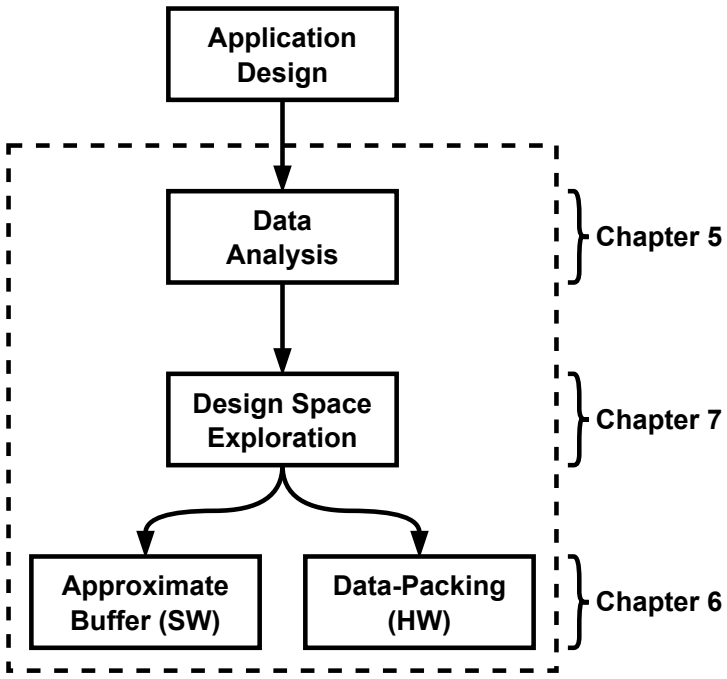


Figure 8.1: Summary of the contributions of this thesis.

Figure 8.1 shows a workflow revolving around the contributions of this thesis. The process starts with the initial reference data processing application designed by the de-

veloper, ideally with an SDF-based MoC. The nature of the data store within FIFOs is evaluated to determine the format to use for alternative data representation with any bit-width (Chapter 5). Then, a DSE is performed to find the appropriate data representation and bit-width to minimize hardware resources usage while maintaining an accuracy constraint (Chapter 7). Finally, the application with reduced hardware resources usage can be executed on its intended architecture, either CPU-based or FPGA-based.

8.2 Future Works

The work of this thesis opens opportunities for future research on the impact of the presented contributions on other aspects of computing architecture, as well as the eventual interaction of complementary approaches.

8.2.1 Impact on Other Parameters

This thesis does not evaluate the impact of the presented contributions on other aspects of a computing architecture. On a CPU-based architecture equipped with caches, concatenating data of reduced bit-width in memory implies a similar behaviour in the different level of caches. A single cache-line will logically hold more data. An interesting future work would be to study the impact of the techniques presented in this thesis on the behaviour of the caches.

Additionally, the impact on energy consumption has not been evaluated. As interactions with external memory consumes orders of magnitude more energy than cache accesses [Hor14], the use of the presented techniques could lead to a reduction of the energy consumption.

8.2.2 Extension with Additional Features

The techniques presented in this thesis could be extended with additional data representation such as Logarithmic Number System (LNS) [DD07] or posits [GY17]. Adding support for alternative data representation specifically suited for a given application may enable additional gains in terms of resources reduction. The overhead from the presented techniques could be potentially reduced or nullified by the use of in-memory computations [Mut+19].

Additionally, while this thesis only considers conventional CPUs and FPGAs, applying the presented concepts on other kind of hardware architectures such as Graphics Processing Units (GPUs) and Application-Specific Integrated Circuits (ASICs) may provide interesting results. One of the main challenges regarding efficient GPU utilization consist in feeding enough data to Processing Elements (PEs). The concepts presented in this thesis could potentially be derived to alleviate this obstacle.

8.2.3 Interactions with Complementary Approaches

Finally, a research opportunity succeeding this thesis would be to perform modification to the computation themselves, such as converting to FxP arithmetic, using the contributions of this thesis as guidelines. With an FPGA architecture running an HLS application, it is already possible to bind the data format used for computation to the format used for storage. This concept could be exported the CPU-based architecture with the use a source-to-source compiler to perform similar operations. The challenge then concerns the repartition of the introduced error between computation and memory modification.

Combining all these research opportunities together could potentially enable unprecedented results but represent a complex multi-criteria optimization problem.

APPENDIX A

French Summary

A.1 Introduction

Depuis les dernières décennies, le volume des données traitées a crû en adéquation avec l'augmentation de la puissance de calcul des systèmes de Calcul Intensif (HPC), ainsi qu'avec la généralisation des systèmes embarqués sous toutes leurs formes, allant de l'Internet des Objets (IoT) aux ordiphones.

Les systèmes embarqués sont les systèmes informatiques conçus spécifiquement pour une application donnée, des équipements indépendants telle qu'une tondeuse à gazon autonome, un appareil photo numérique ou des appareils électroménagers, ou faisant partie d'un assemblage tel que les multiples sous-systèmes d'un véhicule.

La croissance des ressources de calculs va de pair avec l'augmentation et la complexification des systèmes liés à la mémoire. Tant pour les ordinateurs personnels que pour les ordiphones, la quantité de RAM double tout les 2 à 3 ans.[1] Dans un système informatique, la mémoire est utilisée pour stocker les données, pour stocker les instructions programme, pour sauvegarder des valeurs temporaires, et pour les mécanismes de synchronisation.

Un processeur infiniment rapide ne peut fonctionner qu'aussi vite que la donnée lui est transmise. Ce problème nécessitant d'éviter le mur mémoire [WM95 ; JWN10 ; Ziv+17]

1. Exemple avec les ordiphones : iPhone 2G (2007) : 128Mo, iPhone 14 Pro (2022) : 6Go, Plus haut de gamme actuel : 18Go, mais c'est stupide.

est connue depuis environ 3 décennies, amenant à la conception de mécanismes spécifiques pour la mémoire. En conséquence, il est évalué que jusqu'à 80% de la surface de silicium est dédiée aux caches, aux mémoires de stockage, aux contrôleurs mémoires et aux interconnections dans le seul but de transmettre la donnée à l'intérieur d'une puce [DCD97]. De plus, les transferts de données entre la mémoire et les éléments de calculs (PEs) peuvent représenter jusqu'à 62% de l'énergie consommée par un système de calcul.

Par exemple, dans une unité centrale de traitement (CPU) Advanced Micro Devices (AMD) de l'architecture Zen 3, le cache L3 de 32Mo occupe à lui seul 52% de la surface de silicium d'un CCD, et 64Mo additionnel peuvent être empilés par-dessus [Bur+22], pour un total de 12Mo de cache L3 par cœur.

A.1.1 Portée de cette Thèse et Contributions

Les limitations liées à la mémoire sont une considération majeure durant la conception et le déploiement d'application de traitement de données sur un système informatique, aussi bien sur les aspects de capacité, d'énergie, de transmission, et de surface.

Le paradigme de l'Informatique à-Peu-Près (AxC) [HO13] a émergé comme une possibilité pour améliorer l'efficience énergétique et/ou la performance des systèmes informatiques, sacrifiant la précision des résultats d'une application où l'intégrité des données n'est pas critique, comme l'abandon d'une image durant l'opération d'encodage ou de décodage d'un flux vidéo. Les méthodes d'AxC peuvent être séparées en 3 catégories, en fonctions de si elles impactent les données utilisées durant les calculs, la nature des calculs, ou les paramètres du matériel. Ces techniques ont tendance à être difficiles à mettre en œuvre mais sont susceptibles de réduire les contraintes subies par le système de mémoire d'une plateforme de calcul.

Une autre manière de réduire les besoins liés à la mémoire est l'utilisation de modèles de calcul MoC spécifiques. Décrire une application de traitement de données avec un MoC distinct peut permettre la mise en valeur de certaines opportunités d'optimisations mémoire. Les MoCs basés graphes flot-de-donnée ont la particularité de mettre au premier plan la gestion de la mémoire, à la fin en terme d'allocations et que de transferts, mettant en avant les opportunités d'optimisations.

Dans cette thèse, de nouvelles techniques sont développées pour permettre la réduction des contraintes d'empreintes mémoires d'applications de traitement de données en utilisant le paradigme d'AxC avec les représentations de graphe flot de données.

Les contributions principales de cette thèse sont :

1. Une étude de l'impact des largeurs et représentations des données en mémoire sur la qualité du résultat d'une application. Le stockage de donnée est simulé avec un nombre de bit arbitraire, couplé avec une représentation de donnée personnalisable pour minimiser la perte de qualité. Cette contribution a été en partie publiée dans [Mio+20].

2. Une méthode de conversion et de stockage de données à largeur de bit arbitraire en mémoire pour CPU. Cette méthode gère les opérations mémoire pour stocker des données non-alignées dans des segments concaténés. Cette contribution a été en partie publiée dans [Mio+20].

3. Une méthode pour efficacement compacter et extraire des données de paquets pour optimiser les ressources de stockage mémoire pour Réseaux de Portes Programmables *In Situ* (FPGA). Cette méthode est capable, à partir d'une représentation flot de donnée, juste après le processus de détermination de taille des mémoires tampons, de trouver le taux d'empaquetement optimal pour réduire les besoins de ressource matériel. Cette contribution n'a pas encore été publié.

La plupart des contributions de cette thèse ont été intégrée dans l'environnement de développement d'application PREESM.

A.1.2 Outline

La Section A.2 présente l'état de l'art sur les méthodes d'AxC et le MoC flot de donnée. La Section A.3 présente le concept général et l'impact sur la qualité des contributions basé AxC de cette thèse. La Section A.4 fournit une ligne directrice sur la manière de mettre en œuvre ces techniques. Enfin, la Section A.5 conclue ces travaux et propose des perspectives de recherches futures.

A.2 État de l'Art

A.2.1 Calcul à-Peu-Près AxC

L'un des objectifs de cette thèse est de concevoir des techniques basées sur le paradigme de l'AxC. Une technique d'AxC est une technique qui quand appliquée sur un système (logiciel ou matériel), entraîne la production de résultats imprécis mais utilisables. La

compensation pour ces imprécisions est une réduction des ressources nécessaires (puissance de calcul, empreinte mémoire, consommation d'énergie, surface de silicium, ...) ou une augmentation des performances (débit plus élevé, latence plus basse, ...) en fonction des besoins et du type de technique utilisée [BMS22].

Ces techniques d'AxC peuvent être séparées en 3 groupes en fonction de la manière dont elles affectent le systèmes. Les techniques de *Calculs à-Peu-Près* modifient la nature des calculs effectués sur les données dans le but de réduire la complexité. Les 2 approches principales du *Calcul à-Peu-Près* sont le *saut de calcul* [Sid+11 ; Vas+15 ; Meh+09 ; Zha+14] et l'*approximation de calcul* [Sur+15]. Le *saut de calcul* consiste à ne pas exécuter certaines parties d'une application pour réduire la complexité, par exemple en terminant prématurément un processus itératif telle qu'une descente de gradient. L'*approximation de calcul* consiste à remplacer un traitement précis et complexe par un traitement moins précis mais plus simple, comme la substitution d'une fonction mathématique complexe par un groupe de fonctions polynomiales simples.

Les techniques de *Matériel à-Peu-Près* s'appuient sur des comportements ou des modifications spécifiques du matériel. Les 3 approches principales du *Matériel à-Peu-Près* sont les *Unités de Calcul à-Peu-Près* [Gup+11 ; Mah+10 ; KGE11], la *Modification des Paramètres du Circuit* et le *Stockage Non-Fiable* [Smo+13 ; Fru+15 ; Rah+14 ; Mig+15]. Les *Unités de Calcul à-Peu-Près* sont conçues pour être plus rapide et/ou moins énergivore que leurs équivalents précis au prix un résultat légèrement imprécis. Les *Modifications des Paramètres du Circuit* consistent à modifier les paramètres tels que la tension d'alimentation ou la fréquence d'horloge d'un circuit de calcul. Le *Stockage Non-Fiable* exploite le comportement des cellules mémoires défaillantes ou fonctionnant avec des paramètres limites pour réutiliser des ressources autrement perdues ou réduire la consommation d'énergie.

Les techniques de *Données à-Peu-Près* se concentrent sur les données à traiter en elles-mêmes. Les 3 approches principales des *Données à-Peu-Près* sont la *Décimation de Données* [ACN14], l'utilisation de *Données Périmées* [Ren+12 ; RG01 ; KAK18] et l'*Optimisation de Précision* [BSM17 ; Tin+19 ; Hat+17]. La *Décimation de Données* consiste à réduire la quantité de données traitées, soit en volume, soit en fréquence. L'utilisation de *Données Périmées* consiste à utiliser des données potentiellement obsolètes pour réduire le surcoût en temps et en énergie lié aux synchronisations. L'*Optimisation de Précision* repose sur l'utilisation de représentation de données pour réduire la mémoire utilisée par les données.

Cette thèse se concentre spécifiquement sur les techniques de *Données à-Peu-Près*, plus particulièrement l'*Optimisation de Précision*. Des explications plus détaillées sont disponibles dans [Bon19 ; XMK15 ; Mit16].

A.2.2 Modèle de Calculs Flot de Donnée

Les modèles flot de données sont couramment utilisés pour représenter des applications de traitement de données sous une forme abstraite, mettant potentiellement en valeur des opportunités d'optimisations spécifiques. Ces représentations graphiques peuvent également être utilisées parce qu'elles sont adaptées pour ce type d'application ou pour l'architecture matérielle ciblée.

Une application décrite avec le MoC de flot de donnée synchrone (SDF) est composé d'un set de tâches concurrentes, appelé acteurs, interconnectées par des files *premier entré, premier sorti* (Fifo) dirigées et non limitées. Une Fifo connectant deux tâches crée une dépendance de donnée. L'écriture d'un jeton de donnée dans une Fifo n'est pas bloquante, mais la lecture d'un jeton de donnée depuis une Fifo, elle, est bloquante. Un jeton de donnée est indivisible, et n'est produit et consommé qu'une seule fois. Le modèle SDF est indépendant de l'architecture ; il est capable d'exploiter le parallélisme, mais ne le requiert pas. De plus, le modèle SDF est déterministe.

Le modèle SDF tire son intérêt de son analysabilité, sa prédictibilité, et de sa capacité à exposer les opportunités de parallélisme, le rendant adapté pour une exécution efficace sur une architecture matérielle. Les caractéristiques statique et déterministe du modèle SDF permet la mise en place d'optimisation spécifique, adapté pour cibler des systèmes embarqués.

La sémantique d'un MoC flot de donnée ne décrit que les interactions entre les acteurs, mais ne spécifie pas le comportement interne de ces acteurs. La nature des opérations réalisées par les acteurs doit être spécifiée par un langage hôte, tel que le `C` ou le Java, ou avec un langage de description matériel comme le VHSIC Hardware Description Language (VHDL), ou avec un langage permettant de d'écrire à la fois le graphe et le comportement interne des acteurs, tel que le CAL Actor Language (CAL) [EJ03 ; Bha+11].

A.3 Concept de Mémoire Tampon à-Peu-Près

Les applications de traitement de données sont généralement conçues pour utiliser des données avec la norme IEEE-754 de représentation de nombres à virgule flottante simple précision (FP32) ou double précision (FP64). Les applications de traitements intensifs nécessitent en conséquence des caractéristiques mémoires en correspondance avec les capacités de calculs. Cependant, le niveau de précision fourni par l'utilisation des formats FP32 et FP64 est susceptible d'être disproportionné par rapport à l'acquisition de donnée initiale, ou simplement au-delà des besoins de l'utilisateur final.

Cette section présente de concept de Mémoire Tampon à-Peu-Près AxB. Elle se concentre sur la manière dont la réduction de largeur de donnée impact négativement la précision, et comment cette perte de précision peut être atténuée par l'utilisation de représentation de donnée alternative.

L'idée de base des AxBs est de stocker la donnée en mémoire en utilisant moins de bits que la représentation originale, et sans tenir compte de son utilisation durant les calculs.

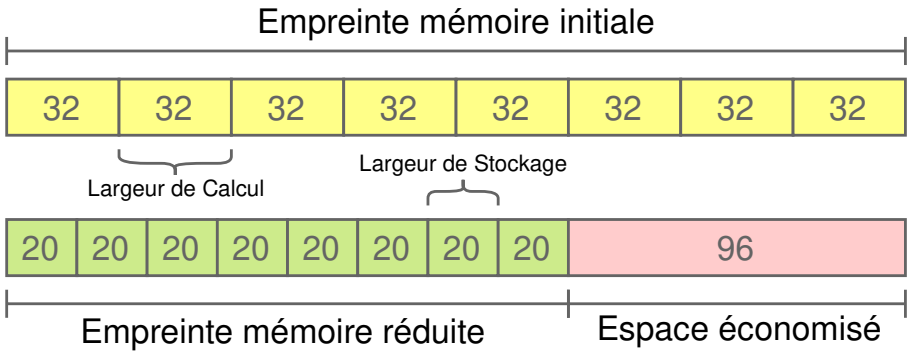


FIGURE A.1 : Illustration simplifié d'un AxB, avec des données converties de 32 vers 20 bits concaténées en mémoire.

Figure A.1 montre une représentation simplifiée du concept d'AxB, avec un jeu de données d'une largeur initiales de 32 bits converties vers 20 bits et concaténées en mémoire.

Les AxBs sont spécifiquement conçus pour cibler les mémoires tampon servant à transmettre de la donnée d'un bloc de calcul à un autre, et sont en particulier adaptés pour être utilisés sur les FIFOs d'un graphe flot de donnée.

Par défaut, la réduction de la largeur en bit d'une donnée FP32 par troncature des Bits de Poids Faible (LSB) a un effet progressif sur la précision.

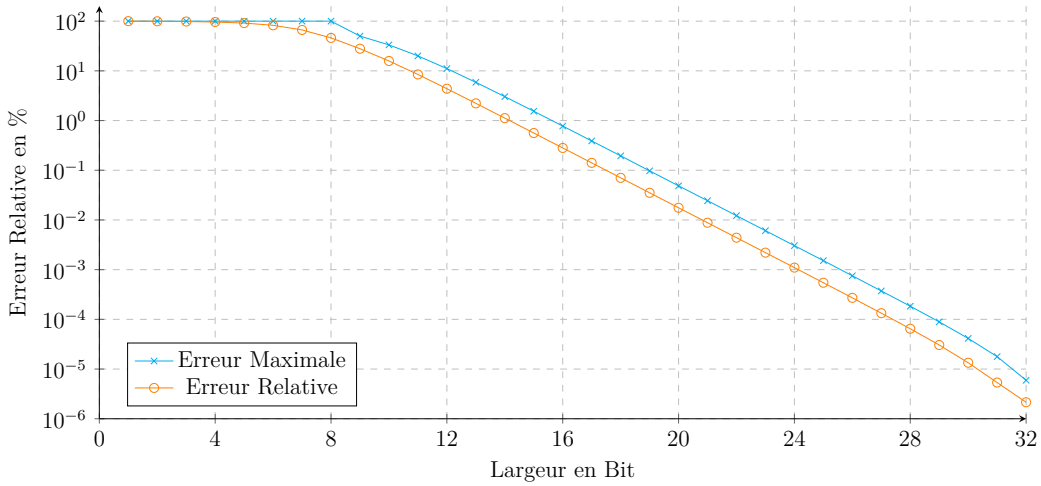


FIGURE A.2 : Erreurs relatives moyennes et maximales introduites par troncature comparées à des nombres réels.

La Figure A.2 montre les erreurs relatives moyennes et maximales introduites par troncature de données en FP32 en fonction du nombre de LSBs enlevés. L'erreur relative initiale d'un FP32 d'environ $5 \times 10^{-6}\%$ monte à 1% pour une taille de 16 bits et à 10% pour une taille de 11 bits.

A.4 Implémentation de Mémoire Tampon à-Peu-Près

Cette section fournit un exemple pour la mise en place du concept des AxBs pour réduire les besoins en ressources mémoires d'applications de traitement de données sur une architecture CPU.

Les AxBs stockent les données avec des largeurs de bits arbitraires, loin des schémas d'accès conventionnels des CPU basés sur des octets.

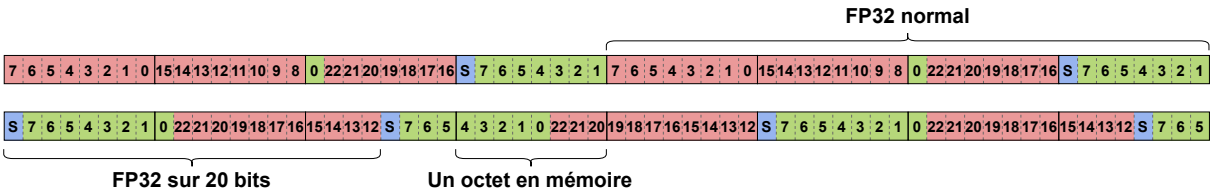


FIGURE A.3 : Représentation en mémoire de FP32 tronqués sur 20 bits et concaténés, avec le stockage conventionnel de FP32.

La Figure A.3 montre un exemple de la manière dont des FP32 tronqués sur 20 bits peut être concaténés en mémoire. La ligne du dessus montre 2 FP32 stockés sur 8 octets, et la ligne du dessous montre 3 données de 20 bits et les 4 premiers bits d'une quatrième, sur le même segment de 8 octets. Comme il n'est pas possible de faire des opérations mémoires d'une largeur de 20 bits avec une architecture CPU conventionnelle, des procédures spécifiques pour l'insertion et l'extraction de donnée dans un AxB doivent être mise en place.

Procédure d'Insertion

Une opération d'écriture d'une donnée dans un AxB n'est pas directe, car les accès mémoires ne peuvent pas se faire avec une granularité au bit, mais sont contraints à des largeurs de 2^N octets.

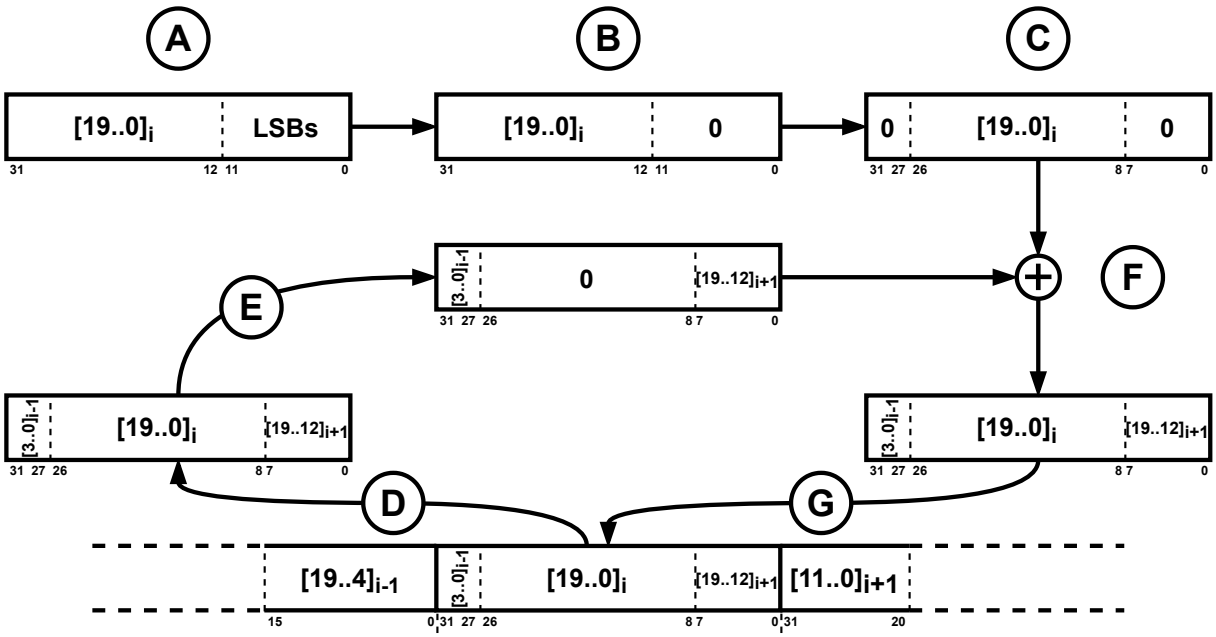


FIGURE A.4 : Procédure d'insertion d'une donnée dans un AxB étape par étape, pour une largeur de 20 bits.

La Figure A.4 montre le processus d'insertion d'un jeton de donnée dans un AxB. En pratique, les accès sont malgré tout fait avec des alignements standards. Dans cet exemple avec une largeur de bit de 20, les accès sont faits avec une largeur de 4 octets et un alignement de 2 octets. Comme les jetons de donnée sont concaténés dans un AxB, un bloc de mémoire (*i.e.* 4 octets consécutifs) contient des bits appartenant à plusieurs

jetons. Il est donc essentiel lors d'une insertion de ne pas modifier les bits appartenant aux jetons de donnée adjacents.

Pour s'en assurer, le processus d'insertion est décomposé en 7 étapes :

A) Le jeton de donnée i est fourni à la fonction d'insertion.

B) Les LSBs sont retirés par un masque de bits pour ne garder que les 20 bits de charge utile.

C) Les 20 bits de charge utile sont décalés vers leurs position final dans le bloc mémoire.

D) Le bloc est lu dans la mémoire.

E) Les bits à remplacer par la charge utile dans le bloc sont mis à 0 avec un masque de bits.

F) La charge utile est insérer dans le bloc.

G) Le bloc est réécris dans la mémoire.

Procédure d'Extraction

Le processus de lecture d'un jeton de donnée depuis un AxB est plus simple.

La Figure A.5 montre le processus d'extraction d'un jeton de donnée d'un AxB. Les commentaires concernant la largeur et l'alignement des accès sont toujours valables.

Le processus d'extraction est décomposé en 3 étapes :

A) Le bloc mémoire de 4 octets contenant la charge utile de 20 bits est lu.

B) Les bits appartenant aux jeton de donnée adjacents sont retirés avec un masque de bits pour ne garder que la charge utile de 20 bits.

C) La charge utile est décalée à son emplacement final.

L'implémentation de ces procédures d'utilisation des AxBs permet un réduction de l'empreinte mémoire d'application de traitement de donnée.

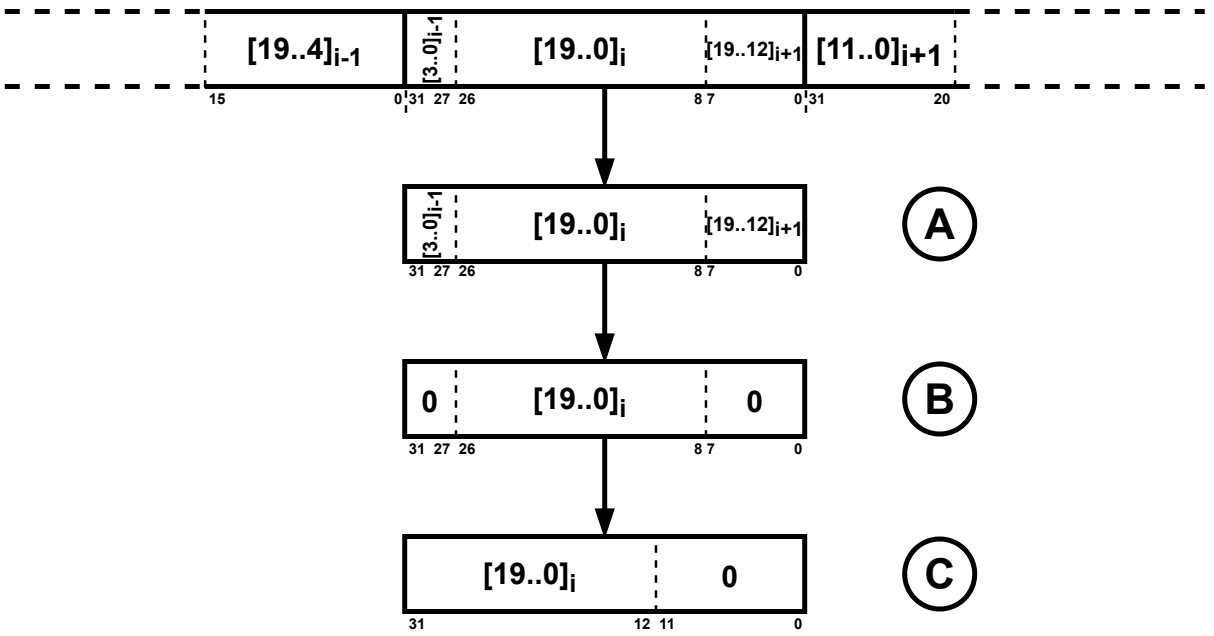


FIGURE A.5 : Procédure d'extraction d'une donnée dans un AxB étape par étape, pour une largeur de 20 bits.

A.5 Conclusion

Les besoins en mémoire des applications de traitement du signal numérique et multimédia ont constamment augmenté depuis les dernières décennies. Des systèmes embarqués aux supercalculateurs, la conception des plateformes de calcul nécessite un équilibre entre la puissance de calcul et les capacités mémoires pour éviter le mur mémoire.

Les contributions présentées dans cette thèse ont pour but de réduire l'empreinte mémoire d'application de traitement de donnée avec le paradigme d'AxC. Ces contributions ont été développées dans le but de cibler les applications de traitement de donnée décrite avec un modèle flot de donnée SDF, mais ne s'y limite pas et peut être appliquées à n'importe qu'elle type d'application de traitement de donnée distinguant clairement la donnée et les calculs. Bien que les contributions soient initialement ciblées vers les architectures CPU, les concepts présentés sont également applicables à la réduction de ressources matérielles sur FPGA. La plupart des contributions de cette thèse ont été implémentées dans l'environnement de développement d'application PREESM.

List of Figures

List of Tables

Listings

Acronyms

ALU Arithmetic Logic Unit. 27

AMD Advanced Micro Devices. 10, 126

API Application Programming Interface. 37–44, 52

ASIC Application-Specific Integrated Circuit. 32, 42, 124

ASLE Approximate Speculative Lock Elision. 23

AUT Auckland University of Technology. 60

AxB Approximate Buffer. 65–67, 69–74, 76–78, 80–82, 84–99, 102, 103, 105–107, 111, 113, 116–119, 130–134, 136–138

AxC Approximate Computing. 12, 13, 17–23, 27, 36, 53, 57, 60, 66, 67, 85, 90, 101, 105, 106, 119, 121, 126–128, 134, 148

BF16 Brain Floating-Point. 32–36, 69

BFP Block Floating-Point. 34

BRAM Block RAM. 97–99, 101–103, 118, 119, 137

CAL CAL Actor Language. 49, 129

CERN European Organization for Nuclear Research. 10, 11

cFP Custom Floatind-Point. 72–74, 78–84, 87, 88, 136, 139

CNN Convolutional Neural Network. 56, 59, 63, 80, 93, 116, 119, 137

CPU Central Processing Unit. 10, 13, 27, 32, 39, 43, 65, 85, 86, 89, 97, 105, 121, 123, 124, 126, 127, 131, 132, 134, 148

CSP Central Signal Processor. 60

DAG Directed Acyclic Graph. 56

DALiuGE Data Activated 流 (Liu) Graph Engine. 53

DIF Dataflow Interchange Format. 53

DLF DLFloat. 32, 34, 35

DNN Deep Neural Network. 34, 56, 59, 116

DPN Dataflow Process Network. 45, 46, 48, 52, 53, 135

DPU Data Processing Unit. 42

DSE Design Space Exploration. 13, 51, 96, 97, 103, 105–108, 111, 115–120, 122, 123, 148

DSP Digital Signal Processor. 32, 53, 98, 102

DWT Discrete Wavelet Transform. 56–58, 63, 77, 101, 102, 119, 135, 137

EIDF Enable-Invoke Dataflow. 53

Fifo First-In First-Out queue. 45–48, 61, 66, 67, 77, 97–102, 105, 106, 118–120, 122, 123, 129, 130, 136–138

FF Flip-Flop. 97, 98, 101, 102

FFT Fast Fourier Transform. 30

FP IEEE-754 Floating-Point. 24–30, 33–35, 66, 68, 70, 71, 75, 135

FP128 IEEE-754 128-bit Quadruple-Precision Floating-Point. 25, 35, 69

FP16 IEEE-754 16-bit Half-Precision Floating-Point. 25–27, 32–36

FP256 IEEE-754 256-bit Octuple-Precision Floating-Point. 25, 35, 69

FP32 IEEE-754 32-bit Single-Precision Floating-Point. 25–27, 32–36, 65, 66, 68–72, 76–78, 80, 81, 87, 88, 92, 130–132, 136, 137, 139, 145

FP64 IEEE-754 64-bit Double-Precision Floating-Point. 25–27, 35, 65, 66, 69, 130

FPGA Field-Programmable Gate Array. 13, 31, 32, 34, 36, 42, 43, 85, 86, 97–103, 105, 106, 118, 120–124, 127, 134, 137, 138, 148

FPU Floating-Point Unit. 24–30, 33, 34, 36

FSM Finite-State Machine. 38

FxP Fixed-Point. 27, 29–31, 34, 66, 69–75, 78, 80–85, 87, 93, 96, 101, 117, 124, 136

GLEAM Galactic and Extra-galactic All-sky MWA. 61

GPGPU General-Purpose computing on Graphics Processing Unit. 42

GPT-3 Generative Pre-trained Transformer 3. 11

GPU Graphics Processing Unit. 10, 32, 33, 42–44, 124

GRT Global Runtime. 52

HDF Heterochronous Dataflow. 52

HLS High-Level Synthesis. 98, 100–102, 106, 124, 137, 138

HoCL Higher Order dataflow Coordination Language. 53

HPC High-Performance Computing. 9, 10, 55, 56, 60, 125, 148

IBSDF Interfaced-Based Synchronous Dataflow. 49

IDWT Inverse Discrete Wavelet Transform. 56, 57, 63, 77

IoT Internet of Things. 9, 125

KPN Kahn Process Network. 45

LHC Large Hadron Collider. 10, 11

LIDE DSPCAD Lightweight Dataflow Environment. 53

LLC Last Level Cache. 10

LNS Logarithmic Number System. 31, 123

LRT Local Runtime. 52

LSB Least Significant Bit. 19, 20, 29, 33, 66–68, 72, 87, 90, 92, 130, 131, 133

LUT Look-Up Table. 18, 19, 31, 98, 101, 102

MAC Multiply-accumulate. 23

MMU Memory Management Unit. 39

MoC Model of Computation. 12, 13, 37, 38, 45–53, 55, 86, 97, 100, 121, 123, 126, 127, 129, 135, 148

MPI Message Passing Interface. 39, 41

MSB Most Significant Bit. 19, 20, 24, 29, 33, 66–68, 92

MSFP Microsoft Floating-Point. 32, 34–36

MWA Murchison Widefield Array. 61, 142

`NaN` Not-a-Number. 25, 27, 31, 34

OS Operating System. 39, 40, 52

PE Processing Element. 10, 38, 39, 43, 44, 46–48, 53, 124, 126

PiMM Parameterized and Interfaced dataflow Meta-Model. 49, 135

PiSDF Parameterized and Interfaced Synchronous Dataflow. 45, 49–52, 61, 135

PREESM Parallel and Real-time Embedded Executives Scheduling Method. 13, 51–53, 61, 86, 90, 100, 101, 103, 111, 118, 121, 127, 134, 135, 137

PSDF Parameterized Synchronous Dataflow. 49

PSNR Peak Signal-to-Noise Ratio. 57, 77–79, 82–84, 92, 96, 97, 101, 102, 115–117, 119, 136

SDF Synchronous Dataflow. 37, 38, 45, 48–50, 52, 53, 86, 97, 100, 113, 120, 121, 123, 129, 134, 135, 137

SDP Science Data Processor. 53, 56, 60, 61, 81, 84, 96, 97, 117, 136

SEP SDP Evolutionary Pipeline. 56, 62, 63, 117, 136

SFC Sequential Function Chart. 38

SIMD Single Instruction Multiple Data. 20, 30, 43, 144

SKA Square Kilometre Array. 10, 11, 53, 56, 60, 61, 117

SLE Speculative Lock Elision. 23

SNR Signal-to-Noise Ratio. 23

SPIDER Synchronous Parameterized Interfaced Dataflow Embedded Runtime. 52

SrDAG Single-rate Directed Acyclic Graph. 61

SSE Streaming SIMD Extension. 20

TF32 TensorFloat-32. 32, 33, 35, 36

TPU Tensor Processing Unit. 30

UML Unified Modeling Language. 38

VHDL VHSIC Hardware Description Language. 49, 129

W-CDMA Wideband-Code Division Multiple Access. 22

Glossary

`bswap` A `bswap` operation reverses the byte order of a data, effectively converting little-endian values to big-endian format and vice-versa. 90, 91

binade A binade is the set of numbers in a binary IEEE-754 floating-point format that all have the same exponent. In other words, a binade is the interval $[2^E, 2^{E+1})$ for some value of E of the exponent. 25, 34, 73–75, 79

normal number are non-zero numbers that can be represented without a leading 0 in the significand. For FP32, normal numbers range from 2^{-126} to $2^{127} \times (2 - 2^{-23})$. 26, 74, 75

subnormal number are non-zero numbers that are represented with a leading 0 in the significand. For FP32, normal numbers range from $2^{-126} \times 2^{-23}$ to $2^{-126} \times (1 - 2^{-23})$. 25, 26, 33, 74

Personal Publications

[Hon+22] Alexandre Honorat, Thomas Bourgoin, Hugo Miomandre, Karol Desnos, Daniel Menard, and Jean-François Nezan, "Influence of Dataflow Graph Moldable Parameters on Optimization Criteria", *in*: *Design and Architecture for Signal and Image Processing (DASIP)*, ed. by Karol Desnos and Sergio Pertuz, Cham: Springer International Publishing, 2022, pp. 83–95 (cit. on p. 51).

[Mio+17] Hugo Miomandre, Julien Hascoët, Karol Desnos, Kevin Martin, Benoît Dupont de Dinechin, and Jean-François Nezan, *Demonstrating the SPIDER Runtime for Reconfigurable Dataflow Graphs Execution onto a DMA-based Manycore Processor*, IEEE International Workshop on Signal Processing Systems (SiPS), Poster, Oct. 2017, URL: https://hal.archives-ouvertes.fr/hal-01637300 (cit. on p. 52).

[Mio+18] Hugo Miomandre, Julien Hascoët, Karol Desnos, Kevin J. M. Martin, Benoît Dupont de Dinechin Kalray, and Jean-François Nezan, "Embedded Runtime for Reconfigurable Dataflow Graphs on Manycore Architectures", *in*: *Proceedings of the 9th Workshop and 7th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, PARMA-DITAM '18, Manchester, United Kingdom: Association for Computing Machinery, 2018, pp. 51–56, URL: https://doi.org/10.1145/3183767.3183780 (cit. on p. 52).

[Mio+20] Hugo Miomandre, Jean-François Nezan, Daniel Menard, Adam Campbell, Anthony Griffin, Seth Hall, and Andrew Ensor, "Approximate Buffers for Reducing Memory Requirements: Case study on SKA", *in*: *34th 2020 IEEE Workshop on Signal Processing Systems (SiPS)*, vol. 2020-October, IEEE Workshop on Signal Processing Systems, SiPS: Design and Implementation, Coimbra, Portugal: Institute of Electrical and Electronics Engineers Inc., Oct. 2020, p. 9195262, URL: https://hal.archives-ouvertes.fr/hal-02612369 (cit. on pp. 12, 66, 127).

[MNM22] Hugo Miomandre, Jean-François Nezan, and Daniel Ménard, "Design Space Exploration for Memory-Oriented Approximate Computing Techniques", *in*: *33rd IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Gothenburg, Sweden: Institute of Electrical and Electronics Engineers Inc., July 2022 (cit. on pp. 13, 106).

Bibliography

[13] *Programmable controllers - Part 3: Programming languages*, Standard, Geneva, CH: International Electrotechnical Commission, Feb. 2013 (cit. on p. 38).

[18] "IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7", *in*: *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)* (2018), pp. 1–3951 (cit. on p. 40).

[19] "IEEE Standard for Floating-Point Arithmetic", *in*: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (July 2019), pp. 1–84 (cit. on p. 24).

[AAB18] Eduard Ayguadé, Lluc Alvarez, and Fabio Banchelli, "OpenMP: what's inside the black box?", *in*: 2018 (cit. on p. 42).

[ACN14] Roberto Airoldi, Fabio Campi, and Jari Nurmi, "Approximate computing for complexity reduction in timing synchronization", *in*: *EURASIP Journal on Advances in Signal Processing* 2014.*1* (Oct. 2014), p. 155, URL: https://doi.org/10.1186/1687-6180-2014-155 (cit. on pp. 22, 128).

[Agr+19] Ankur Agrawal, Silvia Melitta Mueller, Bruce M. Fleischer, Xiao Sun, Naigang Wang, Jungwook Choi, and Kailash Gopalakrishnan, "DLFloat: A 16-b Floating Point Format Designed for Deep Learning Training and Inference", *in*: *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, June 2019, pp. 92–95 (cit. on p. 34).

[Aki+15] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al., "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing", *in*: (2015) (cit. on p. 56).

[All70] Frances E. Allen, "Control Flow Analysis", *in*: *SIGPLAN Not.* 5.7 (July 1970), pp. 1–19, URL: https://doi.org/10.1145/390013.808479 (cit. on p. 38).

[Arr+] Florian Arrestier, Karol Desnos, Julien Heulot, Alexandre Honorat, Daniel Ménard, Antoine Morvan, Jean-François Nezan, Maxime Pelcat, and Claudio Rubattu, "Generating Energy-optimized Adaptive Software on a Heterogeneous MPSoC with PREESM", *in*: () (cit. on p. 55).

[AT05] Gediminas Adomavicius and Alexander Tuzhilin, "Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions", *in*: *IEEE Transactions on Knowledge and Data Engineering* 17.6 (2005), pp. 734–749 (cit. on p. 10).

[Bar+21] Salvatore Barone, Marcello Traiola, Mario Barbareschi, and Alberto Bosio, "Multi-Objective Application-Driven Approximate Design Method", *in*: *IEEE Access* 9 (2021), pp. 86975–86993 (cit. on p. 106).

[Bar17] Benjamin Barrois, "Methods to evaluate accuracy-energy trade-off in operator-level approximate computing", Theses, Université Rennes 1, Dec. 2017, URL: https://tel.archives-ouvertes.fr/tel-01665015 (cit. on pp. 27, 28).

[Bas+21] Syed Muhammad Arsalan Bashir, Yi Wang, Mahrukh Khan, and Yilong Niu, *A Comprehensive Review of Deep Learning-based Single Image Super-resolution*, 2021, URL: https://arxiv.org/abs/2102.09351 (cit. on p. 11).

[BB01] Bishnupriya Bhattacharya and Shuvra Shikhar Bhattacharyya, "Parameterized dataflow modeling for DSP systems", *in*: *IEEE Transactions on Signal Processing* 49.10 (2001), pp. 2408–2421 (cit. on pp. 47, 49).

[Bha+11] Shuvra Shikhar Bhattacharyya, Johan Eker, Jörn W. Janneck, Christophe Lucarz, Marco Mattavelli, and Mickaël Raulet, "Overview of the MPEG Reconfigurable Video Coding Framework", *in*: *Journal of Signal Processing*

Systems 63.*2* (May 2011), pp. 251–263, URL: https://hal.archives-ouvertes.fr/hal-00407945 (cit. on pp. 49, 129).

[BMS22] Alberto Bosio, Daniel Ménard, and Olivier Sentieys, *Approximate Computing Techniques: From Component-to Application-Level*, Springer Nature, 2022 (cit. on pp. 17, 128).

[BOB91] Michael Blair, Sally Obenski, and Paula Bridickas, *Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia*, https://www.gao.gov/assets/imtec-92-26.pdf, 1991 (cit. on p. 27).

[Bon+22] Justine Bonnot, Alexandre Mercat, Erwan Nogues, and Daniel Ménard, "Approximate Computing at the Algorithmic Level", *in*: *Approximate Computing Techniques: From Component- to Application-Level*, ed. by Alberto Bosio, Daniel Ménard, and Olivier Sentieys, Cham: Springer International Publishing, 2022, pp. 109–142, URL: https://doi.org/10.1007/978-3-030-94705-7_5 (cit. on p. 18).

[Bon19] Justine Bonnot, "Analyse d'erreurs pour les systèmes utilisant des calculs approximés", Theses, INSA de Rennes, Oct. 2019, URL: https://tel.archives-ouvertes.fr/tel-03005325 (cit. on pp. 17, 129).

[Bra19] Henry E Brady, "The challenge of big data and data science", *in*: *Annual Review of Political Science* 22 (2019), pp. 297–323 (cit. on p. 10).

[Bro+20] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei, "Language Models are Few-Shot Learners", *in*: *Advances in Neural Information Processing Systems*, ed. by H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, vol. 33, Curran Associates, Inc., 2020, pp. 1877–1901 (cit. on p. 11).

[BSM17] Benjamin Barrois, Olivier Sentieys, and Daniel Ménard, "The Hidden Cost of Functional Approximation Against Careful Data Sizing – A Case Study", *in*: *Design, Automation & Test in Europe Conference & Exhibition (DATE*

2017), Lausanne, Switzerland, 2017, URL: https://hal.inria.fr/hal-01423147 (cit. on pp. 24, 128).

[Buc+01] Joseph Buck, Soonhoi Ha, Edward Ashford Lee, and David G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems", *in*: *Readings in Hardware/Software Co-Design*, USA: Kluwer Academic Publishers, 2001, pp. 527–543 (cit. on p. 52).

[Bur+22] Thomas Burd, Wilson Li, James Pistole, Srividhya Venkataraman, Michael McCabe, Timothy Johnson, James Vinh, Thomas Yiu, Mark Wasio, Hon-Hin Wong, Daryl Lieu, Jonathan White, Benjamin Munger, Joshua Lindner, Javin Olson, Steven Bakke, Jeshuah Sniderman, Carson Henrion, Russell Schreiber, Eric Busta, Brett Johnson, Tim Jackson, Aron Miller, Ryan Miller, Matthew Pickett, Aaron Horiuchi, Josef Dvorak, Sabeesh Balagangadharan, Sajeesh Ammikkallingal, and Pankaj Kumar, "Zen3: The AMD 2nd-Generation 7nm x86-64 Microprocessor Core", *in*: *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 65, 2022, pp. 1–3 (cit. on pp. 10, 126).

[Cas+20] Jorge Castro-Godínez, Julián Mateus-Vargas, Muhammad Shafique, and Jörg Henkel, "AxHLS: Design Space Exploration and High-Level Synthesis of Approximate Accelerators using Approximate Functional Units and Analytical Models", *in*: *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020, pp. 1–9 (cit. on p. 106).

[CDP22] Maxime Christ, Florent de Dinechin, and Frédéric Pétrot, "Low-precision logarithmic arithmetic for neural network accelerators", *in*: *ASAP 2022 - 33rd IEEE International Conference on Application-specific Systems, Architectures and Processors*, Gothenburg, Sweden, July 2022, URL: https://hal.inria.fr/hal-03684585 (cit. on p. 31).

[CH98] Chunsheng Cai and Peter de Boves Harrington, "Different Discrete Wavelet Transforms Applied to Denoising Analytical Data", *in*: *Journal of Chemical Information and Computer Sciences* 38.*6* (1998), pp. 1161–1170, eprint: https://doi.org/10.1021/ci980210j, URL: https://doi.org/10.1021/ci980210j (cit. on p. 57).

[Cho+22] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko,

Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel, *PaLM: Scaling Language Modeling with Pathways*, 2022, URL: https://arxiv.org/abs/2204.02311 (cit. on p. 11).

[CJV07] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas, *Using OpenMP: portable shared memory parallel programming*, MIT press, 2007 (cit. on p. 41).

[Dar+20] Bita Darvish Rouhani, Daniel Lo, Ritchie Zhao, Ming Liu, Jeremy Fowers, Kalin Ovtcharov, Anna Vinogradsky, Sarah Massengill, Lita Yang, Ray Bittner, Alessandro Forin, Haishan Zhu, Taesik Na, Prerak Patel, Shuai Che, Lok Chand Koppaka, XIA SONG, Subhojit Som, Kaustav Das, Saurabh T, Steve Reinhardt, Sitaram Lanka, Eric Chung, and Doug Burger, "Pushing the Limits of Narrow Precision Inferencing at Cloud Scale with Microsoft Floating Point", *in*: *Advances in Neural Information Processing Systems*, ed. by H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, vol. 33, Curran Associates, Inc., 2020, pp. 10271–10281, URL: https://proceedings.neurips.cc/paper/2020/file/747e32ab0fea7fbd2ad9ec03daa3f840-Paper.pdf (cit. on p. 34).

[DCD97] Eddy De Greef, Francky Catthoor, and Hugo De Man, "Array placement for storage size reduction in embedded multimedia systems", *in*: *Proceedings IEEE International Conference on Application-Specific Systems, Architectures and Processors*, 1997, pp. 66–75 (cit. on pp. 10, 126).

[DD07] Florent de Dinechin and Jérémie Detrey, "A Tool for Unbiased Comparison between Logarithmic and Floating-point Arithmetic", *in*: *Journal of Signal*

Processing Systems 49.*1* (2007), pp. 161–175, URL: https://hal-ens-lyon.archives-ouvertes.fr/ensl-00542212 (cit. on pp. 31, 123).

[De +21] Gustavo De Carvalho Bertoli, Lourenço Alves Pereira Júnior, Osamu Saotome, Aldri Luiz Dos Santos, Filipe Alves Neto Verri, Cesar Augusto Cavalheiro Marcondes, Sidnei Barbieri, Moises S. Rodrigues, and José Maria Parente De Oliveira, "An End-to-End Framework for Machine Learning-Based Network Intrusion Detection System", *in*: *IEEE Access* 9 (2021), pp. 106790–106805 (cit. on p. 11).

[Den+09] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei, "ImageNet: A large-scale hierarchical image database", *in*: *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255 (cit. on p. 58).

[Des+13] Karol Desnos, Maxime Pelcat, Jean-François Nezan, Shuvra Shikhar Bhattacharyya, and Slaheddine Aridhi, "PiMM: Parameterized and Interfaced Dataflow Meta-Model for MPSoCs Runtime Reconfiguration", *in*: *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, IEEE, 2013, pp. 41–48 (cit. on pp. 48, 49).

[Des+16] Karol Desnos, Maxime Pelcat, Jean-François Nezan, and Slaheddine Aridhi, "On Memory Reuse Between Inputs and Outputs of Dataflow Actors", *in*: *ACM Trans. Embed. Comput. Syst.* 15.*2* (Feb. 2016), URL: https://doi.org/10.1145/2871744 (cit. on p. 107).

[Des14] Karol Desnos, "Memory Study and Dataflow Representations for Rapid Prototyping of Signal Processing Applications on MPSoCs", 2014ISAR0004, PhD thesis, 2014, URL: http://www.theses.fr/2014ISAR0004/document (cit. on p. 45).

[Din+19] Florent de Dinechin, Luc Forget, Jean-Michel Muller, and Yohann Uguen, "Posits: the good, the bad and the ugly", working paper or preprint, Mar. 2019, URL: https://hal.inria.fr/hal-01959581 (cit. on p. 33).

[EJ03] Johan Eker and Jörn W. Janneck, "CAL language report: Specification of the CAL actor language", *in*: 2003 (cit. on pp. 49, 129).

[Eke+03] Johan Eker, Jörn W. Janneck, Edward Ashford Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong, "Taming heterogeneity - the Ptolemy approach", *in*: *Proceedings of the IEEE* 91.*1* (2003), pp. 127–144 (cit. on p. 52).

[EWT18] Jorge Echavarria, Stefan Wildermann, and Jürgen Teich, "AConFPGA: A Multiple-Output Boolean Function Approximation DSE Technique Targeting FPGAs", *in*: *2018 International Conference on Field-Programmable Technology (FPT)*, 2018, pp. 326–329 (cit. on p. 106).

[Fru+15] Fabio Frustaci, Mahmood Khayatzadeh, David Blaauw, Dennis Sylvester, and Massimo Alioto, "SRAM for Error-Tolerant Applications With Dynamic Energy-Quality Management in 28 nm CMOS", *in*: *IEEE Journal of Solid-State Circuits* 50.*5* (2015), pp. 1310–1323 (cit. on pp. 21, 128).

[Gho+12] Arkadeb Ghosal, Rhishikesh Limaye, Kaushik Ravindran, Stavros Tripakis, Ankita Prasad, Guoqiang Wang, Trung N. Tran, and Hugo Andrade, "Static Dataflow with Access Patterns: Semantics and Analysis", *in*: *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, San Francisco, California: Association for Computing Machinery, 2012, pp. 656–663, URL: https://doi.org/10.1145/2228360.2228479 (cit. on p. 97).

[GLL99] Alain Girault, Bilung Lee, and Edward Ashford Lee, "Hierarchical finite state machines with multiple concurrency models", *in*: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 18.*6* (1999), pp. 742–760 (cit. on p. 52).

[Gol91] David Goldberg, "What Every Computer Scientist Should Know about Floating-Point Arithmetic", *in*: *ACM Comput. Surv.* 23.*1* (Mar. 1991), pp. 5–48, URL: https://doi.org/10.1145/103162.103163 (cit. on p. 27).

[Gri+20] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu, "A survey of deep learning techniques for autonomous driving", *in*: *Journal of Field Robotics* 37.*3* (2020), pp. 362–386 (cit. on p. 11).

[Gup+11] Vaibhav Gupta, Debabrata Mohapatra, Sang Phill Park, Anand Raghunathan, and Kaushik Roy, "IMPACT: IMPrecise adders for low-power approximate computing", *in*: *IEEE/ACM International Symposium on Low Power Electronics and Design*, 2011, pp. 409–414 (cit. on pp. 20, 128).

[GY17] John Leroy Gustafson and Isaac Yonemoto, "Beating Floating Point at Its Own Game: Posit Arithmetic", *in*: *Supercomput. Front. Innov.: Int. J.* 4.*2* (June 2017), pp. 71–86, URL: https://doi.org/10.14529/jsfi170206 (cit. on pp. 33, 123).

[Has+15] Julien Hascoët, Jean-Francois Nezan, Andrew Ensor, and Benoît Dupont de Dinechin, "Implementation of a Fast Fourier transform algorithm onto a manycore processor", *in*: *2015 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, ISSN: null, Sept. 2015, pp. 1–7 (cit. on p. 30).

[Has+17] Julien Hascoët, Karol Desnos, Jean-François Nezan, and Benoît Dupont de Dinechin, "Hierarchical Dataflow Model for efficient programming of clustered manycore processors", *in*: *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2017, pp. 137–142 (cit. on p. 55).

[Has18] Julien Hascoët, "Contributions to Software Runtime for Clustered Manycores Applied to Embedded and High-Performance Applications", Theses, INSA de Rennes, Dec. 2018, URL: https://tel.archives-ouvertes.fr/tel-02132613 (cit. on p. 52).

[Hat+17] Sam Hatfield, Aneesh Subramanian, Tim Palmer, and Peter Düben, "Improving Weather Forecast Skill through Reduced-Precision Data Assimilation", *in*: *Monthly Weather Review* 146.*1* (Nov. 2017), pp. 49–62, URL: https://journals.ametsoc.org/doi/full/10.1175/MWR-D-17-0132.1 (visited on 10/02/2019) (cit. on pp. 27, 128).

[Heu+14] Julien Heulot, Maxime Pelcat, Karol Desnos, Jean-François Nezan, and Slaheddine Aridhi, "Spider: A Synchronous Parameterized and Interfaced Dataflow-based RTOS for multicore DSPS", *in*: *2014 6th European Embedded Design in Education and Research Conference (EDERC)*, Milan, Italy, Sept. 2014, pp. 167–171 (cit. on p. 52).

[Heu15] Julien Heulot, "Runtime multicore scheduling techniques for dispatching parameterized signal and vision dataflow applications on heterogeneous MPSoCs", PhD thesis, INSA de Rennes, 2015 (cit. on p. 51).

[HO13] Jie Han and Michael Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design", *in*: *2013 18th IEEE European Test Symposium (ETS)*, 2013, pp. 1–6 (cit. on pp. 12, 126).

[Hol+14] Simon Holmbacka, Erwan Nogues, Maxime Pelcat, Sébastien Lafond, and Johan Lilius, "Energy efficiency and performance management of parallel dataflow applications", *in*: *Proceedings of the 2014 Conference on Design and Architectures for Signal and Image Processing*, 2014, pp. 1–8 (cit. on p. 55).

[Hon20] Alexandre Honorat, "Modeling, Scheduling, Pipelining and Configuration of Synchronous Dataflow Graphs with Throughput Constraints", Theses, INSA de Rennes, Nov. 2020, URL: https://tel.archives-ouvertes.fr/tel-03337988 (cit. on p. 50).

[Hor14] Mark Horowitz, "1.1 Computing's energy problem (and what we can do about it)", *in*: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2014, pp. 10–14 (cit. on pp. 28, 65, 123).

[Hsu+04] Chia-Jui Hsu, Fuat Keceli, Ming-Yung Ko, Shahrooz Shahparnia, and Shuvra Shikhar Bhattacharyya, "DIF: An Interchange Format for Dataflow-Based Design Tools", *in*: vol. 3133, July 2004, pp. 423–432 (cit. on p. 53).

[Ian+16] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer, *SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size*, 2016, arXiv: 1602.07360 [cs.CV] (cit. on pp. 58, 80, 116).

[Joh97] Gary W. Johnson, *LabVIEW Graphical Programming: Practical Applications in Instrumentation and Control*, McGraw-Hill Visual Technology Series, McGraw-Hill, 1997, URL: https://books.google.fr/books?id=zQNlQgAACAAJ (cit. on p. 38).

[JWN10] Bruce Jacob, David T. Wang, and Spencer W. Ng, *Memory systems: cache, DRAM, disk*, Morgan Kaufmann, 2010 (cit. on pp. 9, 125).

[Kah74] Gilles Kahn, "The Semantics of a Simple Language for Parallel Programming", *in*: *Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974*, ed. by Jack L. Rosenfeld, North-Holland, 1974, pp. 471–475 (cit. on p. 45).

[KAK18] S. Karen Khatamifard, Ismail Akturk, and Ulya R. Karpuzcu, "On Approximate Speculative Lock Elision", *in*: *IEEE Transactions on Multi-Scale Computing Systems* 4.*2* (2018), pp. 141–151 (cit. on pp. 23, 128).

[KGE11] Parag Kulkarni, Puneet Gupta, and Milos Ercegovac, "Trading Accuracy for Power with an Underdesigned Multiplier Architecture", *in*: *2011 24th International Conference on VLSI Design*, 2011, pp. 346–351 (cit. on pp. 20, 128).

[Kir+21] B. Ravi Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A. Al Sallab, Senthil Yogamani, and Patrick Pérez, "Deep reinforcement learning for autonomous driving: A survey", *in*: *IEEE Transactions on Intelligent Transportation Systems* (2021) (cit. on p. 11).

[KKM16] Enagnon Cedric Klikpo, Jad Khatib, and Alix Munier-Kordon, "Modeling Multi-Periodic Simulink Systems by Synchronous Dataflow Graphs", *in*: *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016, pp. 1–10 (cit. on p. 38).

[Knu97] Donald Ervin Knuth, *The art of computer programming*, vol. 2, Pearson Education, 1997 (cit. on p. 26).

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton, "ImageNet Classification with Deep Convolutional Neural Networks", *in*: *Advances in Neural Information Processing Systems*, ed. by F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, vol. 25, Curran Associates, Inc., 2012, URL: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf (cit. on pp. 11, 58).

[KT21] Mikhail Khalilov and Alexey Timoveev, "Performance analysis of CUDA, OpenACC and OpenMP programming models on TESLA V100 GPU", *in*: *Journal of Physics: Conference Series*, vol. 1740, 1, IOP Publishing, 2021, p. 012056 (cit. on p. 45).

[Liv+07] Nikolaos Liveris, C. Lin, J. Wang, Hai Zhou, and Prith Banerjee, "Retiming for Synchronous Data Flow Graphs", *in*: *2007 Asia and South Pacific Design Automation Conference*, 2007, pp. 480–485 (cit. on p. 100).

[LM87] Edward Ashford Lee and David G. Messerschmitt, "Synchronous data flow", *in*: *Proceedings of the IEEE* 75.*9* (Sept. 1987), pp. 1235–1245 (cit. on pp. 48, 113).

[LP95] Edward Ashford Lee and Thomas M. Parks, "Dataflow process networks", *in*: *Proceedings of the IEEE* 83.*5* (1995), pp. 773–801 (cit. on p. 45).

[Mah+10] Hamid Reza Mahdiani, Ali-Akbar Ahmadi, Sied Mehdi Fakhraie, and Caro Lucas, "Bio-Inspired Imprecise Computational Blocks for Efficient VLSI Implementation of Soft-Computing Applications", *in*: *IEEE Transactions on Circuits and Systems I: Regular Papers* 57.*4* (2010), pp. 850–862 (cit. on pp. 20, 128).

[Mat97] MathWorks, Inc, *SIMULINK Dynamic System Simulation for MATLAB: Modeling, Simulation, Implementation*, SIMULINK Dynamic System Simulation for MATLAB: Modeling, Simulation, Implementation, MathWorks, Incorporated, 1997, URL: https://books.google.fr/books?id=HxOWyAEACAAJ (cit. on p. 38).

[Meh+09] Pramod Kumar Meher, Javier Valls, Tso-Bing Juang, K. Sridharan, and Koushik Maharatna, "50 Years of CORDIC: Algorithms, Architectures, and Applications", *in*: *IEEE Transactions on Circuits and Systems I: Regular Papers* 56.*9* (2009), pp. 1893–1907 (cit. on pp. 18, 128).

[Mic+22] Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, Naveen Mellempudi, Stuart Oberman, Mohammad Shoeybi, Michael Siu, and Hao Wu, *FP8 Formats for Deep Learning*, 2022, URL: https://arxiv.org/abs/2209.05433 (cit. on p. 33).

[Mig+15] Joshua San Miguel, Jorge Albericio, Andreas Moshovos, and Natalie Enright Jerger, "Doppelgänger: A cache for approximate computing", *in*: *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 50–61 (cit. on pp. 21, 128).

[Mit16] Sparsh Mittal, "A survey of techniques for approximate computing", *in*: *ACM Computing Surveys (CSUR)* 48.*4* (2016), pp. 1–33 (cit. on pp. 17, 129).

[Mor+21] Chamin Morikawa, Michihiro Kobayashi, Masaki Satoh, Yasuhiro Kuroda, Teppei Inomata, Hitoshi Matsuo, Takeshi Miura, and Masaki Hilaga, "Image and video processing on mobile devices: a survey", *in*: *The Visual Computer* 37.*12* (2021), pp. 2931–2949 (cit. on p. 10).

[MS13] Raj Mishra and Amit Shrivastava, “Implementation of Custom Precision Floating Point Arithmetic on FPGAs”, *in*: *HCTL Open International Journal of Technology Innovations and Research (IJTIR)* Volume 1, January 2013 (Jan. 2013), pp. 10–26 (cit. on p. 33).

[Mul+18] Jean-Michel Muller, Nicolas Brisebarre, Florent De Dinechin, Claude-Pierre Jeannerod, Vincent Lefevre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, Serge Torres, et al., *Handbook of floating-point arithmetic*, Springer, 2018 (cit. on p. 24).

[Mul20] Jean-Michel Muller, “Elementary Functions and Approximate Computing”, *in*: *Proceedings of the IEEE* 108.*12* (2020), pp. 2136–2149 (cit. on p. 19).

[Mut+19] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun, “Processing data where it makes sense: Enabling in-memory computation”, *in*: *Microprocessors and Microsystems* 67 (2019), pp. 28–41, URL: https://www.sciencedirect.com/science/article/pii/S0141933118302291 (cit. on p. 123).

[Mut+23] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun, “A Modern Primer on Processing in Memory”, *in*: *Emerging Computing: From Devices to Systems: Looking Beyond Moore and Von Neumann*, ed. by Mohamed M. Sabry Aly and Anupam Chattopadhyay, Singapore: Springer Nature Singapore, 2023, pp. 171–243, URL: https://doi.org/10.1007/978-981-16-7487-7_7 (cit. on p. 65).

[NL04] Stephen Neuendorffer and Edward Ashford Lee, “Hierarchical reconfiguration of dataflow models”, *in*: *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE ’04.* 2004, pp. 179–188 (cit. on p. 47).

[NMP16] Erwan Nogues, Daniel Menard, and Maxime Pelcat, “Algorithmic-level approximate computing applied to energy efficient hevc decoding”, *in*: *IEEE Transactions on Emerging Topics in Computing* 7.*1* (2016), pp. 5–17 (cit. on p. 20).

[Nvi20] Nvidia, *NVIDIA A100 Tensor Core GPU Architecture Whitepaper*, https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf, 2020 (cit. on p. 33).

[Nvi22a] Nvidia, *NVIDIA Ada GPU Architecture Whitepaper*, https://images.nvidia.com/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf, 2022 (cit. on p. 33).

[Nvi22b] Nvidia, *NVIDIA H100 Tensor Core GPU Architecture Whitepaper*, https://nvdam.widen.net/s/9bz6dw7dqr/gtc22-whitepaper-hopper, 2022 (cit. on p. 33).

[OMK18] Daniel W. Otter, Julian Richard Medina, and Jugal Kumar Kalita, "A Survey of the Usages of Deep Learning in Natural Language Processing", *in*: *CoRR* abs/1807.10854 (2018), arXiv: 1807.10854, URL: http://arxiv.org/abs/1807.10854 (cit. on p. 11).

[Pai+20] Guilherme Paim, Leandro Mateus Giacomini Rocha, Hussam Amrouch, Eduardo Antônio César da Costa, Sergio Bampi, and Jörg Henkel, "A Cross-Layer Gate-Level-to-Application Co-Simulation for Design Space Exploration of Approximate Circuits in HEVC Video Encoders", *in*: *IEEE Transactions on Circuits and Systems for Video Technology* 30.*10* (2020), pp. 3814–3828 (cit. on p. 106).

[PBR09] Jonathan Piat, Shuvra Shikhar Bhattacharyya, and Mickaël Raulet, "Interface-based hierarchy for synchronous data-flow graphs", *in*: *2009 IEEE Workshop on Signal Processing Systems*, Tampere, Finland, 2009, pp. 145–150 (cit. on pp. 47, 49).

[Pel+09] Maxime Pelcat, Jean François Nezan, Jonathan Piat, Jerome Croizer, and Slaheddine Aridhi, "A System-Level Architecture Model for Rapid Prototyping of Heterogeneous Multicore Embedded Systems", *in*: *Conference on Design and Architectures for Signal and Image Processing (DASIP) 2009*, nice, France, Sept. 2009, 8 pages, URL: https://hal.archives-ouvertes.fr/hal-00429397 (cit. on p. 51).

[Pel+12] Maxime Pelcat, Slaheddine Aridhi, Jonathan Piat, and Jean François Nezan, *Physical Layer Multi-Core Prototyping: A Dataflow-Based Approach for LTE eNodeB*, Lecture Notes in Electrical Engineering, Springer-Verlag London, Aug. 2012, p. 224 (cit. on p. 55).

[Pel+14] Maxime Pelcat, Karol Desnos, Julien Heulot, Clément Guy, Jean-François Nezan, and Slaheddine Aridhi, "Preesm: A dataflow-based rapid prototyping framework for simplifying multicore DSP programming", *in*: *Education and Research Conference (EDERC), 2014 6th European Embedded Design in*, Sept. 2014, pp. 36–40 (cit. on p. 51).

[Pli+08] William Plishker, Nimish Sane, Mary Kiemb, Kapil Anand, and Shuvra Shikhar Bhattacharyya, "Functional DIF for Rapid Prototyping", *in*: *2008 The 19th IEEE/IFIP International Symposium on Rapid System Prototyping*, 2008, pp. 17–23 (cit. on p. 52).

[Rae+21] Jack W. Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, H. Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, Eliza Rutherford, Tom Hennigan, Jacob Menick, Albin Cassirer, Richard Powell, George van den Driessche, Lisa Anne Hendricks, Maribeth Rauh, Po-Sen Huang, Amelia Glaese, Johannes Welbl, Sumanth Dathathri, Saffron Huang, Jonathan Uesato, John Mellor, Irina Higgins, Antonia Creswell, Nat McAleese, Amy Wu, Erich Elsen, Siddhant M. Jayakumar, Elena Buchatskaya, David Budden, Esme Sutherland, Karen Simonyan, Michela Paganini, Laurent Sifre, Lena Martens, Xiang Lorraine Li, Adhiguna Kuncoro, Aida Nematzadeh, Elena Gribovskaya, Domenic Donato, Angeliki Lazaridou, Arthur Mensch, Jean-Baptiste Lespiau, Maria Tsimpoukelli, Nikolai Grigorev, Doug Fritz, Thibault Sottiaux, Mantas Pajarskas, Toby Pohlen, Zhitao Gong, Daniel Toyama, Cyprien de Masson d'Autume, Yujia Li, Tayfun Terzi, Vladimir Mikulik, Igor Babuschkin, Aidan Clark, Diego de Las Casas, Aurelia Guy, Chris Jones, James Bradbury, Matthew Johnson, Blake A. Hechtman, Laura Weidinger, Iason Gabriel, William S. Isaac, Edward Lockhart, Simon Osindero, Laura Rimell, Chris Dyer, Oriol Vinyals, Kareem Ayoub, Jeff Stanway, Lorrayne Bennett, Demis Hassabis, Koray Kavukcuoglu, and Geoffrey Irving, "Scaling Language Models: Methods, Analysis & Insights from Training Gopher", *in*: *CoRR* abs/2112.11446 (2021), arXiv: 2112.11446, URL: https://arxiv.org/abs/2112.11446 (cit. on p. 11).

[Rah+14] Amir Rahmati, Matthew Hicks, Daniel E. Holcomb, and Kevin Fu, "Refreshing thoughts on DRAM: Power saving vs. data integrity", *in*: *Workshop on*

Approximate Computing Across the System Stack (WACAS), vol. 44, 2014 (cit. on pp. 21, 128).

[Ram+21] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever, *Zero-Shot Text-to-Image Generation*, 2021, URL: https://arxiv.org/abs/2102.12092 (cit. on p. 11).

[Ren+12] Lakshminarayanan Renganarayana, Vijayalakshmi Srinivasan, Ravi Nair, and Daniel Prener, "Programming with Relaxed Synchronization", *in*: *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability*, RACES '12, Tucson, Arizona, USA: Association for Computing Machinery, 2012, pp. 41–50, URL: https://doi.org/10.1145/2414729.2414737 (cit. on pp. 23, 128).

[Res+20] Diana Resmerita, Rodrigo Cabral Farias, Benoît Dupont de Dinechin, and Lionel Fillatre, "Benchmarking Alternative Floating-Point Formats for Deep Learning Inference", *in*: *COMPAS*, Lyon, France, 2020, URL: https://hal.archives-ouvertes.fr/hal-03625485 (cit. on p. 34).

[RG01] Ravi Rajwar and James R. Goodman, "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution", *in*: *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, Austin, Texas: IEEE Computer Society, 2001, pp. 294–305 (cit. on pp. 23, 128).

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch, *Unified Modeling Language Reference Manual, The (2nd Edition)*, Pearson Higher Education, 2004 (cit. on p. 38).

[Rod08] Ohad Rodeh, "B-Trees, Shadowing, and Clones", *in*: *ACM Trans. Storage* 3.4 (Feb. 2008), URL: https://doi.org/10.1145/1326542.1326544 (cit. on p. 39).

[Rom+21] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer, *High-Resolution Image Synthesis with Latent Diffusion Models*, 2021, arXiv: 2112.10752 [cs.CV] (cit. on p. 11).

[Sah+22] Chitwan Saharia, William Chan, Saurabh Saxena, Lala Li, Jay Whang, Emily Denton, Seyed Kamyar Seyed Ghasemipour, Burcu Karagol Ayan, S. Sara Mahdavi, Rapha Gontijo Lopes, Tim Salimans, Jonathan Ho, David J. Fleet,

and Mohammad Norouzi, *Photorealistic Text-to-Image Diffusion Models with Deep Language Understanding*, 2022, URL: https://arxiv.org/abs/2205.11487 (cit. on p. 11).

[Sav97] John E. Savage, *Models of Computation: Exploring the Power of Computing*, 1st, USA: Addison-Wesley Longman Publishing Co., Inc., 1997 (cit. on p. 45).

[SB18] Sundararajan Sriram and Shuvra Shikhar Bhattacharyya, *Embedded multiprocessors: Scheduling and synchronization*, CRC press, 2018 (cit. on p. 55).

[SCE01] Athanassios Skodras, Charilaos A. Christopoulos, and Touradj Ebrahimi, "The JPEG 2000 still image compression standard", *in*: *IEEE Signal Processing Magazine* 18.5 (2001), pp. 36–58 (cit. on p. 57).

[Sér20] Jocelyn Sérot, "HoCL: High Level Specification of Dataflow Graphs", *in*: *Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages*, IFL 2020, Canterbury, United Kingdom: Association for Computing Machinery, 2020, pp. 11–22, URL: https://doi.org/10.1145/3462172.3462185 (cit. on p. 53).

[She+11] Chung-Ching Shen, Lai-Huei Wang, Inkeun Cho, Scott Kim, Stephen Won, William Plishker, and Shuvra Shikhar Bhattacharyya, *The DSPCAD lightweight dataflow environment: Introduction to LIDE version 0.1*, tech. rep., 2011 (cit. on p. 52).

[Sid+11] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard, "Managing Performance vs. Accuracy Trade-Offs with Loop Perforation", *in*: ESEC/FSE '11, Szeged, Hungary: Association for Computing Machinery, 2011, pp. 124–134, URL: https://doi.org/10.1145/2025113.2025133 (cit. on pp. 18, 128).

[Smi+22] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, Elton Zheng, Rewon Child, Reza Yazdani Aminabadi, Julie Bernauer, Xia Song, Mohammad Shoeybi, Yuxiong He, Michael Houston, Saurabh Tiwary, and Bryan Catanzaro, "Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model", *in*: *CoRR* abs/2201.11990 (2022), arXiv: 2201.11990, URL: https://arxiv.org/abs/2201.11990 (cit. on p. 11).

[Smo+13] Vadim Smolyakov, Glenn Gulak, Timothy Gallagher, and Curtis Ling, "Fault-Tolerant Embedded-Memory Strategy for Baseband Signal Processing Systems", *in*: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21.7 (2013), pp. 1299–1307 (cit. on pp. 21, 128).

[Sou+22] Nicolas Sourbier, Karol Desnos, Thomas Guyet, Frederic Majorczyk, Olivier Gesny, and Maxime Pelcat, "SECURE-GEGELATI Always-On Intrusion Detection through GEGELATI Lightweight Tangled Program Graphs", *in*: *Journal of Signal Processing Systems* (2022), pp. 1–18 (cit. on p. 11).

[Su+12] Ching-Lung Su, Po-Yu Chen, Chun-Chieh Lan, Long-Sheng Huang, and Kuo-Hsuan Wu, "Overview and comparison of OpenCL and CUDA technology for GPGPU", *in*: *2012 IEEE Asia Pacific Conference on Circuits and Systems*, 2012, pp. 448–451 (cit. on p. 44).

[Sur+15] Arjun Suresh, Bharath Narasimha Swamy, Erven Rohou, and André Seznec, "Intercepting Functions for Memoization: A Case Study Using Transcendental Functions", *in*: *ACM Transactions on Architecture and Code Optimization* 12.2 (July 2015), p. 23, URL: https://hal.inria.fr/hal-01178085 (cit. on pp. 19, 128).

[Sur+19] Leonardo Suriano, Florian Arrestier, Alfonso Rodríguez, Julien Heulot, Karol Desnos, Maxime Pelcat, and Eduardo de la Torre, "DAMHSE: Programming heterogeneous MPSoCs with hardware acceleration using dataflow-based design space exploration and automated rapid prototyping", *in*: *Microprocessors and Microsystems* 71 (2019), p. 102882, URL: https://www.sciencedirect.com/science/article/pii/S0141933118303107 (cit. on p. 55).

[Tin+19] Oriol Tintó Prims, Mario César Acosta Cobos, Andrew M. Moore, Miguel Castrillo, Kim Serradell, Ana Cortés, and Francisco J. Doblas-Reyes, "How to use mixed precision in ocean models: exploring a potential reduction of numerical precision in NEMO 4.0 and ROMS 3.6", *in*: *Geoscientific Model Development* 12.7 (2019), pp. 3135–3148, URL: https://www.geosci-model-dev.net/12/3135/2019/ (cit. on pp. 27, 128).

[Tis+20] Alexandre Tissier, Wassim Hamidouche, Jarno Vanne, Franck Galpin, and Daniel Menard, "CNN oriented complexity reduction of VVC intra encoder", *in*: *2020 IEEE International Conference on Image Processing (ICIP)*, IEEE, 2020, pp. 3139–3143 (cit. on p. 11).

[Vas+15] Vassilis Vassiliadis, Konstantinos Parasyris, Charalambos Chalios, Christos D. Antonopoulos, Spyros Lalis, Nikolaos Bellas, Hans Vandierendonck, and Dimitrios S. Nikolopoulos, "A Programming Model and Runtime System for Significance-Aware Energy-Efficient Computing", *in*: *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, San Francisco, CA, USA: Association for Computing Machinery, 2015, pp. 275–276, URL: https://doi.org/10.1145/2688500.2688546 (cit. on pp. 18, 128).

[Vlu+19] Steven van der Vlugt, Hadi Alizadeh Ara, Rob de Jong, Martijn Hendriks, Ruben Guerra Marin, Marc Geilen, and Dip Goswami, "Modeling and analysis of FPGA accelerators for real-time streaming video processing in the healthcare domain", *in*: *Journal of Signal Processing Systems* 91.*1* (2019), pp. 75–91 (cit. on p. 97).

[WM95] William Allan Wulf and Sally A. McKee, "Hitting the memory wall: Implications of the obvious", *in*: *ACM SIGARCH computer architecture news* 23.*1* (1995), pp. 20–24 (cit. on pp. 9, 125).

[Won+18] Kanit Wongsuphasawat, Daniel Smilkov, James Wexler, Jimbo Wilson, Dandelion Mané, Doug Fritz, Dilip Krishnan, Fernanda B. Viégas, and Martin Wattenberg, "Visualizing Dataflow Graphs of Deep Learning Models in TensorFlow", *in*: *IEEE Transactions on Visualization and Computer Graphics* 24.*1* (2018), pp. 1–12 (cit. on p. 55).

[Wu+17] Chen Wu, Rodrigo Tobar, Kevin Vinsen, Andreas Wicenec, Dave Pallot, Baoqiang Lao, Ruonan Wang, Tao An, Mark Boulton, Ian Cooper, Richard Dodson, Markus Dolensky, Ying Mei, and Feng Wang, *DALiuGE: A Graph Execution Framework for Harnessing the Astronomical Data Deluge*, 2017, URL: https://arxiv.org/abs/1702.07617 (cit. on p. 53).

[xAn18] xAndru!, *Wii VC Round-to-Zero*, https://ukikipedia.net/wiki/Wii_VC_Round-to-Zero, 2018 (cit. on p. 26).

[XMK15] Qiang Xu, Todd Mytkowicz, and Nam Sung Kim, "Approximate computing: A survey", *in*: *IEEE Design & Test* 33.*1* (2015), pp. 8–22 (cit. on pp. 17, 129).

[Yu+22] Jiahui Yu, Yuanzhong Xu, Jing Yu Koh, Thang Luong, Gunjan Baid, Zirui Wang, Vijay Vasudevan, Alexander Ku, Yinfei Yang, Burcu Karagol Ayan, Ben Hutchinson, Wei Han, Zarana Parekh, Xin Li, Han Zhang, Jason Baldridge, and Yonghui Wu, *Scaling Autoregressive Models for Content-Rich Text-to-Image Generation*, 2022, URL: https://arxiv.org/abs/2206.10789 (cit. on p. 11).

[ZC20] Feichi Zhou and Yang Chai, "Near-sensor and in-sensor computing", *in*: *Nature Electronics* 3.*11* (2020), pp. 664–671 (cit. on pp. 10, 55).

[Zha+14] Qian Zhang, Feng Yuan, Rong Ye, and Qiang Xu, "ApproxIt: An approximate computing framework for iterative methods", *in*: *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2014, pp. 1–6 (cit. on pp. 18, 128).

[Zho+13] Zheng Zhou, Karol Desnos, Maxime Pelcat, Jean-François Nezan, William Plishker, and Shuvra Shikhar Bhattacharyya, "Scheduling of parallelized synchronous dataflow actors", *in*: *2013 International Symposium on System on Chip (SoC)*, 2013, pp. 1–10 (cit. on p. 46).

[Ziv+17] Darko Zivanovic, Milan Pavlovic, Milan Radulovic, Hyunsung Shin, Jongpil Son, Sally A. Mckee, Paul M. Carpenter, Petar Radojković, and Eduard Ayguadé, "Main Memory in HPC: Do We Need More or Could We Live with Less?", *in*: *ACM Transactions on Architecture and Code Optimization* 14.*1* (Mar. 2017), URL: https://doi.org/10.1145/3023362 (cit. on pp. 9, 125).

DOCTORAT BRETAGNE LOIRE / MATHSTIC

INSA RENNES

Titre : Méthodes de Réduction de Ressources Matérielles Basées Calcul Approximé pour Systèmes Hétérogènes.

Mot clés : Calcul Approximé, Graphe Flux-de-Données, Exploration d'Espace de Conception, FPGA

Résumé : Les applications de traitement du numérique signal et du multimédia traitent un volume de donnée toujours plus gros. Le monde numérique est responsable de 4% des émissions de gaz à effet de serre et 10% de la consommation d'électricité en 2021. Les mémoires occupant jusqu'à 80% de la surface de silicium des puces, et des mémoires externes étant souvent nécessaire pour étendre les capacités de stockage internes, les mémoires sont responsable d'une majeure partie de la consommation d'énergie aussi bien pour l'informatique en périphérie de réseau que des systèmes de calcul haute performance.

Des méthodes basées sur le calcul approximé ont été développées permettant de concéder le niveau de qualité final au profit des ressources matérielles. L'utilisation de modèles de calcul flux-de-données peut faciliter la mise en œuvre de ces techniques de calcul approximé.

Cette thèse présente une méthode de calcul approximé pour réduire les besoins en ressources matérielles liés au stockage de données en mémoire, ainsi qu'un algorithme d'exploration d'espace de conception pour faciliter sa mise en œuvre. La méthode proposée consiste à empaqueter les données en mémoire en utilisant des représentations alternatives, et est utilisable avec des processeurs conventionnels ainsi qu'avec des FPGA. Ces contributions ont en partie été implémentées dans le logiciel open-source PREESM.

Title: Approximated Computing-based Methods for Hardware Resources Reduction Targeting Heterogeneous Systems.

Keywords: Approximate Computing, Dataflow Graph, Design Space Exploration, FPGA

Abstract: Digital signal processing and multimedia applications have to compute an ever increasing amount of data. The digital world accounted for 4% of greenhouse gases and 10% of the electricity consumption in 2021. As the silicon area occupied by the memory can be as large as 80% of a chip and external memories are often required to extend these internal memories, memories are responsible for the major part of the power consumption of both Edge Computing and High-Performance Computing (HPC) systems.

Methods based on Approximate Computing (AxC) have been developped by trading-off output accuracy for reduced resources requirements. The use of Dataflow Models of Computation (MoCs) can facilitate the implementation of AxC techniques.

This thesis presents an AxC-based method for reducing the hardware resources requirements for data storage in memory, as well as a Design Space Exploration (DSE) algorithm to facilitate its implementation. The proposed method consists in packing data in memory using alternative representations and is usable on both conventional Central Processing Units (CPUs) as well as on Field-Programmable Gate Arrays (FPGAs). These contributions have in part been implemented into the open-source PREESM framework.